Does Thought Crime Pay?

Gilad Bracha

gilad@bracha.org

Abstract

Who controls the past controls the future;
who controls the present controls the past.
– George Orwell [11]

We examine the past, present and future of radical innovation in programming languages. How did Lisp, Simula, Actors, Beta, Smalltalk and Self give us the world of C++, Java, Javascript, Perl, Python and PHP? We'll ponder such questions and speculate what new wonders await us down the road.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Object-oriented languages; Constraint and logic languages. ; D.3.3 [*Language Constructs and Features*]: Classes and objects; Modules, packages.

Keywords Programming languages, Objects

1. Introduction

Thought crime is the thinking of heretical thoughts; thoughts that question the assumptions that the majority never question. Innovation and thought crime are related. If you doubt this point, I refer you to the case of one Galileo Galilei.

Fortunately, today we know better. We live in a golden age of technological innovation. Modern society loves innovators - as long as they don't innovate too much. True innovation also involves questioning the assumptions that almost everyone agrees with. This can sometimes make those of us engaged in research feel a bit like thought criminals.

In my field, programming languages, there is a strong notion of *the mainstream*. We speak of mainstream languages, tools and practices. Of course, no one is going to send inquisitors to our homes to persecute us for disagreeing with the mainstream. However, you can run out of funding very fast.

It was not always so. Once upon a time, programming languages introduced new ways of thinking. Languages like Lisp, APL [9], Simula [5], Smalltalk [8], Prolog, Beta [10], Self [13] and Miranda [12] represented true innovation. Each such language opened a door into a new world, a different way of looking at problems. So did other languages like Forth, SETL, Snobol, Lucid and Esterel.

Today the programming language landscape is dominated by a small number of languages that represent an even smaller number of ideas. Advocating for anything that substantially deviates from the norm is a thought crime. In the words of Dijkstra [7]:

> If the truths are sufficiently impalatable, our audience is psychically incapable of accepting them and we will be written off as totally unrealistic, hopelessly idealistic, dangerously revolutionary, foolishly gullible or what have you.

How did we get here, and is this where we want to be? Some might argue that the languages of today evolved by a process akin to natural selection. Current languages would then represent a refinement of the languages of the past in response to the actual requirements of widespread deployment and industrial scale use.

However, the one thing we can say with confidence is that mainstream programming languages are inadequate in the face of the aforementioned requirements. There is constant stream of new versions of these languages, always adding new features. Yet, no matter how many new features are added, there is always a need for more.

Historically, today's mainstream languages represent a school of thought that constructs languages by agglomeration: specialized constructs are tacked on one by one, each tailored to some specific purpose.

These are the *languages of the present.* As noted above, there is little evidence that this approach ever converges to a satisfactory solution. I argue that we can ill afford to let the present control the future.

There is another school of thought in programming language design. It postulates that a language should have a small set of very general constructs rather than a large number of very specific ones. This approach is exemplified by the APL, Lisp and Smalltalk families of languages. Interestingly, these languages have required remarkably small adaptations over the past 30-50 years. Yet today, such languages tend to be relegated to small niches. These are *the languages of the past.*

Many innovations in implementation technology originally developed for non-mainstream languages have been commandeered by the mainstream. Examples include garbage collection (originating in Lisp), JITs (pioneered in APL and later in Smalltalk) and other aspects of advanced object-oriented runtimes (as introduced in Self and Smalltalk systems).

The mainstream also adopts language ideas from its less popular brethren. Object orientation is the most obvious example, coming from Simula and Smalltalk. In addition to the original class based form, we also have prototypes (Self and others). Other ideas include closures (Scheme), actors ([3]), IDEs (Smalltalk and Lisp), and reflection (Lisp and Smalltalk).

As ideas fllter through, they are often attenuated. For example, when Gabriel and Steele et al. debated whether object-orientation had succeeded [6], Gabriel argued that it had failed because the widely adopted version is such a pale shadow of the original ideas. It is small wonder that critics of object orientation are dissatisfied

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2508191

with it; they mistake the dull artifacts of the languages of the present for the real thing.

We can see the wages of sin: thought crime does pay; it just doesn't pay the criminals, it pays the broader society. The innovations of unpopular languages are adopted, albeit often in inferior form, by the popular ones.

Is this a satisfactory situation? Can we afford to to let the present control the past, relegating the most brilliant artifacts in the field of programming languages to the waste basket of history?

