ABSTRACTION: IMPOSING ORDER ON COMPLEXITY IN SOFTWARE DESIGN

Mary Shaw, Carnegie Mellon University

The success of a complex designed system depends on the correct organization and interaction of thousands, even millions, of individual parts. If the designer must reason about all the parts at once, the complexity of the design task often overwhelms human capability. Software designers, like other designers, manage this complexity by separating the design task into relatively independent parts. Often, this entails designing large systems as hierarchical collections of subsystems, with the subsystems further decomposed into sub-subsystems, and so on until the individual components are of manageable size.

For typical consumer products, the subsystems are physical components that can be put together on assembly lines. But the principle of hierarchical system organization does not require an assembly line. Simon[1] tells a parable of two watchmakers, Hora and Tempus. Both made excellent watches and were often visited by their customers. Their watches were similar, each with about 1000 parts, but Hora prospered while Tempus became progressively poorer and eventually lost his shop. Tempus, it seems, made his watches in such a way that a partially assembled watch fell apart any time he put it down to deal with an interruption. Hora, on the other hand, made stable subassemblies of about 10 parts and assembled these into 10 larger assemblies, then joined these to make each watch. So any time one of the watchmakers was interrupted by a customer, Tempus had to restart from scratch on the current watch, but Hora only lost the work of the current 10-unit assembly—a small fraction of Tempus' loss.

Software systems do not require manual assembly of parts, but they are large, complex, and amenable to a similar sort of discipline. Software design benefits from hierarchical system organization based on subsystems that are relatively independent and that have known, simple, interactions. Software designers create conceptual subassemblies with coherent, comprehensible capabilities, similar to Hora's subassemblies. But whereas Hora's subassemblies might have been selected for convenience and physical organization, computer scientists are more likely to create structure around concepts and responsibilities. In doing so they can often state the idea, or *abstraction*, that is realized by the structure; for example, the capabilities of a software component are often described in

[1]Herbert A. Simon, 1997, *Sciences of the Artificial*, 3rd Ed., MIT Press, Cambridge, Mass., pp. 188ff.

terms of the component's observable properties, rather than the details of the component's implementation. While these abstractions may correspond to discrete software components (the analog of physical parts), this is not necessarily the case. So, for example, a computer scientist might create an abstraction for the software that computes a satellite trajectory but might equally well create an abstraction for a communication protocol whose implementation is woven through all the separate software components of a system. Indeed, the abstractions of computer science can be used in non-hierarchical as well as hierarchical structures. The abstractions of computer science are not in general the grand theories of the sciences (though we have those as well; see Kleinberg and Papadimitriou in Chapter 2), but rather specific conceptual units designed for specific tasks.

We represent these software abstractions in a combination of notations—the descriptive notations of specifications, the imperative notations of programming, the descriptive notations of diagrams, and even narrative prose. This combination of descriptive and imperative languages provides separate descriptions of *what* is to be done (the specification) and *how* it is to be done (the implementation). A software component corresponding to an abstraction has a descriptive (sometimes formal) specification of its abstract capabilities, an operational (usually imperative) definition of its implementation, and some assurance—with varying degrees of rigor and completeness—that the specification is consistent with the implementation. Formal descriptive notations, in particular, have evolved more or less together with operational notations, and progress with each depends on progress with the other. The result is that we can design large-scale systems software purposefully, rather than through pure virtuosity, craft, or blind luck. We have not achieved—indeed, may never achieve—the goal of complete formal specifications and programming-language implementations that are verifiably consistent with those specifications. Nevertheless, the joint history of these notations shows how supporting abstractions at one scale enables exploration of abstractions at a larger scale.

