ISSN:2184-044X ISBN:978-989-54312-6-7 © 2019
DOI: 10.36315/2019v2end077

USING ARTISTIC ILLUSTRATION TO COMMUNICATE ABSTRACT AND INVISIBLE IDEAS IN THE SOFTWARE ENGINEERING DOMAIN

David Cutting[1], Andrew McDowell[1], TJ Cosgrove[2], Neil Anderson[1], Matthew Collins[1], & Paul Sage[1]
[1]Dr., School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast (United Kingdom)
[2]School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast (United Kingdom)

Abstract

Computer software is invisible and the discipline of software engineering includes a number of complex abstract concepts which are near impossible to visualize. We educate learners in a variety of ways using metaphor and comparative example but often people need to see the problem before they can understand even the need for a solution. To this end a set of illustrations have been created using the metaphor of bridge engineering to help beginners in software engineering to understand the fundamental problems of coping with changing requirements and evolving systems. These, along with the narrative they describe, are presented as an example of one possible approach to making software engineering more accessible.

Keywords: *Software engineering, art, illustration, science communication.*

1. Introduction

Computer science, and specifically software engineering, is full of complex abstract concepts. Computer software has no physical form and other than perhaps any user interface to a system, there is no visibility to the scaffold and structure that lies behind. These concepts then are hard not just to comprehend in operation but very hard to understand in terms of the fundamental functions they perform. If one cannot see the problem then even the need for a solution is not apparent, let alone what shape or form a solution may take. The aim of software engineering is specifically focused on these abstract invisible ideas; how elements within the software relate and communicate with each other, how they are layered and structured, and how best can we organize our solutions to be both efficient and flexible for future evolution.

Education addresses this in a number of different ways including letting learners experience problems first-hand before introducing software engineering concepts as a way to make their lives easier and more productive. This can work well with an audience of expert learners, those that have a good understanding of core concepts and, critically, wish to become better and more effective programmers. But such an approach can fail to engage a wider audience who often have little or no vested interest in software engineering as a discipline, for example groups such as natural science students who are learning programming but only as a means to an end. The challenge is how to communicate the need for and purpose of software engineering techniques to such cohorts.

One possible approach is through using metaphor and comparison with traditional physical engineering problems – how could one engineer a vehicle to meet emerging and flexible requirements or the relative ease with which a new house is built compared with extending an existing one and having to fit within the existing constraints for example. Our technique has been to pair such metaphors with illustrative line-art created by local artists, forming graphic panels which can be used together to form a storyboard to describe a particular concept. The initial work focused on maintainability in software systems and how poor early decisions with no consideration of future evolution can lead to highly unstable and risky systems after organic growth. Using the metaphor of bridges, these show design evolutions made to fit emerging problems without though of further changes and the end result. In the next section we present these illustrations and storyboard as an example of how such an approach can be used.

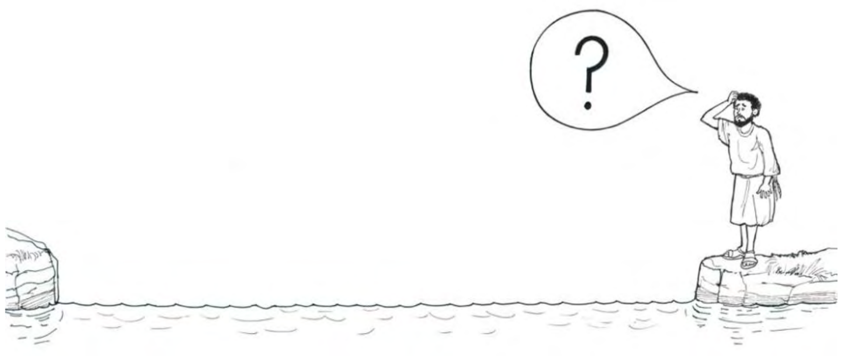
2. Software engineering and building bridges – an example application

Software engineering is about, in addition to other concerns, ensuring that the systems we build are flexible enough to be evolved in the future. Few engineering solutions are ever finished, the requirements evolve over time and this is as true in software as it is in physical disciplines, perhaps even more so. However, our common approach to software as an invisible intangible entity is to solve quickly just the problem in front of us, with little or no regard for how we may need to evolve in the future.

To help understand this let's try and consider the following: what if we built bridges the same way we so often build software?

