Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity

Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, Tillmann Rendel

University of Marburg, Germany

Abstract. What is modularity? Which kind of modularity should developers strive for? Despite decades of research on modularity, these basic questions have no definite answer. We submit that the common understanding of modularity, and in particular its notion of information hiding, is deeply rooted in classical logic. We analyze how classical modularity, based on classical logic, fails to address the needs of developers of large software systems, and encourage researchers to explore alternative visions of modularity, based on nonclassical logics, and henceforth called *nonclassical modularity*.

1 Introduction

Modularity has been an important goal for software engineers and programming language designers, and over the last decades much research has provided modularity mechanisms for different kinds of software artifacts. But despite significant advances in the theory and practice of modularity, the actual goal of modularity is not clear, and in fact different communities have quite different visions in this regard. On the one hand, there is a classical notion of modularity grounded in information hiding, which manifests itself in modularization mechanisms such as procedural/functional abstraction and abstract data types. On the other hand, there are novel (and not so novel) notions of modularity that emphasize extensibility and separation of concerns at the expense of information hiding, such as program decompositions using inheritance, reflection, exception handling, aspect-oriented programming, or mutable state and aliasing, all of which may lead to dependencies between modules that are not visible in their interfaces.

This work is an attempt to better understand the relation between these different approaches to modularity by relating them to logic. *Classical logic* is the school of logic prevalent in modern mathematics, most notably first-order predicate logic. We argue that the "modularity = information hiding" point of view is rooted in classical logic, and we illustrate that many of the modularity problems we face can be interpreted in a novel way through this connection, since the limitations of classical logic as a representation formalism for human knowledge are well-known. This is in stark contrast to the programming research community, in which information hiding is nowadays such an undisputed dogma

of modularity that Fred Brooks even felt that he had to apologize to Parnas for questioning it [9]. Our analysis of information hiding in terms of classical logic suggests that there are good reasons to rethink this dogma.

To make one thing clear upfront: We do of course *not* propose to abandon information hiding or modularity in general; rather, we suggest to investigate different notions of information hiding (and corresponding module systems), inspired by nonclassical logics, that align better with how humans structure and reason about knowledge.

Since there is no precise definition of modularity available, we will use the following working definition in the beginning: Modularity denotes the degree to which a system is composed of independent parts, whereby 'independent' implies reusability, separate understandability and so forth.

A concern is *separated* if its code is localized in a single component of a system, such as a file, a class, or a container. *Information hiding* denotes the distinction between the *interface* of a software component and its *implementation*. The interface specification should be weaker than the implementation so that an interface allows multiple possible implementations and hence leaves room for evolution that does not invalidate the interface. The interface specification being weaker also means that an implementation can have multiple interfaces at the same time; in particular, one can talk about the interface of a module to another module as its weakest interface necessary to satisfy the other module's needs [62]. An interface is also an *abstraction* of the implementation because it does not only hide (parts of) the implementation but also abstracts over it, that is, it allows reasoning on a more abstract level. For instance, rather than describing partial details of a sorting algorithm, it just states that the result is a sorted list.

A key question in information hiding is which information to hide and which information to expose. Parnas suggested the heuristic to hide what is 'likely to change' [58].

Modularity can also be viewed from the technical perspective of module constructs in programming languages. Module constructs typically enforce desirable properties such as separate compilation through type systems or other restrictions and analyses. While we appreciate the numerous wonderful works on module constructs, in this paper we want to discuss the more general question of how to organize, decompose, and reason about complex software systems, which is more basic than the question of how to enforce a given decomposition discipline by module constructs.

In the remainder of this paper, we formulate and defend the following five hypotheses:

1. The modularity and abstraction mechanisms that we use today are in deep ways tied to classical logic (and henceforth called *classical modularity* in the remainder of this paper; Sec. 2).
2. Classical modularity mechanisms and reasoning frameworks do often not align with how programmers reason about their programs (Sec. 3).

3. Successful information hiding is limited by the degree of separation of concerns, the inherent complexity of the system, and the need to support software evolution (Sec. 4).
4. The explanation for these problems is that programs are not like abstract, idealized scientific models – an analogy that has shaped the understanding of modeling in software development – but rather complex real-world systems (Sec. 5).
5. To overcome these problems, we have to weaken the assumptions of classical modularity and investigate notions of modularity based on nonclassical logics. Some existing attempts to escape classical modularity can be understood as being based on nonclassical logics (Sec. 6).

We conclude the paper with a proposal for a novel definition of modularity that makes the connection between a program and the logic in which we reason about its properties explicit.

2 Modularity and classical logic

The classical understanding of modularity is highly related to (and possibly shaped by) classical logic, and therefore, the basic principles and limitations of classical logic are relevant for modularity, too.

The spirit of classical logic is captured by the following quote from Lakatos:

> The ideal theory is a deductive system with an indubitable truth injection at the top (a finite conjunction of axioms) — so that truth, flowing down from the top through the safe truth-preserving channels of valid inferences, inundates the whole system. [37]

This view of logic and proofs can be distilled into a number of basic principles, such as

- *The principle of explosion:* Everything follows from a contradiction.
- *Monotonicity of entailment:* If a statement is proven true, its truth cannot be renounced by adding more axioms to the theory, because *proofs are for eternity*, and if we learn more, we do not have to revise earlier conclusions.
- *Idempotency of entailment:* A hypothesis can be used many times in a proof.

and also a number of "nonprinciples" or "don'ts", such as:

- *Inductive reasoning* – generalizing from examples – is unsound.
- *Reasoning by defaults* (such as "typically birds can fly") or *Occam's razor* (prefer the simplest explanation) is unsound.
- *Closed-world reasoning*, such as drawing conclusions by searching the knowledge database, is unsound.

These basic principles are common to all classical logics [21], that is, the logics most commonly used in mathematical reasoning since Frege's *Begriffsschrift*, most notably first-order predicate logic.

Although, nowadays, these and similar properties are often taken for granted, they are actually specific to classical logics. Nonclassical logics do not have all of these properties, or may allow reasoning using one of the aforementioned "non-principles". For instance, paraconsistent logics give up the principle of explosion, that is, a contradiction has only "local" consequences and does not render the whole theory trivial. Another example are nonmonotonic logics, which give up the monotonicity of entailment. A well-known proof rule which is nonmonotonic is negation-as-failure [11], which means that a proposition is considered true if its negation cannot be proven – an example of closed-world reasoning. A well-known logic that gives up idempotency of entailment and monotonicity is linear logic [22].