Abstractions in Software Systems

In the beginning—that is to say, in the 1950s—software designers expressed programs and data directly in the representation provided by the computer hardware or in somewhat more legible "assembly languages" that mapped directly to the hardware. This required great conceptual leaps from problem domain to machine primitives, which limited the sophistication of the results. The late 1950s saw the introduction of programming languages that allowed the programmer to describe com-

putations through formulas that were compiled into the hardware representation. Similarly, the descriptions of information representation originally referred directly to hardware memory locations ("the flag field is bits 6 to 8 of the third word of the record"). Programming languages of the 1960s developed notations for describing information in somewhat more abstract terms than the machine representation, so that the programmer could refer directly to "flag" and have that reference translated automatically to whichever bits were appropriate. Not only are the more abstract languages easier to read and write, but they also provide a degree of decoupling between the program and the underlying hardware representation that simplifies modification of the program.

In 1967 Knuth[2] showed us how to think systematically about the concept of a data structure (such as a stack, queue, list, tree, graph, matrix, or set) in isolation from its representation and about the concept of an algorithm (such as search, sort, traversal, or matrix inversion) in isolation from the particular program that implements it. This separation liberated us to think independently about the *abstraction*—the algorithms and data descriptions that describe a result and its *implementation*—the specific program and data declarations that implement those ideas on a computer.

The next few years saw the development of many elegant and sophisticated algorithms with associated data representations. Sometimes the speed of the algorithm depended on a special trick of representation. Such was the case with in-place heapsort, a sorting algorithm that begins by regarding—abstracting—the values to be sorted as a one-dimensional unsorted array. As the heapsort algorithm runs, it rearranges the values in a particularly elegant way so that one end of the array can be abstracted as a progressively growing tree, and when the algorithm terminates, the entire array has become an abstract tree with the sorted values in a simple-to-extract order. In most actual programs that implemented heapsort, though, these abstractions were not described explicitly, so any programmer who changed the program had to depend on intuition and sketchy, often obsolete, prose documentation to determine the original programmer's intentions. Further, the program that implemented the algorithms had no special relation to the data structures. This situation was fraught with opportunities for confusion and for lapses of discipline, which led to undocumented (frequently unintended) dependencies on representation tricks. Unsurprisingly, program errors often occurred when another programmer subsequently changed the data representation. In response to this problem, in the 1970s a notion of "type" emerged to help document

[2]Donald Knuth, 1967, *The Art of Computer Programming*, Vol. 1, Addison-Wesley, Boston, Mass.

the intended uses of data. For example, we came to understand that referring to record fields abstractly—by a symbolic name rather than by absolute offset from the start of a data block—made programs easier to understand as well as to modify, and that this could often be done without making the program run slower.

At the same time, the intense interest in algorithms dragged representation along as a poor cousin. In the early 1970s, there was a growing sense that "getting the data structures right" was a key to good software design. Parnas[3] elaborated this idea, arguing that a focus on data structures should lead to organizing software modules around data structures rather than around collections of procedures. Further, he advanced the then-radical proposition that *not* all information about how data is represented should be shared, because programmers who used the data would rely on things that might subsequently change. Better, he said, to specify what a module would accomplish and allow privileged access to the details only for selected code whose definition was in the same module as the representation. The abstract description should provide all the information required to use the component, and the implementer of the component would only be obligated to keep the promises made in that description. He elaborated this idea as "information hiding." Parnas subsequently spent several years at the Naval Research Laboratory applying these ideas to the specification of the A7E avionics system, showing that the idea could scale up to practical real-world systems.

This was one of the precursors of object-oriented programming and the marketplace for independently developed components that can be used unchanged in larger systems, from components that invoke by procedure calls from a larger system through java applets that download into Web browsers and third-party filters for photo-processing programs. Computer scientists are still working out the consequences of using abstract descriptions to encapsulate details. Abstractions can, in some circumstances, be used in many software systems rather than custom-defined for a specific use. However, the interactions between parts can be subtle—including not only the syntactic rules for invoking the parts but also the semantics of their computations—and the problems associated with making independently developed parts work properly together remain an active research area.