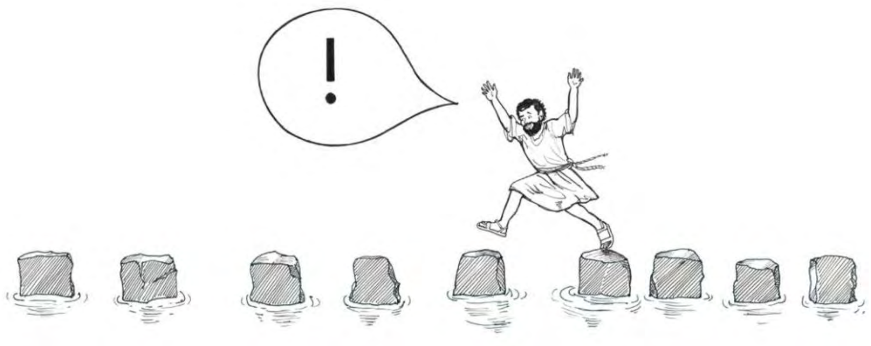
First, we start with an initial problem many years ago, we need to cross the river (Figure 1).

Figure 1. Our initial problem – how to cross the river?



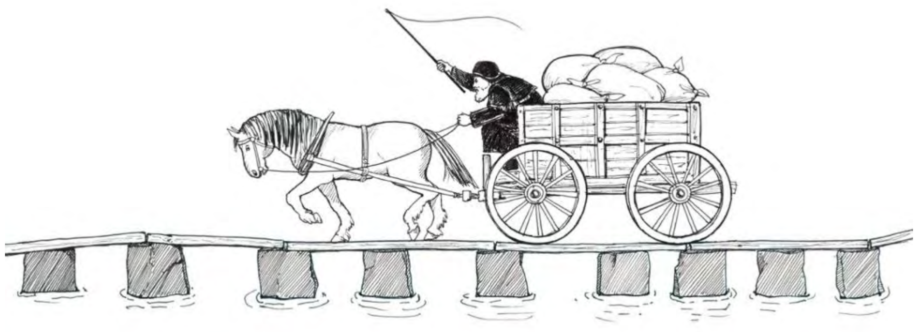
We solve this simply and immediately, dumping stones into the river means we can easily jump from one to the next and cross (Figure 2).

Figure 2.



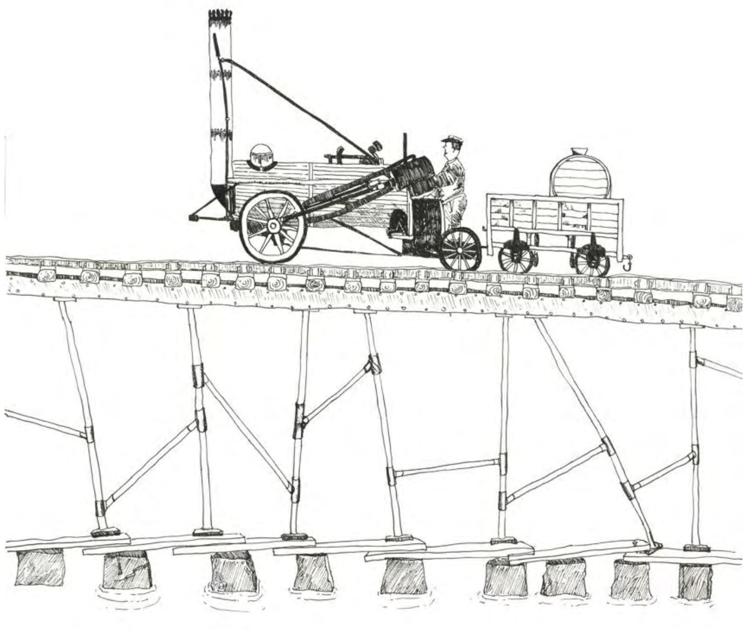
Time goes on and the technical abilities of our race evolve. We invent the wheel and immediately try and attach it to animals. Several unimpressed chickens, pigs, and one angry crow later we try the horse and jackpot – the cart is born. But the cart can't cross the river, it needs a platform over which to travel rather than jumping from stone to stone. Our requirements have changed, the situation has evolved and so must our solution. We fix the problem and lay planks over the stones so now people and carts can cross (Figure 3). We can get to market and buy bronze, or spears, or something.

