Software Engineering: Redundancy is Key

Mark van den Brand and Jan Friso Groote

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

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 does 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.

Keywords: Software engineering, Software quality, Redundancy

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

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

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.

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. What 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.

guage and the target general purpose language is needed as well as thorough knowledge of the application domain.

A second way of improving the quality of software is to train the software engineers better. In terms of risk engineering one could phrase this as experiencing programming not as a set of complex tasks, but as a set of routine task. We will see that this has a notable effect on the quality of software, but it is not sufficient not to use some form of redundancy. Even extremely experienced software engineers make so many mistakes that for improving the reliability of software each task must be carried out in a redundant manner.

The third approach to reduce the number of defects in software is the introduction of redundancy when developing software. In hardware it is common practice to have redundant components. Typically, by using n redundant components that have a failure rate of p, a failure rate of p^n can be achieved. In hardware it is common practice to use more computers running the same program in critical applications to reduce the risk of operational failure due to hardware error. If failure of redundant components is independent, this is an excellent way to achieve a high level of reliability.

Analysing of design artefacts counts as adding a form of redundancy. Deriving metrics from design documents and code has become common practice. These metrics are not only used to predict or justify the development costs but are also used to detect code smell and drive refactorings. Analysing high-level designs using model checking techniques is being introduced in safety critical systems. Also source code is being checked using model checkers [11, 8].

A huge contributor to the success of trendy agile development techniques is that it puts more emphasis on redundancy, in the form of reviewing of design artefacts, short development cycles, closer interaction with the client, writing tests before coding and pair programming [9].

We consider tasks redundant when if they are carried out and at least one of them is wrong, then this will somehow be detected. There are multiple ways of achieving this redundancy. The compilers can be very strict and reduce the number of trivial errors by parsing and type checking. This can explain the failure of COMAL, a programming language with the distinctive feature that it would repair syntax and semantical errors automatically as well as possible, of course leading to less reliable programs [4]. This also puts doubts on Python [10] as a programming language for applications where correctness matters as Python has very relaxed forms of type checking. Other ways are the use of asserts, array bound checking and other run time checks as forms of redundancy to expose runtime errors.

Reuse of components is another way of adding redundancy. Multiple usage exposes more problems, especially if this usage has different natures. The ATerm library [1] is a nice example. This library was initially developed for the new ASF+SDF Meta-Environment [2], but was also adopted, amongst others, by the mCRL2 toolset [5]. The different application area, generating huge state spaces and performing large scale model checking, posed different non-functional requirements and this revealed shortcomings of the ATerm library resulting in a more robust and stable implementation.

Testing, either systematically or ad hoc, is also a form of redundancy. Evaluating or formulating tests are tasks with a high failure rate. It is not uncommon that running tests leads to the detection that a test is flawed. Even the use of correctness proofs where invariants, pre- and post- conditions, modal formulas or behavioural equivalences can be checked are nothing more than activities adding redundancy to the process of software creation. The idea that correct software can be obtained by first specifying the intended behaviour, after which it needs to be implemented and the implementation needs to be proven correct has its limitations. For more complex systems the specifications by itself are already so complex, that they contain lots of mistakes loosing their authoritative value. Furthermore, correctness proofs are generally also not free from mistakes and oversights.

2. How redundant must a programming task be?

It is essential to use redundancy to obtain reliable large programs. Every programming task must be executed with sufficient redundancy, such that the likelihood of each task to be correct is so large that the whole program has a sufficiently low probability to fail. So, assume that there are n elementary program-

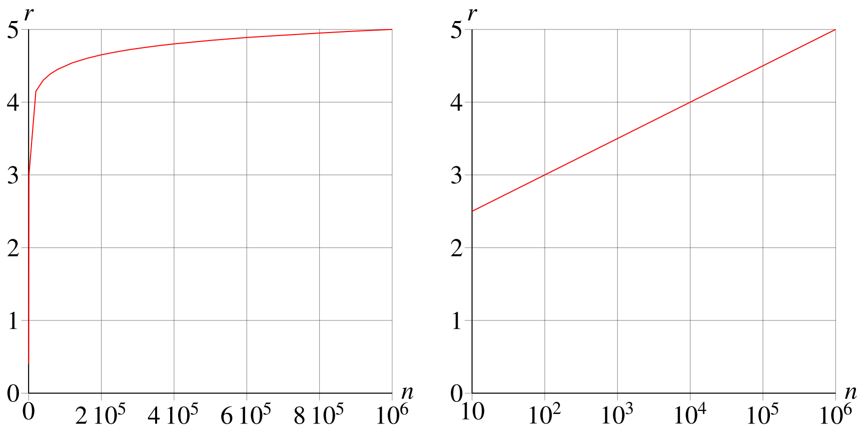

Figure 1: The required redundancy of tasks as a function from the number of tasks ($P = 10^{-4}$ and $p = 0.01$).