So why isn't such a layered abstract description just a house of cards, ready to tumble down in the slightest whiff of wind? Because we partition our tasks so that we deal with different concerns at different levels of

[3]David L. Parnas, 1972, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM* 15(2):1053-1058.

abstraction; by establishing reasonable confidence in each level of abstraction and understanding the relations between the levels, we build our confidence in the whole system. Some of our confidence is operational: we use tools with a demonstrated record of success. Chief among these tools are the programming languages, supported by compilers that automatically convert the abstractions to code (see Aho and Larus in this chapter). Other confidence comes from testing—a kind of end-to-end check that the actual software behaves, at least to the extent we can check, like the system we intended to develop. Deeper confidence is instilled by formal analysis of the symbolic representation of the software, which brings us to the second part of the story.

Specifications of Software Systems

In the beginning, programming was an art form and debugging was very much ad hoc. In 1967, Floyd[4] showed how to reason formally about the effect a program has on its data. More concretely, he showed that for each operation a simple program makes, you can state a formal relation between the previous and following program state; further, you can compose these relations to determine what the program actually computes. Specifically he showed that given a program, a claim about what that program computes, and a formal definition of the programming language, you can derive the starting conditions, if any, for which that claim is true. Hoare and Dijkstra created similar but different formal rules for reasoning about programs in Pascal-like languages in this way.

The immediate reaction, that programs could be "proved correct" (actually, that the implementation of a program could be shown to be consistent with its specification) proved overly optimistic. However, the possibility of reasoning formally about a program changed the way people thought about programming and stimulated interest in formal specification of components and of programming languages—for precision in explanation, if not for proof. Formal specifications have now been received well for making intentions precise and for some specific classes of analysis, but the original promise remains unfulfilled. For example, there remains a gap between specifications of practical real-world systems and the complete, static specifications of the dream. Other remaining problems include effective specifications of properties other than functionality, tractability of analysis, and scaling to problems of realistic size.

---

[4]R.W. Floyd, 1967, "Assigning Meanings to Programs," *Proceedings of Symposia in Applied Mathematics*, Vol. 19-32, American Mathematical Society, Providence, R.I.

In 1972, Hoare[5] showed how to extend this formalized reasoning to encapsulations of the sort Parnas was exploring. This showed how to formalize the crucial abstraction step that expresses the relation between the abstraction and its implementation. Later in the 1970s, theoretical computer scientists linked the pragmatic notion of types that allowed compilers to do some compile-time checking to a theoretical model of type theory.

One of the obstacles to "proving programs correct" was the difficulty in creating a correct formal definition of the programming language in which the programs were written. The first approach was to add formal specifications to the programming language, as in Alphard, leaving proof details to the programmer. The formal analysis task was daunting, and it was rarely carried out. Further, many of the properties of interest about a particular program do not lend themselves to expression in formal logic. The second approach was to work hard on a simple common programming language such as Pascal to obtain formal specifications of the language semantics with only modest changes to the language, with a result such as Euclid. This revealed capabilities of programming languages that do not lend themselves to formalization. The third approach was to design a family of programming languages such as ML that attempt to include only constructs that lend themselves to formal analysis (assuming, of course, a correct implementation of the compiler). These languages require a style of software development that is an awkward match for many software problems that involve explicit state and multiple cooperating threads of execution.

Formal specifications have found a home in practice not so much in verification of full programs as in the use of specifications to clarify requirements and design. The cost of repairing a problem increases drastically the later the problem is discovered, so this clarification is of substantial practical importance. In addition, specific critical aspects of a program may be analyzed formally, for example through static analysis or model checking.

The Interaction of Abstraction and Specification

This brings us to the third part of our story: the coupling between progress in the operational notations of programming languages and the descriptive notations of formal specification systems. We can measure

[5]C.A.R. Hoare, 1972, "Proofs of Correctness of Data Representations," *Acta Informatica* 1:271-281.

progress in programming language abstraction, at least qualitatively, by the scale of the supported abstractions—the quantity of machine code represented by a single abstract construct. We can measure progress in formal specification, equally qualitatively, by the fraction of a complex software system that is amenable to formal specification and analysis. And we see in the history of both, that formal reasoning about programs has grown hand in hand with the capability of the languages to express higher-level abstractions about the software. Neither advances very far without waiting for the other to catch up.