A fundamental concept in logic is the distinction between proof theory and model theory [27]: For a set of axioms A formulated in the logic we can, via the syntactic deduction rules of the proof theory of that logic, prove theorems $A \vdash T$. On the other hand, we have the semantic notion of a *model* or *structure* M of a set of axioms, which is a mathematical structure that satisfies all axioms: $M \models \phi$ for all $\phi \in A$ using an interpretation function that assigns mathematical objects to the symbols occurring in the axioms. These semantic and syntactic views are typically related by soundness and completeness theorems. A soundness theorem says that all theorems that can be deduced from the axioms (the *theory* of the axioms) hold in all models of the axioms. The completeness theorem says that every theorem that holds in all models can also be deduced.

How is this related to modularity? In the following, we discuss some principles that we believe to constitute our understanding of (classical) modularity as information hiding, and relate them to classical logic as described above. We do not claim that all these principles have necessarily been shaped with classical logic in mind, but we believe that classical logic is the best formalization of the notion of abstraction that connects all these principles.

2.1 Information Hiding and Abstraction

Information hiding is to distinguish the concrete implementation of a software component and its more abstract interface, so that details of the implementation are hidden behind the interface. This supports modular reasoning and independent evolution of the "hidden parts" of a component [58]. If developers have carefully chosen to hide those parts 'most likely to change' [58], most changes have only local effects: The interfaces act as a kind of firewall that prevents the propagation of change.

Abstraction can be seen as a different take on information hiding, focusing more on the removal of information and the generalization of concrete to parameterized components that can be instantiated again. This includes the idea of having more than one instantiation of the same abstract component at the some time, thereby promoting code reuse.[1]

[1] Parnas and his colleagues have often used the word *abstract interface* for what we call just *interface* [60, 8]; in Parnas' terminology, an interface is between two software modules and describes the assumptions one module makes about the other.

Both information hiding and abstraction imply some notion of *substitutability*: A module's implementation can be replaced by a different implementation adhering to the same interface, and since the implementation was hidden to other components in the system in the first place, these other components should not be disturbed by the change. Parnas was one of the first researchers to investigate this influence of information hiding on software evolution [58, 63], but the idea shows up in many different forms:

- In structured programming, control structures such as loops hide the details of control flow management. Compilers are then free to choose among different implementations of the control structure with low-level jumps.
- Procedural and functional abstraction hide the implementation of an algorithm behind a procedure or function signature/contract. A procedure or function can then be replaced by a different implementation of the same contract.
- In object-oriented programming, encapsulation can be used to achieve information hiding.[2] Objects of a class can then be replaced by objects of a subclass. The *Liskov substitution principle* [41] codifies this idea: The instances of a subclass should have the same observable behavior as the instances of the superclass when observed through the interface of the superclass, so that substitution of a subclass instance for a superclass instance does not change the observable behavior of the overall program.
- Data abstraction mechanisms hide the internal representation of an abstract data type, for instance, whether a complex number is stored in polar or Cartesian coordinates [67]. Logically, abstract data types are a form of existential quantification [48]. The internal representation of an abstract data type can be replaced with a different representation type supporting the same interface. Reynolds formalized and proved this property of abstract data types in his *abstraction theorem* [67].

The distinction between an interface and implementations of that interface, which is the at the core of information hiding and abstraction, is related to logic. The interface corresponds to a *set of axioms*, and the implementation of the interface corresponds to a *model of the axioms*. Substitutability is reflected by the fact that the same theorems hold for all models of the axioms (by soundness of the logic), hence we cannot distinguish two different models within the theory. The heuristic of hiding what is most likely to change is reflected by the design of axiom systems (say, the axioms of a group in abstract algebra) in such a way that there are many interesting models of the axioms.

2.2 Reductionism and Compositionality

Reductionism is the belief that a complex system can be understood completely by understanding its parts and the rules with which they are composed. This

[2] Encapsulation is a somewhat ambiguous term. We follow Booch's definition [5] here.

very general idea is not limited to software systems, and it has been described many times in the history of sciences, for instance, by Descartes [14]. A more recent take by Dawkins [13] describes *hierarchical reductionism* as the idea that complex systems can be described with a hierarchy of organizations, each of which is only described in terms of objects one level down in the hierarchy. For instance, a computer can be explained in terms of the operation of hard drives, processors, and memory, but it is not necessary to talk about logical gates, or even about electrons in a semiconductor. It is not surprising that this idea has been picked up and advocated in programming once program size became an issue [16, 59].

Reductionism is an implicit assumption underlying classical modularity: When analyzing a modular software system, we want to understand it in terms of our understanding of the modules that constitute the system [63].

In the context of language semantics, the ideas of reductionism have been formally stated as *compositionality*. A semantics is compositional if the meaning of a complex expression is fully determined by the *meanings* of its constituent expressions and the rules used to combine them, rather than by the constituent expressions themselves. Compositionality is deeply grounded in mathematics through its relation to the notions of structure-preserving mappings, that is, homomorphisms and morphisms in universal algebra and category theory respectively, since a compositional function preserves the structure of its argument, and conversely a structure-preserving mapping is compositional [49].

As in the case of information hiding and abstraction, compositionality implies a strong notion of substitutability: If a subprogram is substituted by a different subprogram with the same meaning, the meaning of the whole program will still be the same. In other words, we can successfully reason more abstractly on an expression by thinking of its meaning rather than of the expression itself. When reasoning about the program, we can identify expressions having the same meaning. This process is typically called *equational reasoning*. Since the actual expression is hidden behind its meaning, compositionality can also be seen as a specific form of information hiding by considering the meaning of a program to be its interface.

Classical logic reflects the ideas of compositionality and reductionism in two ways: First, classical logic is compositional in the sense that a subset of the axioms of the theory can be exchanged by other axioms if they are logically equivalent (which means that they have the same deductive closure) without changing the set of theorems that hold for the whole set of axioms. Second, the "meaning function" (such as: determining whether a formula holds in a specific model) of classical logic is also compositional, meaning that the truth value of a composite formula is determined by the truth values of its constituents.