Perhaps this is as it should be. After all, aren't the ideas the important thing, not the specific artifacts in which they were first manifested? One problem with this argument that we still need significant new ideas. The mainstream is not a good incubator for new ideas; it is in fact rather toxic to them. As evidence, consider the list of innovations given above, produced by a very small number of people with relatively minimal resources. Compare it to the intellectual contributions of the mainstream, involving orders of magnitude more people and resources.

An example of what passes as innovation in the mainstream is the practice known as *dependency injection.* Dependency injection (DI) is an attempt to address modularity issues that have cropped up in industrial scale programming. Such modularity issues are best addressed at the language level; the designers of some DI frameworks acknowledge this [1].

The languages of the present, the mainstream, do not address the issue of DI, but neither do the languages of the past. We need *languages of the future* to tackle such questions. We can and have found inspiration for such languages by looking into the past. Combining ideas from Smalltalk and Beta, Newspeak shows how a very pure form of objects naturally answers these problems [4]. Combining the idea that everything is an object, with nested classes and the notion that all computation is based on exchanging messages among objects yields a powerful approach to modularity. Arguably it has been staring us in the face for thirty years.

These problems are not hypothetical. Today, we have examples of entire libraries that need to be evolved independently by multiple developers which are distributed geographically and organizationally. Consider the process of evolving web standards such as the DOM. At any given time, there are multiple proposals for distinct extensions to the DOM API. These proposals are prototyped independently by different teams in different places. To evaluate the proposals, others should be able to run the prototypes, both separately and together in various combinations to see how they interact. It is not attractive to build and/or download many versions of a complete web browser to achieve this.

An alternative is to extend and modify entire libraries, and compose the modifications by mixing and matching them in different ways. Javascript enables this through ad hoc and poorly structured modifications to prototypes. Other languages would be even more hard pressed to deal with the situation, but it can be handled gracefully in a language like Newspeak.

Programming tools and environments can also benefit greatly from looking to the past. We are beginning to see a resurgence in this area [2]. Tools that go beyond the view of programs as static source code to support rich interaction with running programs are desperately needed. Tooling needs to be treated as a legitimate area of research, not a thought crime. We need to understand that programming language design should not be divorced from the tools and environments in which these languages are to be used. Even more broadly, research in programming languages needs to re-emphasize the building of complete systems, just as in the glory days of Xerox PARC.

Logic programming is another old idea that is ripe for re-examination. The need to reason about large datasets should prompt us to revisit Prolog and similar languages. Today we have machines that are a thousand times faster than in the 1980s, and the ability to couple thousands of machines together. Not many conclusions are valid across six orders of magnitude; whatever difficulties the field encountered in decades are almost certainly irrelevant.

No doubt I have overlooked important languages and efforts in this brief summary. What is important, however is that we remain cognizant of the history of programming languages. In particular, education is crucial; students need to be aware of a many different ways of programming, lest they reinvent the wheel and reinvent it badly. We must not let the languages of the present obscure our view of the past, because it is the great languages of the past that can lead us to the languages of the future. Only then can we make thought crime really pay.

References

[1] Java on Guice: Guice user's guide. Available at http://code.google.com/p/google-guice/.

[2] First international workshop on live programming, May 2013. Held at ICSE 2013. See http://liveprogramming.github.io/2013/.

[3] G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.

[4] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in Newspeak. In *European Conference on Object-Oriented Programming*, June 2010.

[5] O.-J. Dahl and K. Nygaard. Simula: An Algol-based simulation language. *Communications of the ACM*, 9:671–678, 1966.

[6] M. Devos, B. Foote, R. Gabriel, and J. N. G. Steele. Debate at OOPSLA 2002. See http://www.oopsla.org/2002/ap/files/pan-1.html. See also http://www.dreamsongs.com/Essays.html.

[7] E. Dijkstra. How do we tell truths that might hurt? In *Selected Writings on Computing: A Personal Perspective*. 1975.

[8] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[9] K. Iverson. *A programming language*. Wiley, 1962. URL `http://books.google.com/books?id=zR81AAAAIAAJ`.

[10] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[11] G. Orwell. *1984*. 1949.

[12] D. Turner. Miranda: A non-strict functional language with polymorphic types. volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

[13] D. Ungar and R. Smith. SELF: The power of simplicity. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, Oct. 1987.