We can see this in the development of type systems. One of the earliest type systems was the Fortran variable naming convention: operations on variables whose names began with I, J, K, L, or M were compiled with fixed-point arithmetic, while operations on all other variables were compiled with floating-point arithmetic. This approach was primitive, but it provided immediate benefit to the programmer, namely correct machine code. A few years later, Algol 60 provided explicit syntax for distinguishing types, but this provided little benefit to the programmer beyond the fixed/floating point discrimination—and it was often ignored. Later languages that enforced type checking ran into programmer opposition to taking the time to write declarations, and the practice became acceptable only when it became clear that the type declarations enabled analysis that was immediately useful, namely discovering problems at compile time rather than execution time.

So type systems originally entered programming languages as a mechanism for making sure at compile time that the run-time values supplied for expression evaluation or procedure calls would be legitimate. (Morris later called this "Neanderthal verification.") But the nuances of this determination are subtle and extensive, and type systems soon found a role in the research area of formal semantics of programming languages. Here they found a theoretical constituency, spawning their own problems and solutions.

Meanwhile, abstract data types were merging with the inheritance mechanisms of Smalltalk to become object-oriented design and programming models. The inheritance mechanisms provided ways to express complex similarities among types, and the separation of specification from implementation in abstract data types allowed management of the code that implemented families of components related by inheritance. Inheritance structures can be complex, and formal analysis techniques for reasoning about these structures soon followed.

With wider adoption of ML-like languages in the 1990s, the functional programming languages began to address practical problems, thereby drawing increasing attention from software developers for whom

correctness is a critical concern—and for whom the prospect of assurances about the software justifies extra investment in analysis.

The operational abstraction and symbolic analysis lines of research made strong contact again in the development of the Java language, which incorporates strong assurances about type safety with object-oriented abstraction.

So two facets of programming language design—language mechanisms to support abstraction and incorporation of formal specification and semantics in languages—have an intertwined history, with advances on each line stimulated by problems from both lines, and with progress on one line sometimes stalled until the other line catches up.

Additional Observations

How are the results of research on languages, models, and formalisms to be evaluated? For operational abstractions, the models and the detailed specifications of relevant properties have a utilitarian function, so appropriate evaluation criteria should reflect the needs of software developers. Expertise in any field requires not only higher-order reasoning skills, but also a large store of facts, together with a certain amount of context about their implications and appropriate use.[6] It follows that models and tools intended to support experts should support rich bodies of operational knowledge. Further, they should support large vocabularies of established knowledge as well as the theoretical base for deriving information of interest.

Contrast this with the criteria against which mathematical systems are evaluated. Mathematics values elegance and minimality of mechanism; derived results are favored over added content because they are correct and consistent by their construction. These criteria are appropriate for languages whose function is to help understand the semantic basis of programming languages and the possibility of formal reasoning.

Given the differences in appropriate base language size that arise from the different objectives, it is small wonder that different criteria are appropriate, or that observers applying such different criteria reach different conclusions about different research results.

---

[6]This is true across a wide range of problem domains; studies have demonstrated it for medical diagnosis, physics, chess, financial analysis, architecture, scientific research, policy decision making, and others (Raj Reddy, 1988, "Foundations and Grand Challenges of Artificial Intelligence," *AI Magazine*, Winter; Herbert A. Simon, 1989, "Human Experts and Knowledge-based Systems," pp. 1-21 in *Concepts and Characteristics of Knowledge-based Systems* (M. Tokoro, Y. Anzai, and A. Yonezawa, eds.), North-Holland Publishing, Amsterdam).