In the field of programming, compositionality is the hallmark of denotational semantics [78] and initial algebra semantics [23]. The denotation of a program is an *abstraction* of the program: different programs (such as `1+2` and `3`) have the same denotation (such as the mathematical object 3). In this sense, the denotation can be understood as an abstraction of the program, or, conversely, a

program can be understood to be a model of its denotation. This may sound somewhat as if it is the other way around, since the denotation function maps syntax to semantics, but it makes sense if we consider the program to stand for its "actual denotation" when executed on physical hardware. The "actual denotations" of `1+2` and `3` are clearly different. They differ, for instance, in their power consumption or heat production of the CPU and required runtime. The difference between denotation and "actual denotation" hints at a principal limitation of compositionality: Those aspects of the "actual denotation" of a program that are abstracted over in its denotation may, to some user, be just as important as those that are reflected by its denotation. To take a practical example, whether a compiler of a programming language performs tail-call optimization [76] or not will often determine whether a program can be executed successfully or terminate with a stack overflow error. One can of course always enrich the semantic domain by more elements of the "actual denotation" (or abstractions of the "actual denotation", such as a specification of the space behavior of procedure calls [12]), but it is not clear when to stop enriching the domains, since different stakeholders need to work with different equivalence classes of programs.

2.3 Idealization

The notion of idealization can be traced back at least to Plato and his idea of *ideas* or *forms* [71]. He holds that there are abstract notions – ideas – that capture the essence of aspects of our real life, yet never actually occur in real life. For instance, there are no perfect circles in real life, yet we can talk about the idea of a perfect circle.

Idealization was used systematically by Galileo, who, in his study of bodies in motion, made assumptions such as frictionless surfaces and spheres of perfect roundness. The motivation for idealization is that actual scientific objects are too complicated, hence they need to be summarized to a few properties relevant to the phenomenon under study.

In the computer science community, Dijkstra motivates idealization from a modularity perspective as follows:

> A scientific discipline separates a fraction of human knowledge from the rest: we have to do so, because, compared with what could be known, we have very, very small heads. [17]

Modularization and idealization are hence rather similar ideas: To deal with complexity by being able to concentrate on those things that are relevant to the task at hand and to ignore the rest for the time being. Idealization is an implicit assumption underlying modularity: Our understanding (or, the interface) of a module is an idealization of the actual implementation of the module. Interfaces are an idealization in the sense that they assume that some aspects of the implementation are not relevant (such as whether a display update is triggered when calling an interface function), which may lead to false assumptions about the implementations of the modules [33].

The axiomatic method of classical logic described in the beginning of this section can be seen as a formalization of idealization, where the axioms play the role of an idea, and its models are the real-world objects captured by that idea.

2.4 Monotonicity

Monotonicity is the idea that we want to prove things "once and for all". It means that we never have to withdraw a conclusion when we learn more. For instance, if we establish a property of a software system in a monotonic logic, we never have to revise that property when more components are added to the system. As described above, monotonicity is one of the defining properties of classical logics.

Program logics such as Hoare logic [28] are typically monotonic: Enlarging the program does not invalidate what was proved about the contained smaller program. Also, operational models of programming languages can in most cases be considered a monotonic logic in the following sense: If we consider the equational theory implied by the operational semantics of the language, then typically we have the property that if $e = e'$, then $E[e] = E[e']$ for an evaluation context E that plugs the expression into a bigger program [50, 81]. This congruence property allows us to reason about the behavior of programs in a modular way: We do not have to revise our conclusions about program behavior when we enlarge the program or use it as subprogram in a bigger program.

2.5 Summary

The common notion of modularity, especially the facet of information hiding, is deeply related to classical logic. We take compositionality of abstractions and monotonicity in reasoning about them for granted. Classical logic shapes our thinking and expectation of modularity. However, as we will argue next, humans (and hence programmers) do not always organize and reason about knowledge in accordance with classical logic, which threatens the implicit assumption of classical modularity, namely that information hiding in the strong sense presented here is the best means to deal with software complexity.

3 Programmers use Nonclassical Reasoning

Although our modularity mechanisms are shaped by classical logic, programmers frequently reason about software systems in nonclassical logics. Programmers use *inductive reasoning*, use *default reasoning and Occam's razor*, and use *negation-as-failure and closed-world reasoning*, as we will illustrate. All these means of reasoning are unsound from the perspective of classical logic (and have been formalized in various nonclassical logics), but are still used in everyday development and maintenance tasks. Hence, classical modularity mechanisms frequently do not support programmers adequately when reasoning about their programs.

3.1 Programmers use Inductive Reasoning

Programmers routinely infer a general software property from observing individual cases. For example, from a lack of bugs in specific cases, developers tentatively infer the lack of bugs of a software. This is the essence of testing. From the perspective of classical logic, however, a successful test case shows nothing; only a failed test case produces new knowledge.

Similarly, developers sometimes explore the behavior of a software module by testing it on some inputs, and infer, through inductive reasoning, general laws on how the module behaves, especially when its APIs are underspecified. Alternatively, they might know the behavior of a module A only partially, use it to build module B, and later learn details about the behavior of A (for instance, corner cases) by testing the complete software.

The success of tools like Daikon [19] or the technique from Henkel and Diwan [26], which discover likely program invariants or algebraic specifications by inductive reasoning over test case results, also illustrates that inductive reasoning over programs *does* produce useful knowledge.

We could argue that inductive reasoning is unsound and programmers should not use it, but there are good arguments to the contrary. First, all basic theories in natural sciences are essentially the product of inductive (or abductive) reasoning; all theories in natural sciences can only potentially be falsified, but never be proven correct [64]. In that sense, inductive reasoning has a quite impressive track record. Second, inductive reasoning is natural human behavior. This hypothesis is supported by the basic learning mechanism of the human brain at the neuronal level, known as Hebbian learning [25]: Our brain learns correlation between different concepts and expects that this correlation will repeat in the future. Conditioned reflexes are a prime example of such learning process. Recent advances in computational neuroscience provide models that successfully explain many higher-level behaviors through the basic mechanism of Hebbian learning [47]. Detailed studies are available mostly for vision (for instance, illusory contours are explained this way), but brain processing uses the same fundamental processing mechanisms for all kinds of information; thus, researchers conjecture that all brain functions might be explained in terms of associative learning.

3.2 Programmers use Default Reasoning and Occam's razor

Programmers tend to use the simplest explanation they can imagine for an experienced phenomenon, such as a bug. Similarly, they tend to predict the simplest behavior consistent with the interface for a software entity, such as an API. Nevertheless, they regard such explanations and predictions as tentative, that is, programmers infer them nonmonotonically and revise them when contradicting evidence is discovered.

