On Variants of Genericity

Thomas Schöbel-Theuer

Universität Stuttgart

E-mail: schoebel@informatik.uni-stuttgart.de

Abstract

We discuss some general styles of human thinking which could be useful at the design stage of a software project.

The central idea of genericity (as viewed by us) is to reduce redundancy *in a software structure. This leads not only to better software structures, but also improves economy (by consuming less human resources). Reduction of redundancy should start in early phases of the development process, in particular in the design stage. Our concepts have proven valuable for a novel operating system design.*

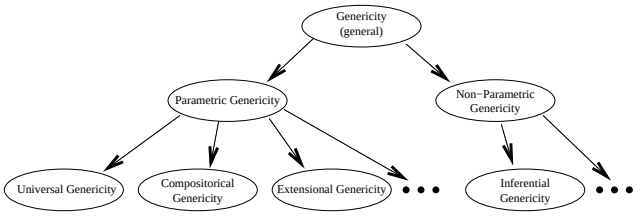
We draw some important conclusions: universal genericity and compositorical genericity are regarded as superior to extensional genericity in many cases. Since OO inheritance is a subcase of the latter, we regard it as an optional subordinated concept. We show that the first two concepts can simulate functional programming (FP) and aspect-oriented programming (AOP), among others.

1 Introduction

The term ”genericity” has already a more or less accepted informal meaning. In [Mey88], Meyer uses the term approximately as a synonym for ”parametric polymorphism” (of types) [CW85]. The common use of ”genericity” shows up e.g. in the keyword `generic` in Ada. We argue that the term ”genericity” should be assigned a more general meaning, such that parametric polymorphism of types becomes a special subvariant of genericity. Additionally, the *scale* is extended: while parametric polymorphism is mostly used for programming in the small (e.g. generic lists or other abstract data types), our notion of genericity fits better for programming in the large.

We identify some subtypes of genericity, called universal genericity, compositorical genericity, and extensional genericity, and we give some examples for them, most from our research on operating system architectures. We argue that the term ”genericity” should not be used standalone, but rather be combined with a qualifying attribute such as ”universal”, ”compositorical”, ”extensional” or ”parametric” (or probably a combination of such attributes for further fine-grained distinctions). This way, we are open for further variants of genericity which may be discovered in the future.

An overview on variants of genericity:



In short, the basic idea of this paper is to reduce redundancy not only in the software structure of a complex system, but to start with redundancy reduction as early as possible, in particular in the design stage.

A great deal of motivation for redundancy reduction is supplied by economical arguments: if the same functionality can be implemented with less redundancy, less manpower is required for the project in total.

The paper is organized as follows: section 2 explains general genericity and motivates it. Section 3 treats subvariants of parametric genericity which are useful as styles of thinking in early design stages. Section 4 shows examples from an operating system design based nearly exclusively on universal and compositorical genericity. Afterwards, simulation of functional programming (FP) and of aspect oriented programming (AOP) by our concepts is sketched. Section 5 concludes the paper.

2 General 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 and subsumed under the general notion of genericity.

Note that reduction of redundancy need not take place in a system as a whole, but may be restricted to parts, subsystems or to particular *views* on a system. The term redundancy is not meant in a formal information theoretic sense,

but informally referring to ”amount of code” or ”effort for the desired functionality”. We are talking about *concepts* in early phases of a software project, graspable by humans, thus formal definitions would be hard to apply.

Our informal definition of genericity is extremely broad, deliberately. If we are not careful, probably almost anything could be subsumed under it[1]. Because of that, we strongly advise to avoid genericity as a standalone term, but rather use it in combination with *qualifying* attributes; e.g. ”universal” as an attribute to genericity is not a property of ”genericity”, but a restricted type of genericity.

Reduction of redundancy may be seen as a very general guiding principle on many things done in software engineering. Humans critically depend on reduction of redundancy, because of ”our inability to do much” [DDH72, Dijkstra's section 2]. Without reduction of redundancy, we have almost no chance against the complexity of extremely large systems. We can improve the ratio between human ability and system complexity by adding manpower to a software project and by parallelizing work on different regions of that system, but practice has shown the limits of such an approach. Another way is to reduce redundancy of the project, such that it becomes smaller and can be handled with less manpower. There is clearly an economic argument for reduction of redundancy. But there is also an argument from tractability: less redundancy increases the scope of what can be handled by human groups.