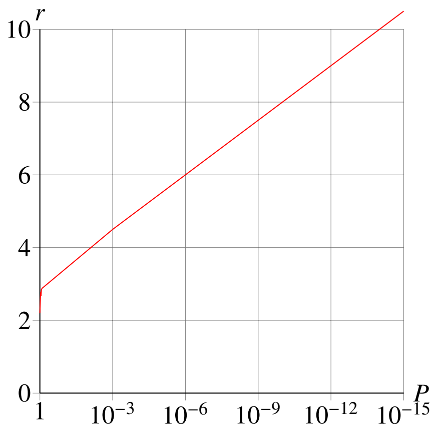

Figure 2: The required redundancy as a function from the required overall reliability ($n = 10^6$ and $p = 0.01$).

system is constructed out of hundreds of thousands of such tasks. Each task is carried out with a certain failure probability p. Also here, we do not know what the exact failure probability is; somewhere in the order of 0.01. We want our program to have a certain probability P to be flawed. Depending on the application, this value can vary from 10^{-1} to 10^{-10}. We will even see that this value does not really matter a lot. Suppose that each programming task is carried out r times, we can establish the following formula relating the probabilities:

$$\boxed{P \quad = \quad 1 - (1 - p^r)^n}$$

from which we derive that $r = \ln(1 - (1 - P)^{1/n}) / \ln(p)$. In Figure 1 this is graphically depicted in both graphs where the graph at the left has a linear domain and the graph at the right uses an exponential scale. It is important to note that the value of r hardly changes with the size of a system given a desired reliability of the system. It is however linearly dependent on the required error probability of the entire system, as the graph in Figure 2 shows. A top-safe system typically has a probability for residual errors of 10^{-10}, which requires that each programming task must be carried out with an eight-fold redundancy. The most

	10^{-4}	10^{-6}	10^{-8}	10^{-10}
10^{-1}	10.0	12.0	14.0	16.0
10^{-2}	5.0	6.0	7.0	8.0
10^{-3}	3.3	4.0	4.6	5.3

Table 1: Required levels of redundancy with different human failure rates

process. Section 5 presents a list of possibilities to increase the redundancy in software construction.

Training people can also help to reduce the number of faults in software. The human failure rate might shift from 10^{-1} to 10^{-2}, say, but given that programming is a difficult task, the effect of training can easily be overestimated (to our knowledge no research has been carried out in this regard, and hence no reliable figures exist). In Table 1 it is shown for human error rates of 10^{-1}, 10^{-2} and 10^{-3} what the effect on the number of redundant tasks is for varying levels of overall reliability of the software (values have been calculated for 10^6 programming tasks). It shows that training people extremely well can reduce the required number of redundant tasks with a factor 2.

The effect of the use of domain specific languages, or any other forms of more abstract programming languages can also be clarified. Suppose that encoding in a higher-level programming language leads to a 10 fold smaller input. Then we can easily see in Figure 1 at the right that the amount of redundancy is only reduced by a half, a small constant value. Given that the level of redundancy is not an extremely high number this is not totally neglectable, but its effect can easily be overestimated.

3. How to check redundancy?

If two redundant views on the same programming task are organised, one must make sure that the two programming tasks are independent and that if one or both of the programming tasks is carried out wrongly this is detected.

A typical example of wrong redundancy is running the same program twice on separate computers. Programming errors are not detected, while their risk of occurrence is many orders of magnitude higher than a hardware error occurring.

There are also some concerns with pair programming, where two programmers sit together working on one program. There is redundancy in entering program code, so this improves the quality of the code because the program code is being discussed and reviewed while entering. However, if the starting point is an erroneous specification, there is no guarantee the errors are detected in this way. The translation of the specification into unit tests may reveal errors and ambiguities in the specification. In this respect 'pair specifying' would be better assuming that the source of the specifiers is by definition correct, although interrogating such a source has its own challenges. The use of rapid prototyping and short development cycles with frequent interaction with prospective users contribute to the robustness of the software in this respect.