Default reasoning and Occam's razor are common in everyday development tasks; consider the following examples: (1) Developers might assume that an API function will not perform the side effect of formatting the harddrive or

modifying the value of the provided arguments unless it is explicitly specified. (2) When a program terminates by printing `"NullPointerException"`, developers typically assume a raised `NullPointerException` as cause, rather than a `println("NullPointerException")` instruction in the program. (3) Developers may expect that getter methods do not mutate the receiver object and can thus be safely invoked in a read-only fashion from multiple threads. (4) Developers might observe patterns in an API from a few of its members, and use such inferred patterns as default rules. Default reasoning is also acknowledged by design principles such as the *principle of least astonishment* [4], which recommends that the API should not contradict common predictions of the programmer.

Of course, these reasoning patterns can lead to invalid results when additional observations are made. This is actually quite common and may be the cause for some debugging efforts. For example, the second author shared the third assumption about thread-safe getter methods and had to revise his reasoning in a program using a well-known open-source library.

Again, we could put the blame for false preliminary conclusions on the programmer, but Occam's razor and default reasoning appear to be "hard-wired" human behavior, as also supported by the law of *prägnanz* in Gestalt psychology, which says that we tend to order our experience in a way which is regular, orderly, symmetric and maximizes simplicity [77].

3.3 Programmers use Negation As Failure and Closed-World Reasoning

Programmers often reason about a closed code base. For example, when removing a method that is presumably no longer necessary, they confirm that the method is actually no longer necessary by checking whether this method is still called in the current code base. These kinds of API changes are of course avoided if possible, if the API is used by a large number of applications outside the control of the programmer or company, but it is well-known that this means that APIs often become a kind of "software asbestos" [35, 3] or leads to versioning problems such as the infamous 'DLL hell', if the API change cannot be avoided.

Reasoning about callers of a method is an example of the negation-as-failure proof rule, which allows to deduce a property $\neg P$ from failure of proving P [11] (for instance, with $P =$ "*method* `foo` *is used*"). It is an example of closed-world reasoning [65] as well, because such (nonmonotonic) reasoning might be invalidated when the considered scenario is extended to elements allowing to prove P. Negation as failure and closed-world reasoning are both incompatible with classical logic.

Conflicts between classical logic and closed-world reasoning frequently arise during software evolution, especially in the context of APIs. Stable APIs are very difficult to achieve in open systems that might be extended by others (it is essentially impossible to ever remove any API functionality). Therefore, many developers are less strict about stability and information hiding and tend toward a closed-world assumption. For example, the Linux kernel developers do not guarantee stable APIs and instead strongly urge maintainers of external code to

submit that code for inclusion in the kernel, so it can be reasoned about and evolved together with the APIs in a closed-world fashion [36].

Closed-world reasoning is also required to establish many other important properties of software, for instance, temporal or concurrency properties. This holds for informal manual reasoning, but is even more obvious when one considers automated tools such as model checkers and static analyses, which assume a closed world when reasoning about source code. A property established in one code base may no longer hold in a larger versions of the program. For instance, suppose a module acquires lock A and then lock B (while holding lock A) and model checking ensures this module is safe; suppose then a new module is introduced, acquiring locks A and B in the reverse order. As we know, this will cause a deadlock, which will affect also the existing module. These limitations are of course well-known in these communities (e.g., [1, 40]); we mention them here to support our point that programmers and their tools routinely and successfully used inductive reasoning.

3.4 Discussion

We have shown that in many cases programmers use reasoning that does not align with classical logic and which causes problems in the context of classical modularity. One could ask, whether we should blame programmers or the modularity mechanisms. For example, we could blame programmers, because they are presumably just too lazy to use proper reasoning, or we could blame modularity mechanisms, because they do not support programmers adequately.

We take sides with the programmers for two reasons: First, we have evidence that various patterns of nonclassical reasoning are "hard-wired" into human intelligence. It seems natural to us that programming methodology should embrace rather than denunciate the way of reasoning humans are born with. Second, many important properties of programs just cannot be established using classical reasoning: Viewed as axioms of logical theories, module interfaces are highly incomplete, that is, for many propositions P neither P nor $\neg P$ can be proven classically (such as in the examples from above: $P =$ "*method* `foo` *is used*", or $P =$ "*the program is deadlock-free*"). Hence, programmers are essentially *forced* to use nonclassical reasoning, because there is no way to prove or reject many relevant properties with classical reasoning.

These two reasons suggests that the power of classical "modular reasoning" is rather limited, because it only works for so few properties. We believe that modularity mechanisms should be adapted accordingly to better reflect how programmers reason about code. In light of this finding, the next section will analyze the limitations of classical information hiding in detail.

4 Limits of Information Hiding

Information hiding is typically regarded as a core achievement and goal of modularity in the struggle to reduce complexity [63]. However, programmers often

experience limitations where information hiding is difficult or does not seem to pay off. In the following, we describe some situations where the limitations of information hiding become apparent.

4.1 Operational Behavior & Interface Detail

If a stakeholder wants to reason about "nonfunctional"[3] aspects of a system, such as time or space complexity or power consumption, he probably needs to reason about implementation details hidden behind abstraction barriers.

For example, when hiding the representation of complex numbers as either Cartesian or polar coordinates [67], the choice of representation is irrelevant from the perspective of Reynold's abstraction theorem, as already discussed in Section 2.1. However, the implementation choice makes a difference when executing the program on physical hardware. For example, different implementations have different time or space behavior of the operations, different rounding errors, different optimizations that the compiler will apply, or different power consumption. To some stakeholders, such concerns may well be important; while some require higher performance, others require higher precision.

To support the information needs of such a stakeholder, one could expose performance and precision information, for example, by adding additional constraints to the interface of the complex-number data type. But with each additional constraint, the possible implementations are more and more limited, until eventually all information is exposed, and just one possible implementation remains. By strengthening the interface, the distinction between interface and implementation is weakened, and information hiding is rendered useless. In logic, this situation is formalized by the notion of *completeness*, which denotes a logical theory that has just one model (up to isomorphism).

In a modular structure based on a nonclassical form of information hiding, it may be possible to establish additional interface properties by nonclassical (e.g., inductive or default) reasoning, without explicitly stating all of them in the interface. A concrete example would be a default rule which says that "getter" methods usually do not perform side-effects. An example of inductive reasoning would be to observe that many functions of an API that access the file system are not thread-safe, and generalize this finding to all API functions that access the file system. In the field of logic, the problem of having to state too many properties explicitly to reason about an 'API' is known as the *qualification problem*, which we will discuss in more detail in Sec. 4.5.