In order to approach our goal of reducing redundancy in a complex software project, we have to start in rather early stages of the project, in particular we have to reduce redundancy during the design stages. Note that there are often many different ways for reduction of redundancy. This is what the rest of this paper is about.

3 Variants of parametric genericity

Parametric genericity is a rather broad subclass of genericity. It subsumes anything which could be done with formalized substitution mechanisms. Since the early days of computer science, hundreds of substitution mechanisms have been invented and implemented. Macro processors have been used very early. Lisp is one of the oldest programming languages based on substitution as a guiding principle, dating from the 1950s. Parametric genericity may be characterized as any tool or formal method, which *substitutes* placeholders (of any kind) by some meaningful entities such as terms, expressions, graphs, strings or whatever. It is independent from substitution time, whether static (at system generation time) or dynamic (at runtime). Formal mathematical substitution models such as the λ-calculus [Fie88] have been used even before the invention of the automatic computer. Today, a rich variety of typed and untyped substitution models has been examined [SPvE93]. Note that parametric polymorphism of types [CW85] is just one special incarnation of parametric genericity, and by far not the only useful one for software architects. Inclusion polymorphism (also called subtype polymorphism) is just another example, which may be obtained by combining compile-time substitution with late binding at runtime (where late binding may be characterized as specialized evaluation order on runtime substitutions). Even the rather powerful type system of Haskell [Tho96] should be regarded as a specialized subclass of parametric genericity. Although a macro processor such as the C preprocessor or Gnu `m4` is a rather simple tool, it may not only simulate most transformations and substitutions on data types[2] (possibly with slightly different concrete syntax), but may execute rather sophisticated transformations on the program text itself which go beyond the capabilities of `generic`s in Ada (of course, they provide less type safety and often induce other problems we don't want to discuss here).

While theorists ask for the formal power of various models, software engineers ask for practical application scenarios and for improvements that could arise for solutions to their problems, and for recipes how to apply a method fruitfully. In order to describe practical application areas and scenarios, we look at some more specialized application styles of parametric genericity we have identified when working on a novel operating system design [ST02, ST03]. Our subtypes of parametric genericity should be regarded as *styles of thinking* for humans at a *concept finding stage* in a software project, which goes beyond architectural styles [SG96]. By the term ”style of thinking”, we mean patterns of reasoning on an *emerging* (not fully constructed or understood) system, recognizable and graspable by humans.

The application area for our styles of thinking will also cover earlier stages like requirements engineering (or domain engineering). Some variants of genericity, in particular universal and compositorical genericity, may lead to a *paradox*: employing them may not only decrease redundancy, but also increase functionality and thus over-satisfy original requirements. In such a case, it is often fruitful to reconsider requirements in an evolutionary mode not only for omissions and slight modifications (e.g. modification of the exact way how a recognized problem should be solved), but also for improvements of the ratio price / functionality. Increased functionality may be cheaper to realize than sticking with a particular problem solution which had been inadvertently fixed as a requirement, but could be changed with low costs.

[1] Future improvements should try the reduce that risk.

[2] Even simulation of automatic *type inference* (cf. Haskell or ML) is principally possible with macro processors, at least if they are Turing-complete, such as is known from the TEX macro processor.

3.1 Universal genericity

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

The term ”simulate” is used informally in the broad sense of ”enabling the same functionality as ..., but probably in a different way”. By the informal term ”simple”, we mean that additional coding effort for interpreting the simulation results or for converting simulation input must be substantially smaller than the coding effort spent in the simulating device itself. Conversions can be usually carried out by rather simple substitutions, thus universal genericity is regarded as a subclass of parametric genericity.

Universal constructs are found in computer science in many places: a universal turing machine is one that can simulate any other turing machine. An *interpreter* for a parser may deal with any deterministic context-free language, by providing a parser table as input which describes a push-down automaton. Universality is always relative to the universum of concepts which can be simulated; in the last example this is only the class of deterministic context-free languages, but not the full class (for example, most ambiguous languages cannot be simulated).