Figure 3. A solution evolved – carts and people.



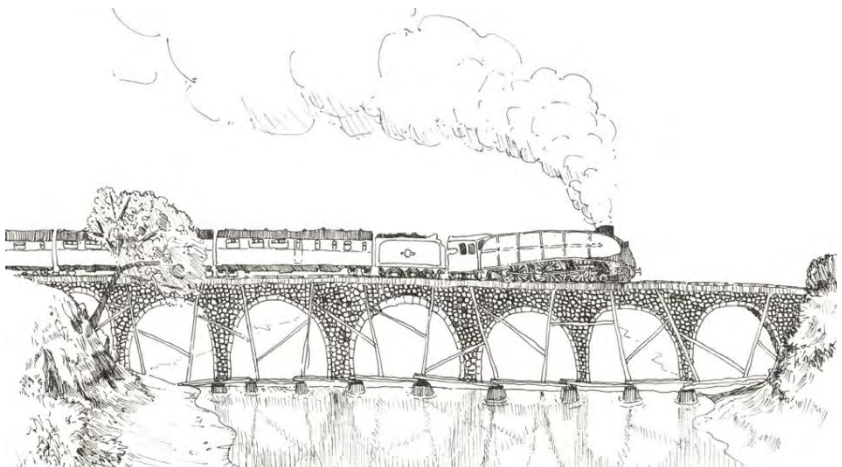
This solution works wonders. The wheel is voted “best thing before sliced bread” four years in a row and horses, carts, and people cross the bridge back and forth. But progress continues, someone has the brilliant idea of building a steam powered cart on rails and, much to the annoyance of the horses’ union the train is born. The train is much heavier than the cart, it can’t go up and down banks, so again the requirements change. Our evolved solution? Raise the deck, lay the rails, building on top of the planks which in turn are on the stepping stones (Figure 4).

Figure 4. Steam powered brilliance with our new raised deck featuring rails.



The train is, if anything, even more successful than the cart if only for the number of murder mysteries which can be staged on them. Trains evolve, they become more powerful, faster, and heavier. To meet these new requirements we again evolve, we use stonework to reinforce in increasing volumes as each new train comes out (Figure 5).

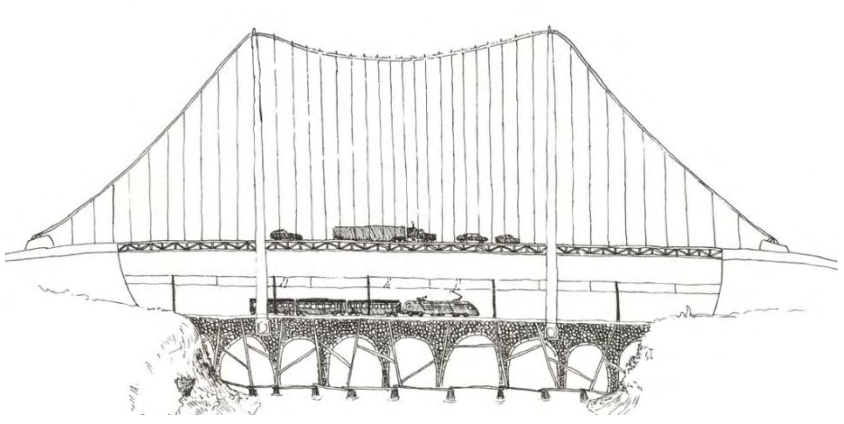
Figure 5. The stone reinforced bridge for high speed trains.



Innovation continues apace and the internal combustion engine is born and put into its own little chariot, the motor car is here. Cars don’t run on rails, that would be too sensible, and cars and trains it turns out don’t mix well. Trains have a habit of vengefully crushing all cars before them, but cars and trains need to cross the river.

Our solution? Fix the problem in front of us, building up from the reinforced train bridge put in another deck and allow cars then lorries to pass unhindered overhead (Figure 6).

Figure 6. Cars and trains all crossing in harmony together.



So now we have a working solution, a bridge allowing the free passage of all sorts over the river, day and night, problem free. We have a car deck supported off the train bridge. The train bridge is stonework put on to reinforce wooden posts and supports used to raise the deck from the river level. These supports sit on the planks underneath which are held up by the original stepping stones put into the river by a crossing engineer long since gone. The system works, the system is complex, the system has evolved. We still need to maintain the bridge and one day our brave, present day, engineer spots a loose cable and begins to tighten (Figure 7). He does this unaware of exactly what it connects to, what connects to what, which components rely on each other, and how the overall system is structured. Figure 8.

