A Return to Elegance: The Reapplication of Declarative Notation to Software Design

David A. Schmidt

Computing and Information Sciences Department, Kansas State University, 234 Nichols Hall, Manhattan, KS 66506 USA.[1]

Abstract. Software design methodologies were introduced to manage the scale of complex systems built in imperative languages under demanding work conditions. To some degree, declarative notations have been ignored for systems building because they lack similar design methodologies. Methodologies useful to object-orientation, namely, software architectures, design patterns, reusable libraries, and programming frameworks, are proposed as a model to be imitated by the declarative programming community. The resulting "declarative design methodology" would hasten the reapplication of declarative notations to mainstream software design and implementation.

1 An Evolution of Programming

Certainly, there is no shortage of design methodologies for software, but at some point after the requirements descriptions, entity-relationship diagrams, and life-cycle models have been written, a programmer must convert descriptions into reality—into actual programs.

In recent years, this final stage—the writing of code—has been deemphasized; in particular, the insights of Dijkstra, Hoare, and Wirth regarding the refinement of specifications into correct, concrete code have been forgotten, and emphases have changed from "do it carefully" to "do it by deadline" and from "prove it correct" to "let the users and testing tools prove it wrong." The contemporary argument is that software is inherently too complex for programmers to understand, let alone prove correct, and only brute-force automated tools can "understand," that is, impassively locate errors. Regardless of its validity, the argument itself is dispiriting, for it implies that the human programmer can not understand what she is doing, let alone take satisfaction from knowing that she has completed the task correctly.

Why has the situation evolved so? There are certainly the pressures of the marketplace:

- The intensive demand for software, coupled with the competition to produce quickly new product to fill new niches, forces unhealthy deadlines onto programmers. It is as if a medical surgeon has been imposed with unrealistic quotas for completing surgeries with the justification that botched

[1] `schmidt@cis.ksu.edu`. Supported by NSF/DARPA CCR-9633388 and NASA NAG-2-1209.

surgeries will be detected later by patients and repaired by user support staff.

- The high demand for programmers has forced a larger percentage of the general population into the profession, at a time when education in mathematics and the hard sciences in the (United States') secondary school systems is suffering. It is as if demand for surgeons motivated training more and more of the general public in performing "elementary" surgeries.

Programming is a difficult intellectual endeavor, and attempts to produce programs quickly or train large segments of the populace at programming are as likely to succeed as efforts to do surgeries quickly or train the general population as surgeons.

But there are also deeper philosophical questions about the evolution of computing:

- Thirty years ago, programming methodology was based on the need to conserve expensive computing resources: A programmer was forced to understand a specification, hand code a solution to the specification, and hand test or argue correctness of the coding before supplying the program to a computer for a one-per-day execution. The resulting program was an entity unto itself, to be described, analyzed, and even proved correct before it was supplied to a computer for a reading. In contrast, modern-day computing resources are cheap and make possible programming methodologies based on experimentation: A programmer understands a specification by trial-and-error, fashioning experiments/scenarios-in-code whose repeated executions lead to repeated repairs until acceptable output is obtained. Such a program is an entity that evolves within and depends upon the computer, and such a program attracts no interest as a literary or mathematical artifact.

  Both methodologies have their merits and both methodologies can be abused, but the latter methodology must be accepted because it is here to stay.

- Unlike forty years ago, when programming was an esoteric activity, practiced by lonely specialists, it has evolved into an industrial activity that demands aspects of science, engineering, and architecture. The imprecise dividing lines between the scientific, engineering, and architectural aspects mean that a programmer specialized in one aspect of programming is asked to perform and even lead endeavors in the other aspects, often with disappointing results.

  Indeed, the tension between the science, the engineering, and the architecture of computing threatens to tear the field itself apart into three subfields. If this happens, to which subfield will programming belong?

Given these pragmatic and philosophical issues, it is no surprise that elaborate software design methodologies have been developed to keep some control over the current, unstable state of affairs.

2 The State of Declarative Programming

No doubt, the use of imperative notations for doing the actual programming of a large system has accelerated the development of software design methodologies—when one writes programs in a notation whose fundamental unit is a primitive manipulation of program state, one is forced to use a separate design methodology to express program structure that is otherwise buried within the sheer scale of the program itself.

In contrast, one sees no similar intensive development of software engineering methodologies in the declarative programming world, possibly because declarative notations do not perform low-level manipulation of program state—the crisis of program scale does not arise as quickly. Also, because it is state-free, declarative notation serves well as both a design notation as well as a programming notation. Hence, the design and programming processes merge, almost invisibly.