A good example for a practical application of universal genericity is the file IO interface of Unix [RT74]: the `read()` and `write()` system calls are equipped with a `length` parameter. Any calls may be individually supplied with different actual parameter values, so the same file may be read by different consumers in differently sized patterns. Before the Unix concept of a file as unstructured sequence of bytes was born, operating system interfaces usually have dealt with different types of files, such as record-structured files. For simulation of record-structured files, we just have to supply the record length constantly as actual value for the length parameter at each call to `read()`. This causes a slight increase in runtime overhead, but the benefits of this type of universality actually do more than outweigh this overhead and have been widely accepted all over the world: not only the internal implementation in the Unix kernel is easier and less redundant as if it had to deal with multiple file types, but even the application programs benefit from a smaller number of system calls and from increased flexibility.

This example of universal genericity shows that ”less” can in effect produce ”more”: by *elimination* of the concept of a ”record”, the system is both simplified and becomes more flexible. Thus, universal genericity is *highly desirable*[3]. Whenever a chance for applying universal genericity is discovered, this should be preferred over other kinds of genericity as treated in the next subsections. Currently, we know no general formal method for achieving universal genericity. However, we encourage system designers to systematically look for opportunities for universal genericity. Good candidates are concepts which are ”similar” to each other.

Note that just taking the union of multiple interfaces to form a new single interface is nearly the opposite of the spirit of universal genericity: doing so will not reduce interface complexity in general. With true universal genericity, all or nearly all methods provided by the interface should contribute to any of the possible simulations. Good designs will provide *orthogonality* on the methods: no method could be simulated by combination of others.

Universal genericity may be achieved by employing both abstraction and generalization in the sense of Návrat [Náv96], but not all abstractions or all generalizations or all combinations thereof will necessarily lead to universal genericity (e.g. lead to extensional genericity instead). Some types of universal interpreters may employ generalizations in which the interface is not extended, but rather shaped very differently.

A very loose and incomplete characterization of universal genericity in OO terms could sound like: design a superclass, for which never a subclass will have to be implemented, because the (nevertheless rather simple) implementation of the superclass already provides anything you will need (for the desired application range).

3.2 Compositorical genericity

Compositorical genericity is the ability for performing compositions on instances of functional units.

Known examples are mathematical composition of functions, programming with combinatory logic or combinatorical style in functional programming [SPvE93] (which also shows that composition can be expressed by substitution), Unix shells controlling ”filter” processes interconnected with pipes, or Unix `make` which controls parallel invocation of compilers and linkers for building an executable by composition from many source files. Further examples illuminating the advantages of compositorical genericity will be shown in section 4.

New functionality is created by composition of instances of functional units to a network. It draws heavily from the cartesian product which potentially leads to exponential explosion of possible combinations. The philosophy can be characterized as nearly the opposite of extension of interfaces: in order to remain combinable with each other, interfaces should remain *compatible* with each other. New functionality is either created *inside* instances of functional units without alteration of interfaces, or by composition to new network configurations.

[3] Note that higher-level abstractions are not precluded by universal genericity – we just argue that they should be based on a *small* number of *universal* low-level abstractions.

Compositorical genericity should be applied after applying universal genericity to interfaces, in order to reduce the variety of interface types. In ideal case, only one or a low number of universal interface types should be employed. This way, maximum combinability will be ensured.

3.3 Extensional genericity

Extensional genericity is probably the most wide-spread subtype of parametric genericity. Simple examples are use of include files or other forms of *inclusion by reference* such as www links to other contents inside of web frames. Object oriented extension of interfaces (and often of implementations) by use of *inheritance* are more sophisticated typical examples. Note that inclusion polymorphism (also called subtype polymorphism) [CW85] is a subclass of extensional genericity restricted to transformations on types, while parametric polymorphism is an analogous subclass of the more general parametric genericity. Thus, parametric polymorphism should be regarded as the more general concept (note that use of inclusion polymorphism has often been motivated by its ability to runtime dispatch, but parametric polymorphism can also be implemented by runtime substitution).

The philosophy of extensional genericity is to add new functionality by extension of existing interfaces and implementations. In practice, this leads often to complex relations between classes and to a ”balconize-whenever-seems-necessary” mentality (in particular when requirements are changing), such that redundancy is eventually increased (e.g. incompatible extensions). However, in principle extensional genericity is able to reduce redundancy because the parts which are included by reference have to be denoted only once. However, this property is not characteristic of extensional genericity as such, but rather is characteristic of *any* form of genericity (by definition; this is a justification for the broadness of our definition). In our opinion, these interrelations of concepts are often either confused or not considered by many people, because the mechanisms of extensional genericity are often easier to understand than those of other forms of genericity.