Figure 7. The engineer and a ratchet.

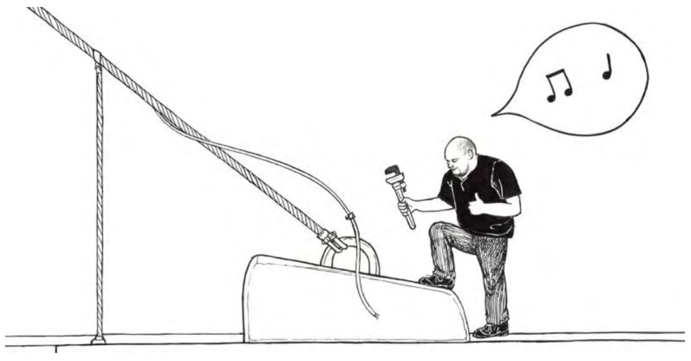
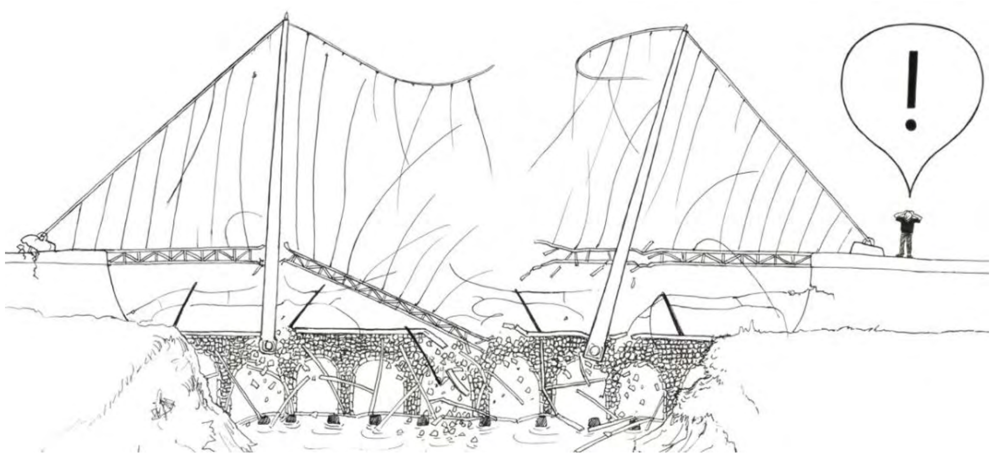


Figure 8. The result of tightening something just a little too much-



Because we evolved at every step just to meet the problem in front of us, because we never took the time to consider the future or even look deeply at the past, we built a system with complex and unpredictable emergent properties. At each step the decision seemed right and passed our tests (it met the change in requirements), but ultimately we ended up with something unstable and dangerous.

The reality is that we even probably held the information needed to predict what would happen, to understand the complex beast we had created, but for our last engineer this would have been too voluminous and too disorganized to actually be of use (Figure 9).

What we should have done, what we could in hindsight have done, was to take the time at each step to consider what had come before and what might come afterwards. No system is ever truly finished, we can't predict the future but through controlled processes, careful planning, and construction in a modular and flexible fashion to allow for less risky future evolution, we can do a lot to help our future selves and avoid the dangerous instabilities that can result.

Figure 9. Trying to make sense of it all, and failing.

3. Conclusion

The example shown above was created in 2015 from a fairly simple concept. The artwork assets and the general story narrative around them have since been reused in numerous presentations, some winning awards, to a variety of audiences including the general public (Cutting and Noppen, 2015). The eye-catching nature of the illustrations act as an attractor, help to step away from the dry nature of the subject at hand, and succeed in communicating the problems inherent in system evolution in an effective manner. Work is now underway to investigate ways in which other computer science concepts may be opened up using illustrative art.

Acknowledgements

The authors wish to acknowledge and thank the artists who created the visual assets Justin Harris and Amy Hunter. We would further thank the original project supervisor Dr. Joost Noppen for his assistance and support.

References

Cutting, D. & Noppen, J. (2015), "Déjà vu all over again: why software changes fail and how we can help", *Proceedings of the 2015 Postgraduate Research Day*, University of East Anglia.