Dynamic Technology Holdings

Programming in the 21st Century: Introducing Mathematical Methods

TALK PREVIEW

Abstract. Can mathematics help software developers write better programs in less time? We advocate development through formal methods — i.e. using mathematical techniques to implement *dependable* software. Our scientific approach to software development entails mathematical modelling (e.g. algebraic specifications) as well as functional programming — a paradigm that sees computation as the evaluation of *mathematical* functions. In our talk we will briefly demonstrate development through mathematical methods and contrast it with mainstream object-orientation, which will serve to lay the ground to an in-depth discussion of software development with mathematics and functional programming as well as the benefits that come with the approach: reduced development effort, superior product quality, lowered cost of maintenance, greater adaptability, and ultimately higher economic value of the software product.

Andrea Falconi
andrea.falconi@gmail.com

January, 2010

The Perennial Software Crisis. Our lives have come to depend on software for almost anything, yet software reliability is still a somewhat elusive target — software crashes, security exploits, and other debacles often make the headlines. Indeed, the sheer size and complexity of the systems we engineer today make it extremely hard (not to mention expensive!) to build a dependable product from the outset. Equally elusive has been the battle to reduce the software maintenance effort and accommodate an ever shrinking time to market. As the IT industry braces to face the challenge posed by the new multi-core hardware architectures, many have begun questioning the effectiveness of mainstream imperative, object-oriented engineering techniques.

We believe that much of the struggle comes from not using mathematical techniques to build software: it is common practice in our industry to specify requirements in plain English without giving precise definitions of the terms and their relationships; designs may be sketched out, but the properties of those models are never studied; a lot of implicit assumptions are made when coding, but it then becomes impossible to verify that they hold true in all circumstances; although a healthy trend has recently emerged to exercise the code through automated sets of test cases, developers never produce proofs that undesirable behaviour cannot occur. We believe that the new century will see a radical shift in the way we develop software: from qualitative, informal, ad-hoc thinking to a systematic approach in which software is developed using mathematical techniques, finally aligning software engineering to other well-established engineering fields.

Why Maths Matters. Mathematics and functional languages provide a powerful framework to model software problems, devise solutions, and *prove* them correct. Experience has shown that not only could the adoption of mathematical techniques boost team productivity by a factor of 10, but also result in superior software quality with very low defect rates and reduced maintenance effort, thus making the approach of utmost relevance to today's IT industry where timely delivery of reliable software is the exception rather than the norm and the maintenance effort can swallow up to 70% of the entire software life-cycle. In a nutshell,

$$maths + functional\ programming = higher\ economic\ value\ of\ the\ software$$

A Comparative Study. Over the years, we have conducted several software development experiments and comparative studies to demonstrate the potential of mathematical methods. Invariably, the approach based on mathematical methods always outperformed mainstream, object-oriented engineering techniques in terms of dependability, quality, and lower costs of production and maintenance of the software product. To provide a frame of reference, we present here some figures regarding the development of general-purpose libraries for financial interchanges based on the ISO 8583 standard. Using algebraic specifications and functional programming, we have developed our own tool for handling ISO 8583 messages. Far from being the usual proof-of-concept toy software, the tool, nicknamed "Eezi 8583", provides a high-level language to specify ISO 8583 messages from which validating parsers are automatically generated; because no programmer intervention is needed, the tool may well be used by analysts to produce an "executable" specification of the interface between two financial institutions. In this regard, Eezi

8583 surpasses in features even jPOS (www.jpos.org), a well-established ISO 8583 framework with several years of development under its belt.

In developing Eezi 8583, our intention has been to provide strong evidence of the benefits of the "mathematical approach" on a development ground familiar to enterprise application developers. Indeed, jPos is representative of typical problems in the domain of enterprise software: data transfer and validation, systems integration, and large, open-ended number of customisable "business rules". Interestingly enough, most enterprise software developers would maintain that problems in their domain (such as the above) have little or nothing to do with mathematics, unless of course the task at hand involves "crunching" numbers — e.g. computing the interest of a financial asset. But what would happen if, instead of modelling software using mainstream object-orientation, we looked at it through the lenses of mathematics? What if we replaced classes, objects, inheritance, etc. with functions, morphisms, natural transformations, and so on?[1] You can find below a comparison chart relating jPOS to Eezi 8583; we believe the numbers speak for themselves.

	jPOS	Eezi 8583	Improvement
Development Effort (man days)	130	18	7×
Size (1000 lines of code)	13	0.6	21×
Defects Before Release	65	2	32×
Automated Tests	183	75000	409×

- ☐ jPOS version 1.6.6 (revision 2858); compared code from `iso`, `iso.packager`, and `iso.validator` packages.
- ☐ Estimated effort to release fully debugged and tested code at 100 lines of code per day for jPOS.
- ☐ Estimated average of 5 defects per 1000 lines of code for jPOS.
- ☐ Actual figures for man days and defects given for Eezi 8583.

Thinking More Proficiently. This is evidence that mathematics and functional programming empower the software developer with an intellectual framework which provides a better context to perform thorough analysis, design, coding, and verification; moreover, continuous mathematical practice sharpens our capabilities of abstraction and logical thinking, which are critical to successfully carry out all of the above activities. As the above figures eloquently show, the net results of improving the thought process underpinning the whole business of problem solving are reduced development effort, superior product quality, lowered cost of maintenance, greater adaptability, and ultimately higher economic value of the software product.

[1] Indeed the list of mathematical constructions applicable to modelling and solving software problems is a very long one: the result of centuries of accumulated wisdom and perfected abstractions. This, of course, begs the question of why software developers should venture in the quagmire of object-orientation instead of standing on the firm ground of mathematics. . .