Another motivation for extensional genericity has been *reuse*. However, inclusion by reference means just to reuse the referenced item in order to create a *new* one. This is similar to creating a copy (we could call it *logical copy*) and extending it. When iterated, this easily leads to code bloat on sourcecode level, just the opposite of our goal. It is important to recognize that reuse may also be achieved with universal and compositorical genericity. While reuse by inclusion will always produce some new item, universal and compositorical generity may just bear a *potential* of forming new items, but it need not be carried out at once (*late composition* or *late application*). Additionally, creating a new functional item for compositorical genericity will *multiply* the potential of forming combined items instead of actually creating exactly one logical copy.

In our opinion, the mentioned problems result from *unreflected* use of extensional genericity when other forms of genericity would be more appropriate.

As a result from this discussion and by support of the next section, we suggest to prefer both universal genericity and compositorical genericity over extensional genericity whenever possible. We will see that universal and compositorical genericity are sufficient for doing anything that could be done with a computer. As a consequence, OO inheritance is not regarded as vital, but rather as an optional concept.

4 Examples from operating systems

We explain our subtypes of genericity by examples from the operating systems area, where our original motivation comes from. We have developed a novel architecture for full-scale operating systems [ST02, ST03]. According to figure 2, it may be characterized as a ”pipes and filters style” in the sense of software architecture [SG96].

The wires in figure 2 may be viewed as ”transportation channels” which *logically* transport instances of an abstraction we call ”nest”. The nest abstraction is a *universal* address space abstraction with an additional `move`-operation to allow for reorganizations in the virtual address space (space management; details may be found in [ST03]). Nests are used to simulate not only files, but also whole filesystem (sub)trees and process images, among others (such as pipes and sockets). Thus, nests are striking examples of universal genericity.

The boxes in figure 2 are called ”bricks” and may be characterized as ”transformers” between nest instances. They are depicted with inputs on their left and outputs on their right, similar to functional units in electrical engineering [Hot74] or automation control [IEC]. Wires are directionally drawn from left to right. By dynamic wiring, we implement anonymous connection-oriented directional communication (as opposed to OO, which employs connectionless undirected communication, often on known partners). Note that communication normally goes from right to left, in the opposite direction of logical transportation. Wired networks of brick instances are striking incarnations of the principle of compositorical genericity.

Bricks may be implemented *stateless*. A stateless brick instance may be de-instantiated at any time provided that currently no activity is going on inside it, and later be re-instantiated (with the same wire connections), such that there is no observable difference in behaviour from the outside. As an example, the `buffer` brick instance in figure 2 roughly resembles a ”buffer cache” in conventional op-