4.2 Large Systems

When information is hidden behind an abstraction barrier, there are potential stakeholders (or concerns), who are interested in that hidden information. There-

[3] Actually the word *nonfunctional* is a misnomer, since nonfunctional properties are just as important aspects of the function of a system as its "functional" properties. The wording is unfortunate, because it is an excuse to pretend that some aspects of a system can be ignored when "modeling" a system, see also the discussion in Sec. 5.

fore, the success of information hiding depends on whether such potential stakeholders (or concerns) and their information needs are relevant for the system or not – and, the larger the software system, the more likely a relevant stakeholder exists. In that sense, we argue that strict information hiding is problematic in large systems. This has strong implications on practice, as we exemplify with the Linux kernel.

Surrounding the origins of the Linux kernel there is a well-known debate about how to design an operating system kernel between Linus Torvalds, the original developer behind Linux, and Andrew Tanenbaum, an operating system researcher [15]. At the heart of the debate lies another debate about modularity. Academics argued that kernels of operating systems should be written using loosely coupled independent modules and that interface boundaries should be enforced through shared-nothing, message-passing-based concurrency (known as microkernel design [79]). In contrast, Linux uses a monolithic kernel design which does not enforce information hiding strictly.

In a nutshell, Torvalds' motivation for neglecting information hiding is that parts of the kernel are highly interdependent.[4] They require so much knowledge about each others implementation that there remains little to hide. Torvalds himself provides (among many others) the following example:

> This is an example of how things [different modules] are *not* "independent". The filesystems depend on the VM [Virtual Memory subsystem], and the VM depends on the filesystem. You can't just split them up as if they were two separate things (or rather: you *can* split them up, but they still very much need to know about each other in very intimate ways).[5]

So, programmers of the Linux kernel, and in fact of most other operating systems as well, accept a weaker form of modularity because so much information would have to be exposed in interfaces that information hiding does no longer add enough value.

A related issue of large systems arises with crosscutting concerns such as transactions or concurrency. The problem that such concerns are very hard to modularize with classical modularity mechanisms has been the motivation for aspect-oriented programming [32].[6] If such concerns are *not* modularized, however, a basic assumption of information hiding, namely monotonicity, does not hold anymore: Composing two programs which are each separately correct with respect to, say, lock-based concurrency or transactions, are in general no longer correct when composed. More importantly, the noncomposability can in general not be deduced from the interfaces of these components (or it is at least not clear

[4] Actually, also performance is a common argument for monolithic kernels. Although some modularity mechanisms may arguably add some performance penalties, we ignore this aspect to concentrate on the issue at hand.

[5] `http://kt.earth.li/kernel-traffic/kt20050103_289.html#1`

[6] There is no consensus whether AOP solves these problems (e.g., [34]) but this is not relevant to our point.

how to document the components in such a way that it is). Hence, the monotonicity assumption of classical modularity fails when concerns are not properly separated.

4.3 Separation of Concerns and the Dominant Decomposition

When taking the point of view that what is hidden behind an interface is (or belongs to) a concern, it becomes obvious that better separation of concerns reduces the amount of information hiding in the system. For instance, in the canonical AOP example of updating a display when a figure element changes [33], a figure element module hides less information behind its interface when the display updating logic is separated from the figure element module. In that sense, and contrary to the common notion that information hiding and separation of concerns go hand in hand, information hiding and separation of concerns can actually be contradictory.

The *tyranny of the dominant decomposition* [80] also reflects a major limitation of information hiding: What can be hidden behind an interface depends on the choosen decomposition, but there is no "best" decomposition; rather, from each point of view (such as the points of views of the different stakeholders) a different decomposition (and hence information hiding policy) would be most appropriate. What one stakeholder would hide as an implementation detail behind an interface is of primary importance to another stakeholder, who would hence choose a different decomposition that exposes that information.

4.4 Software Evolution

Even if a software system is successfully modularized, and the information needs of all stakeholders and concerns are reflected in the interfaces of components, information hiding might still hinder software evolution. This might be surprising at first, because information hiding is supposed to facilitate software evolution by hiding design decisions behind interfaces, so that they can be changed at will. The problem is that the original developers have to anticipate change and to modularize the software accordingly.

Unfortunately, it is not clear how to decide up-front which design decisions need to be hidden and which need to be exposed. Parnas heuristic of hiding what is most likely to change is difficult to follow.[7] If a design decision is exposed in the interface of a component, this aspect of the component cannot be evolved in a modular fashion later. But if the design decision is hidden behind the interface, software evolution might bring a new stakeholder (or concern) into the system which needs to access that hidden information. So, to support the information need of this stakeholder (or concern), the design decision should not have been hidden in the first place.

[7] The wording 'most likely' indicates that one has to use nonclassical – such as inductive or probabilistic but in any case nonmonotonic – reasoning to determine the modular structure of a system. Hence the illusion of staying within classical logic all the way through breaks together one way or the other.

An example for this situation is discussed in the aforementioned display-update example: When the `Point` figure element hides its update logic behind its interface, the system cannot evolve to support also a `Line` abstraction based on `Point`, since the update logic of `Line` cannot be implemented without detailed knowledge about the update logic of `Point` [33].

One could argue that successful modularization just needs better planning [61] to better assess what is likely to change, but we believe that this is an implausible assumption because large-scale software systems are assembled from many independently developed and independently evolving parts; hence, a big global "plan" is infeasible and unanticipated changes are unavoidable in long-living projects. In fact, the mere assumption of a monolithic global plan is contradictory to modularity.

4.5 Information Hiding and Classical Logic

As lesson, we infer from these examples that the larger and more complex a software system is, the harder a strict classical discipline of information hiding can be maintained. There are many concerns that, when separated, need to expose implementation detail in such a way that information hiding is impaired. Developers have to decide what information to hide and what to separate. This is a fundamental problem of classical modularity, which can be traced back to problems well-known in classical logic. For instance, the *qualification problem* describes the problem that

> in order to fully represent the conditions for the successful performance of an action, an impractical and implausible number of qualifications would have to be included in the sentences expressing them [44].

McCarthy gives the following example:

> The successful use of a boat to cross a river requires, if the boat is a rowboat, that the oars and rowlocks be present and unbroken, and that they fit each other. Many other qualifications can be added, making the rules for using a rowboat almost impossible to apply, and yet anyone will still be able to think of additional requirements not yet stated.

From the perspective of modularity, the qualification problem is clearly about information hiding, or more precisely, about the difficulty of information hiding in classical logic.

We believe that a possible solution can be to restrict the expectations of information hiding driven by classical logic (for instance, not to expect to prove program properties "once and for all"), and open us to less strict forms of information hiding, as we will discuss in Section 6.

