

# Increasing Software Reliability through Use of Genericity

Thomas Schöbel-Theuer, Universität Stuttgart  
schoebel@informatik.uni-stuttgart.de

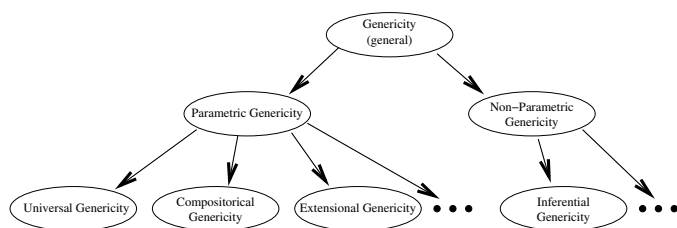
**Abstract**— In our opinion, methods for *construction* of reliable software are of great importance. Reliability engineering must start in the earliest phases of a software project, and it has to consider not only the architectural level, but should be in the mind of humans even earlier when analyzing the problem space.

In a previous paper, we argued that *reduction of redundancy* of software is a central factor for increasing economy and for increasing the scope of what can be handled by human groups. In this note, we argue that reduction of redundancy is also a central factor for increasing reliability. We review our subclasses of genericity, called universal genericity, compositorial genericity, and extensional genericity, with respect to their impact on reliability. We find that the superiority of universal genericity and compositorial genericity over extensional genericity with respect to their *potential* for decreasing redundancy will take over to the field of reliability, meaning that universal genericity has the highest potential for building more reliable systems.

As a consequence, OO analysis and design methods (which mostly prefer extensional genericity) are *not* the highest potentials for making complex systems extremely reliable.

## I. INTRODUCTION

In a previous paper [1], we proposed to assign a more general meaning than usual (e.g. the keyword `generic` in Ada) to the term "genericity": Any automatic tool or systematic method, which *allows* for reduction of human-visible redundancy in some software structure without decreasing functionality, is called a generic tool or a generic method. We introduced several subclasses of genericity (not necessarily excluding each other) as depicted:



*Parametric genericity* is a rather broad subclass of general genericity subsuming anything which could be done with formalized substitution mechanisms. Examples are macro processors,  $\lambda$ -calculus and corresponding evaluator machines, and many more. Note that parametric polymorphism of types [2] is regarded as a true subclass of parametric genericity restricted to type concepts, mostly used for achieving *genericity in the small*, while we also address *genericity in the large* (e.g. on architectural level).

*Universal genericity* is the ability of an interface or an implementation to *simulate* other interfaces or implementations in a rather *simple* way. Examples are universal turing

machines, parser generators, generic file I/O in Unix (as opposed to record-structured I/O), many kinds of interpreters, and many more.

*Compositorial genericity* is the ability for performing compositions on instances of functional units. Examples are combinatorial style in functional programming, Unix shells or Unix make controlling "filter" processes interconnected with pipes or intermediate files, architectural styles like "pipe and filters style" [3], and many more.

*Extensional genericity* is characterized as inclusion by reference followed by extension, e.g. include files, embedded www frames, object oriented extension of interfaces and implementations by use of inheritance, and many more. Inclusion polymorphism (also called subtype polymorphism) [2] is regarded as a true subclass of extensional genericity restricted to transformations on types.

When used as styles of thinking, these classes of genericity can be used by humans (similar to tools) in very early stages of a software project, even on the problem space before entering the solution space (e.g. by decomposing problems according to one of the building principles) and even before writing a specification.

## II. IMPACT ON RELIABILITY

Our original motivation for reduction of redundancy was driven by economical arguments and tractability arguments. We found that universal genericity has the highest *potential* for reduction of redundancy, because it may be *potentially* applied in an a-priori *unbounded* number of places, even *dynamically* and in unforeseen places. Compositorial genericity bears a *potential* for *exponential* growth of possible combinations of functional units to networks (cartesian product) and thus bears a *potential* for exponential reduction of redundancy. In contrast, extensional genericity bears only a *potential* for *linear* reduction of redundancy, because at each application of inclusion by reference a "logical copy" is created [1]. When multiple kinds of genericity apply to (parts of) a system in parallel, our argumentation applies to each of those applications separately.