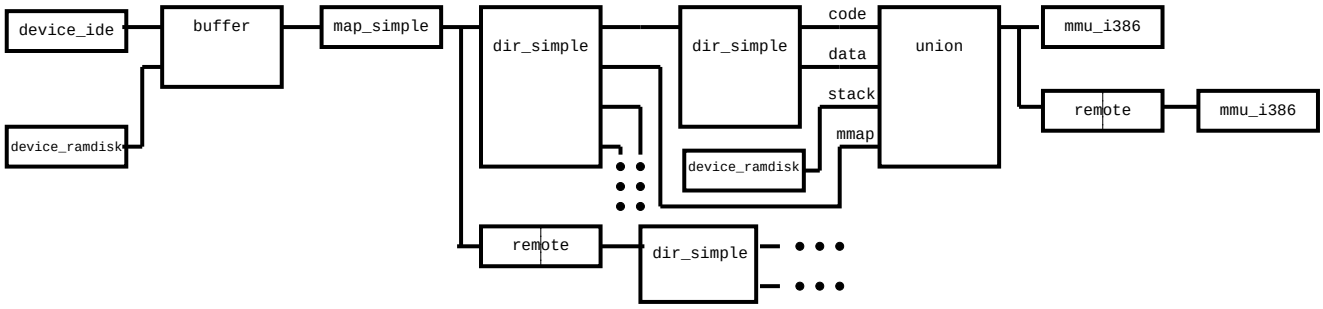


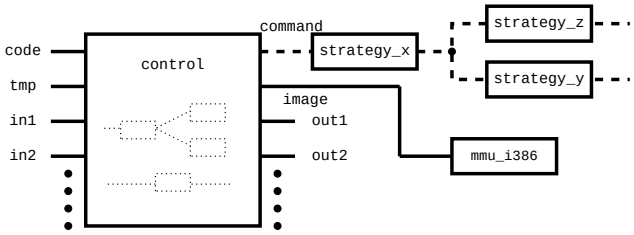
Figure 2. stripped-down OS scenario

erating systems by use of internal data structures such as hash tables for keeping a transient mapping from logical addresses to physical addresses (of the cached data blocks). A stateless `buffer` implementation will keep its internal state in its lower input which is usually connected to `device_ramdisk`, as well as the data blocks which are maintained by the cache and passed by reference. If *all* state information is always kept in an input instead of inside the instance, the instance may be de-instantiated at any time when there is no activity, and re-instantiated without a noticeable effect on runtime behaviour. If statelessness is applied to any brick type, we get a network of instances which *delegate* the responsibility for keeping state transitively to their predecessors until some `device_*` is reached, which itself delegates it to the hardware (for performance reasons, so-called *pseudo-stateless* buffers may be inserted to keep some state for some limited time). Statelessness allows for enormous simplification of reconfiguration, such as process migration on a network of computers.

The property of statelessness is in fundamental contrast to OO style of thinking, where both state and behaviour are regarded as essential for objects. Bricks are rather similar to *components*, which are also stateless according to Szyperskis definition in [Szy98], but components cannot be instantiated (we may even instantiate bricks *recursively*).

Our ”pipe and filter style” differs from known implementations of that style in the operating system area, such as UIO [Che87] or stackable filesystems [HP94], in a number of ways. First, it has no ”consuming” semantics like pipes, where processing of data would ”destroy” old data and produce a new version instead. Rather, a brick adds to the ways we look at the system, by generating a new *view* on the data: both the old view and the new view provided by the transformation will exist in parallel; the ”old” view may for example be used by parallel wiring to other ”consumers” or ”clients”. Second, it uses a *universal* (but nevertheless rather simple) abstraction called nest on all layers of the whole operating system, not limited to filesystems. We take advantage of the fact that bricks may be instantiated *dynamically* at runtime, thus resembling the dynamic nature of filesystem subtrees directly by recursive instantiation of `dir_*`-bricks. We use a dynamic version of compositorical genericity and draw from its ability to create new functionality by dynamic creation of new network configurations.

Now we look at an architectural feature which makes it unique, even for an application software engineer: the way how instances are generated.



The idea is to create and maintain brick instances of any type by a special brick called `control`. However, a `control` instance does not create other brick instances directly, it rather creates an `image` which may be thought of a ”process image” which ”contains” the instances and their wiring. In order to really ”execute” the instance network, some `mmu` instance like `mmu_i386` has to follow after (possibly indirectly), similar to the method as ”normal” ”processes” are brought to execution in figure 2. This allows for opportunities to apply additional transformations on the `image`, such as composition with other images.

Moreover, there are further opportunities for compositorical genericity: the *commands* for instantiation and de-instantiation are issued on ”control lines” depicted as dashed wires. Over these control wires, one can get information on the instantiated network (such as the wiring graph structure) as well. Thus a dashed control wire provides a *view* on the system structure. There may exist multiple views *in parallel*. For example, `strategy_*`-instances may perform simple checks like disallowing cyclic wiring to enforce hierarchical layering, or may *transform* between views, or even create *virtual views* which do not exist in ”reality”.

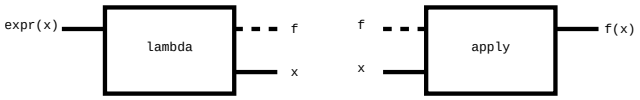
What are the benefits of compositorical genericity on the `strategy_*` level?

As an example, `strategy_transparency` may provide network transparency, by automatically inserting `remote`-instances into the system wherever it is necessary to span several hosts on a computer network, and by providing a single virtual system image. The automatically inserted `remote`-instances may be hidden for users of `strategy_transparency`, so that they get the virtual impression that no network would exist at all, and as if everything would execute on a single "virtual computer".

As another example, we may create hardware platform transparency by inserting hardware-specific brick versions at the right places. Multiple views may not only exist in parallel, but may be *combined* with each other using compositorical genericity. When hardware transparency is combined with network transparency, it should not make any difference whether one buys a SPARC or an Intel computer and connects it to the network.