5 Programs are not models

What is the cause of the failures of classical modularity discussed in the previous two sections? We believe that the answer to this question lies in the notion of

modeling and idealization from natural sciences (and, eventually from Plato's ideas and Aristotle's notion of *essence*), as discussed in Section 2: A scientific model[8] of a physical phenomenon removes information not relevant for the purpose of the model, and makes simplifying assumption to distill the core of the phenomenon the model is supposed to illustrate. It is not surprising that classical logic is a good match to describe natural scientific models, since historically, mathematics and logics were developed as auxiliary sciences to support natural science.

It seems tempting to assume that software is a model in that sense, too, and we believe that this is indeed a wide-ranging implicit assumption of many software researchers. Software engineering books talk about "modeling" all the time. There is even a branch of software engineering called "model-driven development", in which the actual programs are explictly called "models". The Object Management Group defines: "An object models a real world entity" [52], and the point of view that programs should model the real world has been quite important in Simula and the whole Scandinavian tradition to OO programming [42].

However, large software systems are not like that. They have to take into account the desires and needs of many different stakeholders. They have to deal and interact with the real world, which means that simplifying assumptions often turn out to be false. Instead of being like a scientific model, software systems are more like a mix of many overlapping and interacting models. One could of course say that a mix of overlapping models is another, more complicated model. But we believe that it is no longer useful to consider big programs to be models of a part of reality, but rather to *be* a part of reality. For instance, while at some early stage the programming concept of an order may have been a model of a hand-written document in a company, there are nowadays typically no artefacts beyond the record in the database that represent the order: It *is* the order. In contrast to a natural science model, a program is not describing a physical phenomenon – it *is*, when running, a physical phenomenon.

In natural sciences the problems of idealization and abstract models are well-known, of course. For instance, when trying to compute the movements of actual bodies in motion, aspects such as friction or air resistance have to be taken into account – the simplifying assumptions do not hold anymore. Taking all these additional influences into account turns Galileo's simple models into highly complex computations. Even in natural sciences itself, there is discussion about whether scientific models are really accurate descriptions of physical phenomena [10], and a process of de-idealization and de-simplification is proposed to turn the model into an accurate description of reality [46, 39].

The problem of multiple overlapping models, which manifests itself as the tyranny of the dominant decomposition in software (cf. Sec. 4.3), is also well-known in natural scientific modeling:

> All of our theories and models are tightened together only because they apply to the same empirical reality but do not enter into any further

[8] In this section we use the term *model* to denote *scientific model* and not as the term is used in logic, which is confusingly different.

> relations (deductive or otherwise). We are confronted with a patchwork of theories and models, all of which hold *ceteris paribus* in their specific domains of applicability. [24]

Complex, de-idealized patchworks of scientific models, as required for simulations of the real world, are much more akin to large software systems, since both have to deal with many aspects of reality and do not have the luxury to abstract over aspects that are inconvenient for a simple, elegant model.

The misleading analogy between programs and natural scientific models explains the failure of information hiding, since classical logic – the foundation of information hiding – is the framework in which scientific models are implicitly or explicitly formulated.

6 Towards Nonclassical Modularity

In the beginning of Sec. 1, we have pointed out several modularity mechanisms that can be used to improve extensibility or separation of concerns, but have been criticized for restricting information hiding and modular reasoning, for instance inheritance [75], aspect-oriented programming [2], reflection, aliasing and mutation [51, 54], multithreading [20], and exception handling [69].

Many of these modularity mechanisms can be understood to leave the "safe" world of classical logic, and indeed they can be understood to correspond to different nonclassical logics. Here are a few examples.

Aspect-Oriented Programming and Reflection. In earlier work, the first author has shown that aspect-oriented programming can be understood in terms of a nonmonotonic logic called *default logic* [56]. The idea is that one can reason by default that the semantics of a method call is to execute the corresponding method body, similar to how classes themselves can be interpreted as defining a default behavior that may be refined by subclasses (see discussion of inheritance below). Aspects that intercept such method calls are considered exceptions to that default rule. Hence, in this setting, one can – using defaults – reason locally about the program behavior. In case one learns later that the default assumption turns out to be wrong, there is a controlled process of updating the conclusions one has drawn from the invalid default assumption [56]. Using the logic proposed in this paper, one can establish a property such as "display updating is consistently applied when the data changes" modularly, by only considering the aspect that maintains this property.

Reflection (e.g., [74]) is also known to be a powerful modularity mechanism (e.g., [31]), but is in conflict with information hiding, since implementation details of foreign modules can be observed and modified. Not surprisingly, reflection is also frowned upon in classical logic ever since the paradoxes of naive set theory (Russel paradox, Cantor paradox, etc.) have been discovered – all of which rely on a form of reflection, namely self-application.

Aliasing and Mutation. The program-verification community has developed separation logic [68] to reason about programs using pointers, aliasing, etc., in a modular way. Separation logic is a nonclassical logic, since the structural rules of weakening (monotonicity of entailment) and contraction (idempotency of entailment) do not hold [53]. Instead, the so-called *frame rule* allows the programmer to reason about each routine separately, given that the parts of the heap that are modified by each routine are disjunct [68]. The frame rule solves a particular instance of the *frame problem* [45], which has been a major motivation for the development of many nonclassical logics and is at the same time a typical modularity problem, namely how to specify what a module *does not* do, without enumerating all possibilities [6].

Separation logic seems to be compatible with classical information hiding at first. However, the frame rule forces one to make all sharing and aliasing in the program explicit in the specification, which is contrary to the idea of using *implicit* communication via shared variables to reduce coupling and hence improve modularity [18, Sec. 2] [72, Sec. 4.8.2]. Specifying sharing and aliasing explicitly has a ripple effect, because typically the callers of the components that share variables have to know about this, hence the callers of the callers have to know, and so forth [72, Sec. 4.8.2], which means that the usage of separation logic in such cases becomes a form of closed-world reasoning. So one has only two choices: Either give up the modularity that can be gained by implicit communication, or use a stronger – unsound – form of the frame rule, similar to circumscription [44], that allows one to tentatively compose proofs of properties of program parts even if they do potentially communicate implicitly.

Inheritance. Ideas to understand classes in object-oriented languages as giving nonmonotonic default definitions that may be refined by subclasses are almost as old as object-oriented programming itself [66, 73]. More recently, variants of separation logic (see above) have been proposed to reason about object-oriented programs [57]. These logics illustrate that inheritance is a nonclassical modularity mechanism.