Such elegances should be advantageous, but they have a surprising drawback: Not enough attention has been paid to the architectural and engineering aspects of declarative programming. In this regard declarative programming is immature, evolving little from its original conceptions. If one was to list the major advances in declarative programming within the past twenty years, one might list specific language constructs such as arithmetical-and-subset contraints, pattern matching, and bounded subtyping rather than architectural or engineering advances such as "declarative entity-relationship diagrams," "declarative design patterns," and "declarative programming frameworks."

3 What Can Be Learned from Object-Orientation?

Perhaps the most interesting aspect of object-orientation is its attempt to address problems of scale and reuse within a universe of standardized, connectable components. In its ideal manifestation, object-orientation makes a programmer do most of her programming during the design stage, when an architecture is selected, components are precisely specified to instantiate the architecture, and connection points between the components are explicitly defined and connected. The final stage of implementation is reduced to the selection of off-the-shelf components that meet specification (or the coding of new components that provably meet specification).

Of course, the reality of object-oriented programming rarely matches this ideal, but there is much to admire, most notably the attempt to integrate architecture, engineering, and science in the programming process. To reduce this to a (slightly awkward) slogan, *using object-orientation, one programs by engineering scientifically-well-founded objects to fit into an architectural layout.* A pleasing consequence is that the Dijkstra-Hoare-Wirth approach to method specification and correctness is compatible with and even integral to the object-oriented programming process—*components must meet specifications.*

Important natural consequences of object-orientation have been the establishment of standard libraries of components, the documentation of "design patterns" of object combinations that let one assemble systems that match standard "software architectures," and the production of "frameworks," which are partially completed software architectures that can be customized for specific applications. None of these developments are language-based, nor are they necessarily paradigm-based—indeed, they take on an identity of their own.

4 A Possible Future for Declarative Programming

The elegance of declarative programming has never been called into question, but its applicability to real-life scenarios has. Some common complaints are

1. inefficient implementation, e.g., heap-storage management
2. lack of well-documented support libraries for common applications, e.g., graphical interfaces, networking, and file manipulation
3. lack of a "killer app," that is, an important application area that is ideally suited to computing by declarative means
4. a programming style that is too difficult for the average programmer to master

Some of these complaints can be refuted—consider Java's success as a refutation of Complaint 1—and others appear to have strong validity but are slowly being addressed (e.g., Complaint 2). Perhaps others are a matter of fate (e.g., Complaint 3), and others are speculation (Claim 4). But a new complaint must be added to the list:

5. lack of a mature software design methodology

It is time for the declarative programming community to adopt a design methodology that mimics the best aspects of existing methodologies, in particular, those aspects of object-orientation that emphasize integration of architecture, engineering, and science in the programming process. Such a methodology should encourage the description of standard declarative software architectures, the collection of useful design patterns, and the formulation and documentation of standard libraries for important application areas.

The benefits from these efforts will be at least two-fold:

- The efforts will foster the maturity of the declarative programming field.
- The efforts will establish a vocabulary that is understandable by the many in the mainstream software engineering community, easing adoption and integration of declarative programming into the mainstream.

Because of its elegance, declarative notation can serve as the language of specification (of architectures, of design patterns) as well as the language of implementation.[2] Such an elegance has a good chance at impressing practitioners of software design. And at the very least, those who already believe that declarative notations are merely "specificational" in nature can use the notations to devise specifications.

Regardless of the impact of these efforts on the software engineering mainstream, it is time for declarative programming technology to take the step towards codifying, documenting, and demonstrating its design techniques as a comprehensive "declarative design methodology."

5 Conclusion

It is discouraging that modern-day systems building is so complex as to make it difficult or impossible for any one person to understand, appreciate, and take pride in the completion of a program. To make systems building manageable and understandable by the individual, an elegant integration of the architectural, engineering, and scientific aspects of programming must be undertaken. Declarative programming methodology has much to offer in this regard, and the time is right for the reapplication of declarative notation, in the guise of a declarative design methodology, to the software development process.

6 References

- Timothy Budd, *Understanding Object-Oriented Programming With Java*, Addison-Wesley, 1998.
- Erich Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- Mary Shaw and David Garlan, *Software Architecture: Pespectives on an Emerging Discipline*, Prentice Hall, 1996. Or, David Garlan and Mary Shaw, An Introduction to Software Architecture. In V. Ambriola and G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, Singapore, pp 1-39, 1993. Also available as `http://www.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/intro_softarch.html`.

[2]But one should never underestimate the impact that a visual notation brings. Visual representations of declarative constructs become especially useful at the design level, and their development should be strongly encouraged.