Further examples for `strategy_*` are automatic load balancing, fault tolerance, automatic insertion of `adaptor_*` for transparent mixing of different interface versions, or centralized enforcement of security policies using `check_*` instances.

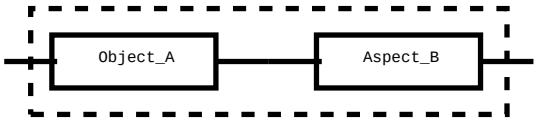
Now we demonstrate the theoretical power of universal and compositorical genericity: we sketch the simulation of basic concepts of functional programming (FP) [Fie88]:



The idea is to create a brick `apply` taking a function `f` and an argument `x`, in order to logically produce a result `f(x)`. With `lambda` bricks, we can simulate λ-expressions by connecting its output `x` to free `x`-wire inputs (corresponding to occurrences of a free variable `x`) of another subnetwork `expr(x)`, whose output is in turn connected to the input of `lambda` (leading to a cycle). By transforming complex λ-expressions of the *pure* λ-calculus [Fie88] into equivalent networks of `lambda` and `apply` instances and by use of an appropriate `strategy_lambdarewriting` which simulates graph reduction, we may simulate the very heart of functional programming. Note that this will work on a purely syntactic level, because the "contents" of the nest instances in the network is never actually used, but rather the wiring structure itself is "abused" for representation of an algebraic structure. Probably "semantic" evaluation by message passing in a static network is also possible.

We conclude from this simulation that universal and compositorical genericity are sufficient to do anything that can be done with a computer. This fact could be regarded trivial because we could use universal interpreters inside bricks; this means that in general universal genericity alone would be sufficient. However, software engineering deals with *complexity of software structures* in practice. Thus compositorical genericity is added as a structuring principle in order to overcome the complexity problem of the software structure itself. We do not claim that compositorical genericity is always more elegant than extensional genericity in any application area, but we provide some evidence that it could be more elegant in many more cases than has been expected until now; further research is required to check that.

As another demonstration for the power of compositorical genericity, we simulate a key feature[4] of aspect-oriented programming (AOP, [K+97]; we assume the reader to be familiar with it): `strategy_combine` modifies commands for instantiation of bricks representing objects, say `Object_A` in such a way that the following pair of brick instances is always created in place of it:



The dashed surrounding box shall indicate that from a virtual viewpoint, the pair of instances is playing the same role as was originally planned for `Object_A` alone. By transparently pairing with `Aspect_B` (in general, we may exploit subsets of the cartesian product of various `Object_*` with various `Aspect_*`), we can dynamically create a combined functionality similar to AOP[5]. In contrast to most incarnations of AOP, the combination need not be executed at compile time, although (dynamic) code-generation for a combined `Object_A_Aspect_B`-brick in the style of macro or inline expansion and automatic replacement by an appropriate `strategy_replace` is possible in our system (as well as for performance improvement on arbitrary but static subnetworks).

Note that generalizations to bricks with multiple inputs and outputs (e.g. for simulating state in aspects by stateless bricks) or different types of inputs / outputs (interface types) are possible; we have just chosen to simplify our examples by treating only the basic idea.

Our FP and AOP simulation examples indicate the strength of the principles of universal and compositorical genericity. Note that simulation of basic OO inheritance is shown in [ST03], and we assume and expect that many other concepts can be simulated rather easily with brick networks, such as subject-oriented programming [SOP]. Thus

[4] We are not sure whether all details of AOP implementations such as described in [K+01] can be simulated with our approach (in particular, simulation of advanced join point models may become hairy); we leave this question for further research.

[5] We assume that only a fixed set of receptions-pointcuts exist which have been made explicit in the object interface, and that before-, after- and around-advice has been translated to ordinary procedures located in `Aspect_*`. Calls-pointcuts may be simulated by swapping `Object_*` with `Aspect_*`. We just focus on the basic principles here.

we supply high plausibility for the superiority of universal and compositorical genericity over extensional genericity in many cases. If we had employed conventional OO design methods instead, our problems would have tried to be solved very likely with extensional genericity, which would have led to much more complicated and/or less dynamic solutions.

