

# Extreme Terseness: Some Languages Are More Agile than Others

Stephen Taylor

Lambent Technology, 81 South Hill Park, London NW3 2SS  
sjt@lambenttechnology.com

**Abstract.** While XP principles are independent of the languages in which software is developed, we can distinguish properties of programming languages that affect the agility of development. Some languages are inherently more agile than others, and the experience of developing software in these languages reflects this. A family of languages descended from the mathematics notation developed at Harvard in the 1950s by Iverson<sup>1</sup> shares properties of extreme terseness and abstractive power with weak data typing. The history of software development in these languages foreshadows some of the characteristics of XP projects. To these linguistic communities, XP offers the prospect of rehabilitating styles of software development that fell into disrepute with the rise of software engineering. Conversely, these languages offer XP practitioners the possibility of radical condensation of the conversation between developer and customer.

## Introduction

Code volume affects key XP processes:

- *Communication* XP development places unprecedented emphasis on communication within a project and the use of code as a medium for it. The terseness of code is a reflection of a language's abstractive or expressive power. Terse code facilitates precise communication about processes and data structures in the same way that university educations and their associated vocabularies enable graduates to communicate precisely about ideas, theories and observations in their fields of study. (See Iverson on notation as a tool of thought.<sup>2</sup>)
- *Refactoring* Code volume is always a significant term in the cost of refactoring, and a term independent of complexity. Terseness contributes to XP's virtuous circle of simplicity. While more work, thought and time goes into writing a line of terse code than a line in a verbose language, the code volume ratio between terse and verbose code usually has the terse code finished first. A larger benefit comes from refactoring. The lower the code volume, the less code to review, analyse and rewrite. Terseness benefits refactoring even more than initial development.
- *Simplicity, quality, fun* While it is still possible to write verbose code in a terse language, the abstractive and expressive power of extremely terse languages permits solutions of startling and breath-taking simplicity. (These solutions are often correspondingly fast.) For example, a small spreadsheet, complete with GUI, can

be written as 3 lines of primitive K. A single 55-character expression in J calculates and plots a Mandelbrot set. The author recently used sparse-array techniques to optimise a memory-intensive APL process; the refactoring took an hour, writing and testing just four lines of new code.

## Communication

A recent project illustrates the effect that development in a terse language can have on communication between programmer and customer. (The project diary is available online at <http://aplxp.blogspot.com>.)

The project automates the preparation of letters describing the surrender value of pension policies. The processing rules are complex, to accommodate variations in legislative requirements and product specifications that have accumulated over decades. Where the sales people discard old products in favour of new ones, the maturities clerks never forget; they only accumulate more rules. New clerks take months to become proficient in interpreting the checklists that guide them through the processes.

Senior clerks train new clerks, and worked with me to identify the processing rules. Communication difficulties emerged immediately. While the trainers could *demonstrate* a process, attempts to reason about it or discuss hypothetical variations quickly exceeded their linguistic skills. I tried every kind of representation of the rules I could think of, but nothing communicated. Despite working directly with the customers, I couldn't verify any conceivable version of the rules except by coding it and asking for feedback.

So we did that, one case at a time. But using an APL interpreter enabled me to collapse an otherwise iterative process into an uninterrupted conversation.

The trainer picked a representative case; brought up the mainframe inquiry screens she used and showed me what data she was looking at. Using the APL interpreter's immediate-execution environment, it took me only minutes to replicate the data sets in ad-hoc structures in my workspace. Now she described what she did with the information, and I coded the rules. Generally, I spent less time coding a rule in APL than she spent describing it in English. I used her vocabulary to name the APL functions names. The result is that she reads the application code about as well as I read Italian: not precisely, but well enough to follow the gist.

When we completed the process I ran the code and compared the answer to hers. When we have a difference to reconcile, the APL interpreter allows us to step through the rules together, examining partial results to find where we diverge. About one time in three or four, the error is hers – the processes are *that* hard for humans to get right.

When we agree, I append the case to the acceptance test suite and we start another example, altering or expanding rules coded previously. Every time we finish a case, the test suite allows us to discover immediately if the revised rules broke earlier test cases, and to resolve any discrepancies.

The effect on our communication is radical. An entire cycle of me taking notes, writing code, her writing, loading and running acceptance tests and reporting problems, me coding corrections for her to retest – all that collapses into a single conversation, from which we emerge with both automated acceptance tests and also code that passes those tests. Moreover, my customer has unprecedented confidence in the

accuracy and completeness of our code, and our ability to change the rules and the tests at need.

When we don't know something we want or need to know, we make up our own answers. Human beings do that. XP counters with high-bandwidth communication between programmers, and the importance of an On Site Customer to make frequently and quickly the decisions that only Business should make.

Shortening communication paths in a project is a valuable and virtuous circle. Here APL's terseness and abstractive power enabled a significant shift in the communication process. Some *quantitative* changes are large enough to produce a *qualitative* change. That happened here, producing levels of *involvement* with and *ownership* of the software not previously experienced or exhibited by the customer.

## Languages Descended from Iverson's Notation

The languages listed below, descended from Iverson's original executable mathematical notation, all share the qualities illustrated above. Some of the benefits associated with agile development processes are already exhibited in the history of software development in these languages, as a reflection of the impact terseness has on the cost of change over time.

- APL
- A+
- J
- K

Links to resources for exploring these languages can be found at the web log diary for the project from which the anecdote above was taken: <http://aplxp.blogspot.com>.

## Conclusion

Terseness in languages particularly suits them to agile development. Extremely terse languages participate more fully in XP's virtuous circle of simplicity. Some less well-known languages offer order-of-magnitude improvements in terseness. Agile process managers looking for more edge will consider these languages as candidates for roles in some XP projects.

## References

1. Iverson, K.E.: A Programming Language, John Wiley & Sons, New York, 1962
2. Iverson, K.E.: Notation as a Tool of Thought, 1979 ACM Turing Award Lecture, Communications of the ACM, Vol. 23 (8), August 1980