Science of Computer Programming 97 (2015) 75–81

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Software engineering: Redundancy is key

Mark van den Brand, Jan Friso Groote *

Department of Mathematics and Computer Science, Eindhoven University of Technology, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

HIGHLIGHTS

- We provide a simple model to explain the use of redundancy in software construction.
- For reliable programs a redundancy level of appr. 4 to 8 is required.
- Testing, verification, model/implementation comparison typically add one redundancy level.
- Training of programmers or using higher-level languages hardly reduce the required redundancy level.

ARTICLE INFO

Article history:
Received 28 September 2013
Accepted 5 November 2013
Available online 20 November 2013

This article is written in celebration of Paul Klint's 65th birthday

Keywords:
Software engineering
Software quality
Redundancy

ABSTRACT

Software engineers are humans and so they make lots of mistakes. Typically 1 out of 10 to 100 tasks go wrong. The only way to avoid these mistakes is to introduce redundancy in the software engineering process. This article is a plea to consciously introduce several levels of redundancy for each programming task. Depending on the required level of correctness, expressed in a residual error probability (typically 10^{-3} to 10^{-10}), each programming task must be carried out redundantly 4 to 8 times. This number is hardly influenced by the size of a programming endeavour. Training software engineers do have some effect as non-trained software engineers require a double amount of redundant tasks to deliver software of a desired quality. More compact programming, for instance by using domain specific languages, only reduces the number of redundant tasks by a small constant.

Paul Klint and I met at the end of the 1980s when I was a PhD student in Nijmegen and we were both working, independently, on the generation of programming environments. My programming environment generator was based on Extended Affix Grammars and semantic directed parsing. During several visits to CWI/UvA I got more acquainted with the approach by the group of Paul which had a strong focus on incremental techniques using ASF+SDF to describe (programming) languages. In 1991 Paul offered me a job as assistant professor at the University of Amsterdam, where I started in 1992. This was the starting point of a long and fruitful cooperation. I worked on the design and implementation of the new ASF+SDF Meta-Environment [2]. Paul gave me the opportunity to explore my own ideas, and we had quite some discussions on the technical consequences. Paul is a tremendous scientific coach. He gave the opportunity to supervise my own PhD students and prepared me very well for my current job in Eindhoven. I am very grateful for his guidance over the last (almost) 25 years!

Mark van den Brand

* Corresponding author.
E-mail addresses: M.G.J.v.d.Brand@tue.nl (M. van den Brand), J.F.Groote@tue.nl (J.F. Groote).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.020

My motivation to come to CWI in Amsterdam in 1988 was my desire to learn more about making correct software. I felt that this could only be achieved by learning about the mathematics of programming, whatever that could be. To my surprise it appeared to me that in the department AP (Afdeling Programmatuur, later SEN, Software Engineering) only the group of Paul Klint really cared about programming, where all other groups were doing some sort of mathematics generally only remotely inspired by it. Observing Paul's group struggling primarily with the ASF+SDF project [7] and having very similar experiences later working on the mCRL and mCRL2 toolsets [5], have convinced me that neither committed and capable people, nor the current software engineering techniques, nor the mathematical apparatus of the formal methods community is enough to construct highly reliable software. Without forgetting what we know and have achieved a next step forward is required. To me it now appears that we have to proceed along the line of consciously employing redundancy.

Jan Friso Groote

1. Redundancy

Engineers construct artefacts far beyond their own human reach and comprehension. In particular these artefacts must have failure rates far lower than their own imagination. Typical failure rates for safety critical products are in the order of 10^{-10}. This means that a typical engineer will never see his own products fail. In most engineering disciplines, design and realisation are separate activities carried out by different people. This forces designers to make detailed, trustworthy and understandable designs for the constructors. The constructors validate the design before starting the actual construction.

The situation is entirely different for software engineers. They create products that are far more error prone than the physical products of their fellow engineers. Often the designer of the software also realises the software. Furthermore, it is common practice for software engineers to resolve observed problems in their own products on the fly. One may ask what the reason for this situation is, and thus find ways to improve the quality of software.

An important observation is that the production of software is a human activity and that humans make lots of errors. From risk engineering the typical rates of errors that humans make are known [13]. For a simple well trained task the expected error probability is 10^{-3}. Typical simple tasks are reading a simple word or setting a switch. Routine tasks where care is needed typically fail with a rate of at least 10^{-2}. Complex non-routine tasks go wrong at least one out of 10 times. An example of a non-routine task is to observe a particular wrong indicator when observing an entire system. Under the stress of emergency situations 9 out of 10 people tend to do this wrong. These failure rates contrast sharply with those found in common hardware, where a failure probability of 10^{-16} per component per operation is considered high [12].

What does this mean for the construction of systems by software engineers? Programming consists of a variety of tasks: determining the desired behaviour via requirements elicitation, establishing a software architecture, determining interfaces of various components, finding the required software libraries, retrieving the desired algorithms, coding the required behaviour, and finally testing the developed product. All these tasks are carried out by humans. It is not clear how to divide this variety of activities into tasks that are comparable to those used in risk engineering. But typically interpreting and denoting a requirement, writing down an aspect of an interface definition or writing a line of code could be viewed as a task. Each of these tasks has in principle a different failure rate. The risk of writing an erroneous requirement is higher than writing a wrong assignment statement, but for the sake of simplicity we ignore this. As we will see the precise nature of a task is also not really important. We assume that we understand that constructing a program can be divided into a number of *programming tasks* or *tasks* for short. At the end we go into this a little deeper and contemplate about typical tasks in the different phases of programming. It is obvious that a computer program is the result of literally hundreds of thousands of such tasks, and that the majority of these tasks are at least of the complexity of what risk engineers would identify as 'a routine task'.

We set the probability of a failing task to p, ignoring that different tasks have different failure rates. A value of $p = 0.01$ is quite a decent estimate. It would be good when concrete values for p for different types of programming tasks would be established, but to our knowledge no such results exist. We can easily derive that a typical program consisting of n programming tasks fails with probability $1 - (1 - p)^n$. A simple program built in 10 steps fails with probability 0.1, one with 100 steps fails with probability 0.6 and a program comprising 250 tasks already fails with a probability higher than 90%.

There are three major ways to reduce the number of errors in programs all widely adopted in programming. The first one is to reduce the number of tasks when programming. Moving from low level assemblers to higher-level languages is a typical way of reducing the number of tasks to accomplish a program with the desired functionality. Another example is the current tendency towards domain specific languages where code is generated based on a minimal description of a particular application in a certain domain, while all domain specific information is being generated automatically. There is no doubt that higher-level programming languages and domain specific languages are of great value. But even with them, programmed systems are so large that the number of tasks will always remain substantial. Given the high human failure rate, this means that higher-level languages on their own will never be able to provide the required quality. Building transformers for domain specific languages are substantial efforts also consisting of a huge number of programming tasks, and therefore also a fallible activity. Furthermore, when building a transformer for a domain specific language detailed knowledge of the underlying semantics of the domain specific language and the target general purpose language is needed as well as thorough knowledge of the application domain.