Temporal Logics. Temporal logics, such as linear-time logic (LTL), are a common formalism to reason about temporal properties of programs, especially concurrent programs [43]. Temporal logics assume a closed world, which means that the whole program (or state machine) to be verified must be fully known, and results established for one program do not automatically hold for extensions of that program (nonmonotonicity) or compositions of multiple programs [1].

Closed-World Modularity. Some approaches embrace closed-world reasoning and instead focus on tool support to dealing with nonmodular systems. For example, *FEAT* helps to discover and document scattered concerns and can afterward support reasoning about the still-scattered concerns (a closed knowledge base) by providing navigation support [70]. *Virtual separation of concerns* [30] emphasize editable views on code and whole-program analysis to reason about scattered implementations of a concern instead of enforcing a separation into modules.

Ideas of *effective views* [29] and *on-demand remodularization* [55] take this even a step further and actually rewrite the source code on demand to match the form of locality or information hiding that the programmer needs for a task.

Error Handling. Lanier remarked that software "breaks before it bends" [38]. That is, a single failure (such as a null-pointer access) in a minor part can cause inconsistencies in the whole program, just like a single inconsistency in a logical theory allows to prove every proposition (including contradictory ones). We believe this similarity is not accidental: Software inherits this property from the principle of explosion of classical logic. One of the common points is that both a software module and a logical theory require the perfect consistency (such as: freedom of bugs), even if software modules are in daily practice are rarely exempt from inconsistencies.

Interestingly, both in logic and in software different but similar means have been developed to deal with such explosions/crashes. In logic, the field of *paraconsistent logics* [7] deals with logics that are inconsistency-tolerant, that is, where one can still draw reasonable nontrivial conclusions even if there is an inconsistency in some part of the theory. This is similar in spirit to attempts in computer science to limit the effects of errors, such as null pointer errors or nontermination, that would otherwise destroy a running program immediately.

For instance, insulating faults by partitioning a software in different processes is a traditional best practice in the Unix culture, because it increases stability.

More recently, Martin Rinard and his group introduced the notion of *failure-oblivious computing* [69] whose idea is that applications should continue to produce reasonable results despite unexpected errors, and proposed innovative and even surprising techniques for doing so. These techniques are often met with resistance, because they change the local semantics of the program (for instance, by skipping loop iterations to improve performance). That is contrary to the spirit of classical modularity; however, the only alternative is the principle of explosion.

Nonstrict programming languages (such as Haskell) can also be understood to restrict the propagation of a common error, namely nontermination. The connection to paraconsistent logics becomes particularly obvious when one identifies inconsistency with nontermination, which is also suggested by the fact that the same symbol, $\bot$, is used to denote both inconsistency in logic and nontermination in denotational semantics.

Discussion. While the results discussed in this section are rather preliminary and require a more formal investigation, we still consider it striking that many program structuring mechanisms that have been criticized for violating information hiding are at the same time similar to developments in nonclassical logics. Also, the motivations for these developments are often similar as well, for instance, avoiding the propagation of errors for both error recovery mechanisms and paraconsistent logics, or avoiding an excessive number of qualifications for both aspect-oriented programming and nonmonotonic logics (cf. Sec. 4.5).

We believe that these similarities indicate that the programming community should acknowledge that programs are a form of *knowledge representation*, and the same considerations with regard to modularity, extensibility, ease of reasoning, and so forth, apply to both logics and programs. Until now, programming languages have usually been developed independently of logics, and logics to reason about various properties of the program have only been added as an afterthought. We believe that there is a lot of potential in the idea to make use of the connection between programming on one hand and logics and knowledge representation on the other hand, and develop modularity constructs and their logic side by side instead.

7 Conclusions

The traditional point of view on modularity as information hiding is deeply rooted in classical logic and inherits both its merits and limitations. As a device to structure the knowledge embodied by large-scale software system it is problematic, since there is a deep mismatch between the idealizing form of modeling for which classical logic was designed, and the multi-stakeholder reality of complex software systems. Some existing ideas to escape the limitations of traditional information hiding can be understood as being based on nonclassical logics. We propose to turn this observation into a principled design methodology for future modularity mechanism in which the modularity mechanism, its information hiding policy, and a corresponding (potentially nonclassical) logic are developed side by side. It does not make sense to judge a modularity mechanism through the glasses of a logic that does not match to the logic with which knowledge is organized in this modularity mechanism.

In fact, we believe that it is useful to adopt a novel definition of modularity[9] that takes the relation between programs, its modules, and the logic we use to reason about the program into account.

Instead of talking about modularizing *concerns* – a term that has often been understood in a rather syntactic way – we propose to talk about modularizing *properties* of a program. Since the way we establish a property depends on the logic, modularity is also relative to the used logic L. Hence we can define property P to be modularized in a program unit U (which we assume to include its interface to the rest of the program) of a program, if P can be proved from U in L, or, using formal notation, $U \vdash_L P$.

Under this definition, cohesion denotes that a program unit modularizes a single (or few) coherent program properties. Program units are coupled, if an important property can only be proved from a larger set of program units, or even the whole program. A perfect (perhaps unattainable) modularization is one where all properties required by the specification are modularized.

We hope this definition will help to correct what we perceive to be a *modularity bias*: That some desired properties – such as the aforementioned "functional"

[9] This direction was actually suggested by an anonymous reviewer of this paper.

properties of a system – are more important to modularize than other ("non-functional") properties.

8 Acknowledgements

We particularly thank David L. Parnas for feedback, historical notes, and interesting discussions about an earlier draft of this paper. We also thank the reviewers for quite helpful comments. The authors of this work are supported by the ERC Starting Grant No. 203099.