5 Conclusions

We have presented means for reducing redundancy in software structures, motivated by strong economical arguments. We have also provided a rough taxonomy on that means which is open for future enhancement and improvement. We have suggested to change current nomenclature such that genericity is no longer a (near-)synonym for (variants of) polymorphism, but rather broadened over any part of a software system, whether static or dynamic, while polymorphism remains restricted to the area of type concepts.

We have argued and provided some evidence based on design examples and on general theoretical arguments, that universal genericity and compositorical genericity as defined by us should be prefered over extensional genericity when possible. This is in contrast to OO style of thinking. Although OO based methods such as generative programming [CE00] have recently focussed on a similar goal to reduce redundancy and have employed many variants of (object) composition, we go one step further by (1) emphasizing the importance of universal genericity, (2) conceptually subordinating extensional genericity (which could be regarded as the heart of OO), and (3) employing statelessness. In effect, we take OO no longer as vital basis. In our example design from the operating system area, we have avoided inheritance nearly completely, but rather used universal and compositorical genericity in preference.

We have presented general principles, but examples only from a special application domain. In order to become a mature discipline in software engineering, much more work is needed on universal and compositorical genericity, its applications, on appropriate infrastructures, and in particular on design methods leading to them.

References

[CE00] CZARNECKI, KRZYSZTOF and ULRICH W. EISENECKER: *Generative Programming*. Addison Wesley, 2000.

[Che87] CHERITON, DAVID R.: *UIO: A Uniform I/O System Interface for Distributed Systems*. Transactions on Computer Systems, 5(1):12–46, 1987.

[CW85] CARDELLI, LUCA and PETER WEGNER: *On Understanding Types, Data Abstraction, and Polymorphism.* Computing Surveys, 17(4):471–522, 1985.

[DDH72] DAHL, O.-J., E. W. DIJKSTRA and C. A. R. HOARE: *Structured Programming*. Academic Press, 1972.

[Fie88] FIELD, ANTHONY J.: *Functional Programming*. Addison-Wesley, 1988.

[Hot74] HOTZ, GÜNTER: *Schaltkreistheorie*. De Gruyter, 1974.

[HP94] HEIDEMANN, JOHN S. and GERALD J. POPEK: *File-System Development with Stackable Layers*. Transactions on Computer Systems, 12(1):58–89, 1994.

[IEC] *IEC Standard 61131-3*. `http://www.holobloc.com/stds/iec/sc65bwg7tf3/html/news.htm`.

[K+97] KICZALES, GREGOR and OTHERS: *Aspect-Oriented Programming*. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241. Springer-Verlag, 1997. `http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-ECOOP9%7/for-web.pdf`.

[K+01] KICZALES, GREGOR and OTHERS: *An Overview of AspectJ*. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241. Springer-Verlag, 2001. `http://aspectj.org/documentation/papersAndSlides/ECOOP2001-Overview.pdf%`.

[Mey88] MEYER, BERTRAND: *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Náv96] NÁVRAT, PAVOL: *A Closer Look at Programming Expertise: Critical Survey of Some Methodological Issues*. In *Information and Software Technology*, number 38. Elsevier, 1996.

[RT74] RITCHIE, DENNIS M. and KEN THOMPSON: *The UNIX Time-Sharing System*. CACM, 17(7):365–375, 1974.

[SG96] SHAW, MARY and DAVID GARLAN: *Software Architecture, Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SOP] *Homepage of the Subject-Oriented Programming Project*. `http://www.research.ibm.com/sop/`.

[SPvE93] SLEEP, M. R., M. J. PLASMEIJER and M. C. J. D. VAN EEKELEN (editors): *Term Graph Rewriting: Theory and Practice*. Wiley, 1993.

[ST02] SCHÖBEL-THEUER, THOMAS: *Skizze einer auf nur zwei Abstraktionen beruhenden Betriebssystem-Architektur: Nester und Bausteine*. Arbeitspapier und Vortrag auf dem Herbsttreffen der GI, Fachgruppe Betriebssysteme, Berlin, 7. – 8.11.2002.

[ST03] SCHÖBEL-THEUER, THOMAS: *Eine neue Architektur für Betriebssysteme*. Unveröffentlichtes Manuskript, 2003. Erhältlich auf Anfrage bei `schoebel@informatik.uni-stuttgart.de`.

[Szy98] SZYPERSKI, CLEMENS: *Component Software*. Addison-Wesley, 1998.

[Tho96] THOMPSON, SIMON: *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.