

<CTRL> + <ALT> + <TOOL PARADIGM SHIFT>?

Russ Freeman
Ergnosis Ltd
Third Floor, 14 King Square
Bristol, BS2 8JJ, UK
+44 (0)117 924 8915

russ.freeman@ergnosis.com

Phil Webb
Ergnosis Ltd
Third Floor, 14 King Square
Bristol, BS2 8JJ, UK
+44 (0)117 924 8915

phil.webb@ergnosis.com

ABSTRACT

Despite being laden with elaborate features and time-saving gadgetry, modern Integrated Development Environments (IDEs) continue to be little more than turbocharged text editors with loosely integrated compilers. Years of incremental evolution of these environments have created a species of large, lumbering, often incomprehensible products with little direct support for the day-to-day tasks that developers actually find themselves doing.

In our poster we outline a fresh tool platform – development of which is well under way – which defines a revolutionary new environment for software development. Program editing is considered first and foremost as a direct manipulation of a rich semantic model whose persistent representation as plain-text source code is somewhere between conveniently incidental and almost entirely irrelevant.

Categories and Subject Descriptors

D.2.3 [Software Engineering]: Coding Tools and Techniques – Object-oriented programming, Program editors.

D.2.7 [Software Engineering]: Distribution, Maintenance and Enhancement – Restructuring, reverse engineering, and reengineering.

General Terms

Design, Languages

Keywords

Refactoring, tools

1. BACKGROUND

To understand the problem with today's tools, we must remember that they are built on legacy concepts several decades old. Take compilation. Not so long ago, a programmer would launch a lengthy compilation and return an hour or so later to try out the result of a small change to her code. Compilation dominated the software development mindset – the notorious “edit-compile-debug” cycle.

In recent years, the compilation process has been gradually elided by continual advances in computing power. “Compilation” is now of little interest; it is as though the source code itself were directly executable. However, yesterday's economics is still reflected in the architecture of today's tools and in their key areas of proficiency – historically expensive areas such as compilation, text editing and command-line tool integration. Yet the relentless advance of Moore's Law (the doubling of computing power every

two years or so) means that the relevance of these competencies in today's tools will continue to erode.

We are therefore seeing a gradual shift in development tool success factors to new higher-level qualities such as support for program comprehension, agile development and semantic editing. Indeed, we anticipate that the core features of existing development tools will reach a critical state of oversupply within two years [2].

Of course, the productivity benefits brought by each generation of software tools – with each generation taking a step in the direction of greater abstraction – rarely translate directly into shorter product development times. Instead, we realize the benefits in other ways, usually by building more complex and powerful software. There is a ratcheting effect, customer expectations are raised and businesses are soon obliged to use the new technology simply to remain competitive.

Our plans are ambitious. Nonetheless they are achievable, and already substantially validated. Our poster will emphasise what we can do today, as well as hinting at what we will be able to do in the future.

2. SOURCE AS SEMANTIC DATABASE

The key step of treating textual editing as the manipulation of a rich semantic model enables an entirely new kind of development environment: one which is able to take advantage of the meaning and structure of the program in each and every aspect of its operation.

Mainstream tools today are painfully unaware of the semantics of the program under construction. Programmer's editors and IDEs force development in largely non-semantic terms – source files and lines of text. Modeling tools (CASE tools) suffer from the converse problem: they provide high-level modeling features, but rarely any way to directly execute the resulting model in a popular programming language.

Even today's leading-edge tools, most of which offer refactoring facilities, provide only weak semantic features poorly integrated with other aspects of the product. Why? Because tool vendors are struggling to make semantics “work” for architectures that were designed to address an entirely different set of problems. (The “bolted on” feel of refactoring features should be a giveaway smell to any software architect!)

From the outset, our architecture has been designed to treat program source code as a rich semantic database for which the choice of source files as a persistence format is essentially arbitrary. This core relational architecture allows views to be opened on any relation defined by the language specification,

rather than just physical containment (see Figure 1 for some familiar examples). Java is currently the only supported language, but the core infrastructure is language-agnostic.