References

1. M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15:73–132, January 1993.
2. J. Aldrich. Open modules: Modular reasoning about advice. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 144–168, 2005.
3. T. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm. Study of an API migration for two XML APIs. In *Proc. Conf. Software Language Engineering (SLE)*, LNCS. Springer, 2010.
4. J. Bloch. How to design a good API and why it matters. In *Companion Int'l Conf. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 506–507. ACM, 2006.
5. G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, 2007.
6. A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Trans. Softw. Eng.*, 21:785–798, October 1995.
7. M. Bremer. *An Introduction to Paraconsistent Logics.* Peter Lang Publishing, 2005.
8. K. H. Britton, R. A. Parker, and D. L. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 195–204. IEEE Press, 1981.
9. F. P. Brooks. The mythical man-month: After 20 years. *IEEE Software*, 12:57–60, 1995.
10. N. Cartwright. *How the laws of physics lie.* Clarendon Press, 1983.
11. K. L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.
12. W. D. Clinger. Proper tail recursion and space efficiency. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 174–185. ACM, 1998.
13. R. Dawkins. *The Blind Watchmaker.* Norton & Company, 1986.
14. R. Descartes. *Discourse on the Method of Rightly Conducting One's Reason and of Seeking Truth in the Sciences.* Available online at `http://www.gutenberg.org/etext/59`, 1637.
15. C. DiBona, S. Ockman, and M. Stone, editors. *Open Sources: Voices from the Open Source Revolution.* O'Reilly & Associates, Inc., 1999.
16. E. W. Dijkstra. The structure of "THE"-multiprogramming system. In *Proc. ACM Symposium on Operating System Principles*, pages 10.1–10.6. ACM, 1967.

17. E. W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*, chapter EWD 447: On the role of scientific thought, pages 60–66. Springer-Verlag, 1982.
18. E. Ernst. Method mixins. In *Proc. Net.ObjectDays/GSEM*, pages 145–161. GI, 2005.
19. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.
20. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338:153–183, June 2005.
21. D. Gabbay. Classical vs non-classical logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming, volume 2*. Oxford University Press, 1994.
22. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
23. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1):68–95, 1977.
24. S. Hartmann and R. Frigg. Scientific models. In S. Sarkar and J. Pfeifer, editors, *The Philosophy of Science: An Encyclopedia, Vol. 2*, pages 740–749. Routledge, 2005.
25. D. Hebb. *The organization of behavior*. John Wiley & Sons, 1949.
26. J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *In ECOOP*, pages 431–456. Springer, 2003.
27. P. Hinman. *Fundamentals of Mathematical Logic*. A K Peters, 2005.
28. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
29. D. Janzen and K. De Volder. Programming with crosscutting effective views. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 195–218. Springer-Verlag, 2004.
30. C. Kästner and S. Apel. Virtual separation of concerns – A second chance for preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, 2009.
31. G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 481–490. ACM, 1997.
32. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
33. G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 49–58. ACM, 2005.
34. J. Kienzle and R. Guerraoui. AOP: Does it make sense? The case of concurrency and failures. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 113–121. Springer Berlin / Heidelberg, 2002.
35. S. Klusener, R. Lämmel, and C. Verhoef. Architectural Modifications to Deployed Software. *Science of Computer Programming*, 54:143–211, 2005.
36. G. Kroah-Hartman. The Linux kernel driver interface. `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/stable_api_nonsense.txt`.
37. I. Lakatos. A renaissance of empiricism in the recent philosophy of mathematics. *Br J Philos Sci*, 27(3):201–223, 1976.
38. J. Lanier. One half of a manifesto: Why stupid software will save the future from neo-darwinian machines. *wired 8.12*, 2000.
39. R. Laymon. Idealizations and the testing of theories by experimentation. In P. Achinstein and O. Hannaway, editors, *Observation Experiment and Hypothesis in Modern Physical Science*, pages 147–173. M.I.T. Press, 1985.

40. H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. *SIGSOFT Softw. Eng. Notes*, 27:89–98, November 2002.
41. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
42. O. L. Madsen and B. Møller-Pedersen. A unified approach to modeling and programming. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS'10)*, volume 6394 of *LNCS*, pages 1–15. Springer, 2010.
43. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1992.
44. J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.
45. J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.
46. E. McMullin. Galilean Idealization. *Studies in the History and Philosophy of Science*, 16:247–273, 1985.
47. R. Miikkulainen, J. A. Bednar, Y. Choe, and J. Sirosh. *Computational Maps in the Visual Cortex*. Springer, 2005.
48. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.
49. R. Montague. Universal grammar. In *Formal Philosophy*, pages 222–246, 1970.
50. J. H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.
51. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 158–185. Springer-Verlag, 1998.
52. Object Management Group (OMG). Object management architecture guide, ed 2.0, 1992.
53. P. W. O'Hearn, David, and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5:215–244, 1999.
54. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 268–280. ACM, 2004.
55. H. Ossher and P. Tarr. On the need for on-demand remodularization. In *Position Paper for Aspects and Dimensions of Concern Workshop, ECOOP*. Citeseer, 2000.
56. K. Ostermann. Reasoning about aspects with common sense. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 48–59. ACM, 2008.
57. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 75–86. ACM, 2008.
58. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
59. D. L. Parnas. On a "buzzword": Hierarchical structure. In *Proceedings of IFIP Congress '74*, pages 336–339. North-Holland, 1974.
60. D. L. Parnas. Use of abstract interfaces in the development of software for embedded computer systems. Technical report, NRL Report No. 8047, 1977.
61. D. L. Parnas. The secret history of information hiding. In *Software pioneers: contributions to software engineering*, pages 399–409. Springer-Verlag, 2002.
62. D. L. Parnas. Precise documentation: The key to better software. In S. Nanz, editor, *The Future of Software Engineering*, pages 125–148. Springer, Berlin Heidelberg, 2011.

63. D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 408–417. IEEE Press, 1984.
64. K. R. Popper. *Conjectures and refutations: The growth of scientific knowledge.* Harper & Row, 1968.
65. R. Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76, 1977.
66. R. Reiter. *A logic for default reasoning*, pages 68–93. Morgan Kaufmann Publishers Inc., 1987.
67. J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
68. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.
69. M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proc. Symposium on Operating Systems Design & Implementation (OSDI)*, pages 303–316, 2004.
70. M. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 406–416. ACM, 2002.
71. S. D. Ross. *Plato's Theory of Ideas.* Oxford University Press, 1951.
72. P. V. Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.
73. E. Sandewall. Expert systems. chapter Nonmonotonic inference rules for multiple inheritance with exceptions, pages 239–247. IEEE Computer Society Press, 1990.
74. B. C. Smith. Reflection and semantics in LISP. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 23–35. ACM, 1984.
75. R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *Proc. Conf. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 200–214. ACM, 1995.
76. G. L. Steele. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMDBA: The ultimate GOTO. Technical report, Massachusetts Institute of Technology, 1977.
77. R. Sternberg. *Cognitive Psychology.* Thomson Wadsworth, 2008.
78. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* MIT Press, 1977.
79. A. S. Tanenbaum, J. N. Herder, and H. Bos. Can we make operating systems reliable and secure? *Computer*, 39:44–51, 2006.
80. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119. IEEE Computer Society, 1999.
81. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.