Classical testing where a program is run a number of times by testers trying to observe whether there is something obviously wrong, generally observed by a crash of the program, is also not effective in finding (hidden) errors. Many erroneous behaviours of the program are easily overlooked in this way. A large set of test cases that are simply run, but that do not contain built in checks that guarantee that their outcomes are correct is of limited value also. This implies that the tests have to be carefully designed in order to be effective. This is actually another form of introducing redundancy. The design of a test may reveal errors and/or ambiguities in the specification or requirements. The use of an extended set of unit and regression tests prevents the introduction of trivial errors when performing refactoring or doing maintenance. The application of test coverage tools helps in guaranteeing that all programmed tasks are tested, and this helps enormously. This is easily explained by the reliability formula on the previous page, as the overall reliability of a program is determined by the part with the smallest level of redundancy, even if this regards a small part of the whole program.

software can either be generated from the model, or it can be coded by hand. In the former case, the code generator has to be tested or proven correct to increase the redundancy. In the last case, redundancy can be achieved using model based testing against the executable model.

4. What is an elementary software engineering task?

It is not straightforward to employ redundancy to programming, because it is not obvious what a programming task is. To illustrate this we look at three different classes of tasks in program construction and elaborate on them. It is not at all intended to be an exhaustive classification of tasks, simply because we are not able to give that at this stage.

- Programming can most easily be divided into different tasks. The natural notion is to view writing a line of code as a task, but this might be an oversimplification. Given that a line of code can easily be split into more lines indicates that this notion of a task is not stable. More stable notions are the realisation of a function, an iteration or even a function point. When programming there is already built in redundancy, and therefore some parts of a program could be considered not to belong to the primary programming tasks, such as the declaration of variables or writing of assertions. The advantage of considering writing a line of code a task is that these tasks are easily identifiable and can easily be coupled to concrete human activity.
- It is less clear how to divide the design of algorithms and communication schemes into tasks. Writing down pseudo-code can be divided in a similar way as programming. If the correctness of the algorithm or protocol must be shown, writing each step of the correctness proof can be seen as an elementary task. Formulating a test scenario for each conceivable run through the algorithm could also fall into this category.
- Identification of user needs is much harder to split into elementary tasks, and we might even call in the help of psychology to address this. Describing each elementary concept of the user might be an elementary task, as is the identification of each relation between them. Formulating each operation that can be carried out on these concepts and identifying that the user understands each concept, relation and operation yields a set of elementary tasks. Verifying that each relevant notion complies with interface standards and best practices can be regarded as elementary tasks too.

5. Possible forms of redundancy

Since we are developing software we are using forms of redundancy, but given the fact that we are becoming more dependent on software, consciously dealing with redundancy is getting more important. In Section 2 we derived that programs should have between four and eight forms of redundancies. Here we will give a preliminary list of (possible) ways of increasing redundancy. This list contains mature and proven technologies as well as unexplored approaches. Some of the approaches are already discussed in previous sections.

Programming languages that facilitate statical analysis of the developed programs If a programming language facilitates the introduction of static types, typed variables, etc. the more checking can be performed by the compiler and the more unintended behaviour can be detected. If the language allows the explicit formulation of statically checkable invariants then even more unexpected behaviour can be detected and prevented.

Reviewing Any form of reviewing uncovers errors and increases the reliability of the software. Reviewing can be done at the level of requirements, design, coding and testing. Even reviewing the outcomes of tests or the verifications of requirements is useful, because these might be incorrect, although our model in Section 2 assumed that such comparisons between redundant activities are impeccable.

Testing Although testing only reveals errors but does not show the absence of errors, it is a common practice to increase the quality of software. There are various ways of using and developing tests. Tests can be written based on the requirements. Tests can be written before the actual coding starts, again a popular way of working in agile software development. Tests can be written after an error has been detected but before it is corrected. Tests can automatically be generated from specifications, so-called model based testing. The redundancy increases if testing is supported by tooling to measure the test coverage.