We view reliability as the outcome of a human effort to produce systems with a certain degree of reliability. Such an effort is often long-lasting and leads to cyclic revisions and improvements similar to a control loop. Highly reliable systems are seldomly produced at once in one single shot. When human resources are limited, the final outcome is likely to be reached *earlier* (or sometimes even at all) if the intended

functionality can be achieved with a less redundant software structure. Code checking, bug fixing, testing, all become easier and their amount is reduced ("simplicity"). Thus the likelihood of reaching a certain degree of reliability within a fixed time will increase with the decrease of redundancy in the software structure solving one single problem instance<sup>1</sup>. Thus we conclude that genericity as defined by us is also a tool for increasing the likelihood of reaching a high degree of reliability.

Now we turn to the question of the impact of each of our subclasses of genericity on the *potential* for accelerating human processes for production of reliable systems.

For good reliability of a universal interface or implementation, it must be correct for the whole class of problems for which it is intended to be universal. The main effort will go into making the universal machine as reliable as possible. Good universal concepts require only "trivial" adaptations to new application environments, so the effort for getting that adaptation reliable enough should be relatively small (but nevertheless may be very important for the overall reliability of the whole system). In summary, universal genericity creates an *opportunity* to solve some reliability issue on a large problem class approximately *once and forever*.

Compositional genericity has to deal with reliability of each component type as well as of all potential compositions of instances to networks. While the first usually introduces no additional problems, the latter may become expensive due to potentially exponentially many combinations. By introduction of the concept of *classes* of networks and by proving reliability on class level, this complexity should be reducible. As pointed out in [1], good compositional designs use a high degree of universal genericity on the interfaces between functional units. Thus in many cases it should be sufficient to prove reliability of components against interfaces, not particular network configurations. As a consequence, interdependencies between instances become *orthogonal* ("blackbox composition"). In ideal case, the reliability of the full class of all composable networks should become treatable this way; some parameters like number of instances, input/output degrees, nesting levels etc. may be used in coarse models of total reliability of classes of networks. However, in general some specific problems like *error propagation* require attention in compositional networks.

Extensional genericity focuses on reuse of existing components by extending them. In many cases (e.g. OO inheritance, friend classes), *whitebox extension* is used or can at least potentially be used and must therefore be checked. In case of whitebox extension, the reliability of the extended system need not linearly depend on the reliability of the reused component, but rather may require a separate in-depth examination of the more complex total system. At a higher level, many OO systems create large runtime heaps of object instances

interconnected with pointers; sometimes *composite objects* are built via dynamic pointers. Reasoning on such structures may become extremely hard. Thus we conclude that the probability for getting large extensional systems sufficiently reliable is in many cases lower than with compositional or universal genericity.

### III. CONCLUSIONS

The relation universal > compositional > extensional genericity does not only hold for the *potential* for reduction of redundancy, but also for the potential for increasing reliability. As always when making statements on *potentials* of general classes of SE methods, concrete special instances may behave differently.

Although extensional genericity bears the lowest potential, most research effort of the past has gone into it, e.g. OOAD. To exploit the larger potential of compositional and universal genericity, much more work is needed on them, independent from OO and no longer regarding OO as an inevitable basis for new research directions.

### REFERENCES

- [1] T. Schöbel-Theuer, "On variants of genericity," in *Proceedings of the Fifteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2003)*. Knowledge Systems Institute, 2003, pp. 359–365.
- [2] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *Computing Surveys*, vol. 17, no. 4, pp. 471–522, 1985.
- [3] M. Shaw and D. Garlan, *Software Architecture, Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [4] M. R. Lyu, *Software Fault Tolerance*. Wiley, 1995.

<sup>1</sup>There is no contradiction to methods introducing redundancy by replication of problem instances, solving each independently, and finally selecting the "best" version for increased reliability [4]. Our arguments apply to problem solving on a replica. However, under limited manpower constraints or when the ratio price/reliability matters, it may be sometimes beneficial to spend the human effort on an extremely reliable solution for an unreplicated problem instance instead of forking less reliable copies.