Any attempt to explain software engineering to a lay audience soon falls back on analogy: building software is like building a bridge, a car, a television set. A large part of the established practice within software engineering is also based on this premise. However, the analogy is false in some important ways, and herein originate many of the problems that have bugged the software industry since its birth.

Software is never finished. Early process models of software engineering, based on conventional wisdom from other engineering disciplines, imagined it as a cascade of phases, in which there would not be any going back: requirements capturing, analysis, design, implementation, testing and maintenance. After all, once a bridge is built, you do not have to redesign it, right?

Wrong — for software. Though these phases exist and it is important to distinguish them, it has now been known for a long time that things are not that clear-cut: a software project does not move as a whole from one phase to the next. More importantly, the process is not that linear: for any serious piece of software, every single one of these phases is passed through again and again. Software is never finished, and we’d better accept that and rethink our methods accordingly.

Of course, software projects do finish (possibly less quickly than foreseen); but the software product stays. However, it does not stay like a bridge, majestically, unchanging and for all to use and admire: it is copied and re-used in contexts it was not developed for, it is extended with little useful bits and pieces of functionality, it is ported to new platforms, cut up and partially rewritten, until it ends up in that black zone we call legacy code — for the next project team to try to understand, curse at and be secretly afraid of.

Verification is never finished. The central belief in the formal methods community, voiced strongly and eloquently by its pioneers, is that programs should not be deployed, maybe not even written, unless and until they are provably and totally correct. Establishing correctness is usually called verification. We have since seen that this very hard for practical software, for a large variety of reasons ranging from fundamental to pragmatic; however, watered-down versions, involving approximative methods for weaker notions of correctness, have been reasonably successful; an example are the type systems referred to in the manifesto [10], but in specialised contexts also more sophisticated methods such as model checking [2].

Given that software is never finished, however, and that all phases of development are passed through again and again, it follows that verification is never finished either. Every extension, every modification, every partial reuse needs to
be re-verified in order to uphold the claim of correctness. Well-known is the case of the Ariane 5, which failed due to the reuse of a fragment of software that was correct in its original context. Especially in such a situation, where the original context is lost and the original design team is no longer around, re-verification can be even harder than the first time around.

Transformation as a way of life. In software engineering, change is the norm, not the exception. That being the case, we should also put change at the centre of our development methods. This is what is proposed under the header of model transformation \[9\]. To appreciate what this entails, one first has to broaden the mind: from software equating (textual) programs, to software consisting of all digital resources involved, including programs but also design documents, scripts, libraries, auxiliary inputs, configuration settings and whatnot: everything that is required to actually deploy the software. We use the term “model” to encompass each and every such artefact. Now, every change can be understood as a transformation of one set of models to another.

If we know the source model(s) of a given transformation to be correct — for instance because we made a dedicated verification effort — then ideally we should be able to conclude, without further ado, that the target model(s) are as well. For this to be true, it is enough to know that the change does not destroy correctness. If the change is ad hoc, say a textual edit to some input file, then this is in general impossible to know. Therefore, model transformation focusses on the concept of rules as the motor behind changes: every individual change is then an instance of a general rule. The preservation of correctness under change can now ideally be shown on the level of the transformation rules.

Challenges. Model transformation is a very active field of research. The ideal of correctness-preserving transformation \[7\] is one aspect of that research. Also growing is the insight that transformations often need to be invoked backwards, imposing a bidirectionality constraint \[11\]. To realise this, graph transformation is widely considered to be a good formal basis \[3,4,5\]: a lot of recent effort has been dedicated to lifting existing verification methods to that setting \[8,6\].

Model transformation as a means of mastering change offers a lot of promise. To fulfil that promise, further research is sorely needed. We will never go back to a world without software; so we’d better make sure it is long-lasting, forever software!

References

http://infoscience.epfl.ch/record/52559