Formalisation of requirements Requirements elicitation is a challenge in many software projects. Often natural language is used to describe the requirements. Proper reviewing of requirements already reveals many shortcomings, however formalisation, either mathematically or as an executable specification, improves the overall quality of the requirements, because in case of the former of rigorousness and in case of the latter of the immediate feedback to the users.

Re-use The fact that software components/libraries are used in different applications makes the components/libraries more robust. Each application may ask for different (non-)functional requirements.

Specification of interfaces The precise specification of the application programming interface (API) of a software component is an important requirement in order to make this component re-usable. The robustness of the component improves if the component checks whether the calling software and the component itself adhere to the desired contract. This can be done via runtime checks (expensive) or via static checks based on theorem proving, equivalence checking or the verification of modal formulas.

Adherence to architectural rules Software architecting is rapidly maturing. Where in the past software architecting was about components and their relationships, it has moved towards the relationships to its environment and the rules to enable the evolution of the software. The formulation of architectural rules and adherence to these rules when developing the software, improves the overall quality of the software.

Redundant software components This form of redundancy is closest to hardware redundancy. Redundancy is obtained trough the independent development of components with the same functional behaviour. In its most extreme form two independent groups develop components that can be executed in parallel. These components need not be programmed in the same language. A variant is the development of a (executable) model, that can be used for prototyping, testing or code generation. If the model is machine processable it can be used for simulation and/or model checking.

6. Final words

We are still struggling to find a workable software engineering approach to construct programs on which we can rely. We want to achieve the situation where if we state that software is ready, then we can be reasonably sure that no residual errors remain.

In the past many techniques have been proposed that do have their merits to increase the quality of software, but completely in accordance with expectation none counts as the silver bullet [3]. We argue that combining the different techniques is a viable way of obtaining the desired quality. Each programming task only needs to be carried out a limited number of times, independently of the size of a programming endeavour and hardly influenced by the quality of the program developers.

It is required to identify different programming tasks more explicitly than is done up till now, and to identify four to eight redundant activities for each programming task. In the exposition above examples have been given, but it is still open to come up with a workable scheme, and it might be that new ways of adding redundancy to programming must be invented.

It would be very useful to refine the rather simplistic reliability model that we provide and to quantita-
[illegible]

not in the least as it requires field trials on a substantial scale (cf., e.g., [6]). The benefit of such activity is of course huge. We will have the tools to isolate and identify the most effective software engineering practices, allowing us to abandon the less effective techniques.

[1] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. Software: Practice and Experience, 30(3): 259-291, 2000.

[2] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L.M.F. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J.Visser. The ASF+SDF meta-environment : a component-based language development environment. Proceedings CC 2001, editor R. Wilhelm, Lecture Notes in Computer Science 2027, pp. 365-370, Springer, 2001.

[3] F.P. Brooks. The mythical man month. Essays on software engineering. Addison-Wesley. 1995.

[4] B.R. Christensen. The Programming Language COMAL - Denmark. International World of Computer Education, 1(8):26-29, 1975.

[5] S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, J.W. Wesselink, and T.A.C. Willemse. Ins and outs of the mCRL2 toolset. Proceedings TACAS 2013, editors N. Piterman, S.A. Smolka, Lecture Notes in Computer Science 7795, pp. 199-213, Springer, 2013.

[6] C. Jones. Software assessments, benchmarks, and best practices. Addison-Wesley, 2000.

[7] P. Klint. A meta-environment for generating programming environments. ACM Transactions on Software Engineering and Methodology 2:176-201, 1993.

[8] D. Kroening, E. Clarke and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. Proceedings of DAC 2003, pp. 368–371, ACM Press, 2003.

[9] S. McConnell. Code Complete. Microsoft Press, 1993.

[10] G. van Rossum and F.L. Drake Jr. An introduction to Python. Network Theory Limited. 2011.

[11] B. Schlich, S. Kowalenski. Model checking C source code for embedded systems. Inernational journal on software tools and technology transfer, 11:187-202, 2009.

[12] B. Schroeder, E. Pinheiro, W.-D. Weber. DRAM Errors in the wild: a large-scale field study. In proceedings of the eleventh international joint conference on measurement and modeling of computer systems (SIGMETRICS'09), pp. 193-204, ACM, 2009.

[13] D.J. Smith. Reliability, maintainability and risk. Practical Methods for Engineers. Elsevier. 2011.