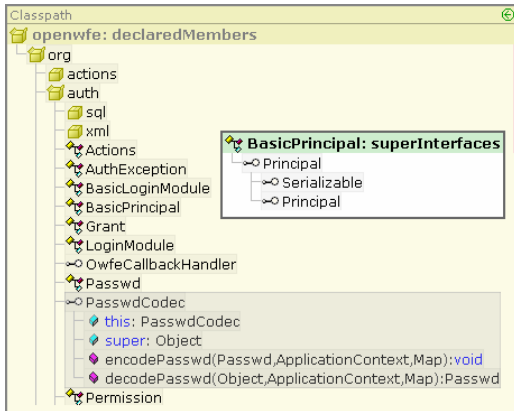


Figure 1. Relational semantic browsing

2.1 Refactoring micro-kernel

The first version of our tool platform will be a powerful validation of our semantic core, and will consist of a next-generation refactoring environment based around a kernel of low-level primitives which can be used to compose well-known existing refactorings like *Extract method* [4].

The potential ergonomic benefit of a kernel of low-level composable primitives is profound. Our research [1] suggests that a significant amount of program edits are in fact micro-refactorings, although they are often not recognized as such; other research [5] suggests that developers rarely use the larger and more complex refactorings their tools support. We believe this is simply because programmers are happiest carrying out stepwise refinement of their code, at least until they become competent with – and in the case of tools, trusting of –larger-scale transformations; a macro-refactoring will not be attempted as a single black-box operation unless its parts are well understood. This in turn will only happen if those parts can be separately executed and their independent value appreciated.

Our refactoring kernel is built on a language-independent semantic abstraction layer which will allow us to bring the same benefit to each language we support. Initially the focus is Java and we are today in a position to demonstrate the power of the new approach. In a nutshell, the kernel allows the developer to push and pull fragments of code along edges of control flow, introducing or eliminating degrees of freedom in a deep and intuitive way not possible with today's refactoring tools.

2.2 Direct manipulation

Our philosophy is that the essential interaction a developer needs to have is with the program being built – not with the tool itself. Yet most IDEs still require the developer to view and edit their program *through* windows, *via* menus, etc., inducing a feeling of detachment from the underlying software. In our view the program, along with the language it is written in, *is* both the tool and the user interface. (This is perhaps the main thing that distinguishes us from Simonyi's vision for intentional programming: we push programming languages to centre stage, rather than consigning them to the rubbish bin.)

Our approach means that virtually every view of the software will be directly editable, and moves us at last away from our long-standing obsession with source files – while storing source code in files for compatibility for as long as our users require it.

And by cleanly separating syntactic and semantic layers, we can easily switch between logical and physical views of the code (see Figure 2), blurring the traditional distinction between code editors and design tools.

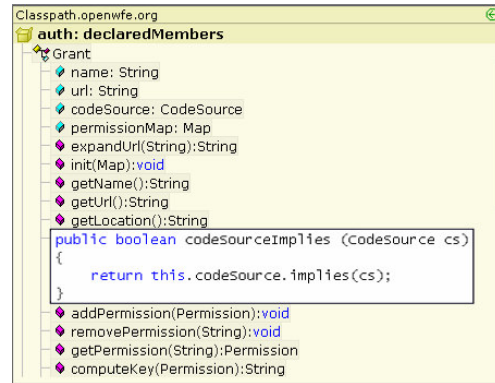


Figure 2. Syntactic view of method inline with semantic view of class

3. SUMMARY

By building a tool on powerful but transparent semantic infrastructure, we challenge traditional ways of *thinking about* software and software development, allowing a greater focus on more interesting design problems. No longer will renaming or moving a program element be considered a genuine *development* activity. Programming can become more about domain-specific modelling. (In this respect at least we share Simonyi's vision.)

For mainstream languages, such as Java and C#, tool support is improving, but vendors are still stuck in a local maximum of goodness-of-fit. We believe the only way to go that extra step by and move tools to the next generation is to discard most of what has gone before and build tools from first principles.

4. REFERENCES

- [1] Perera, R. *Refactoring: to the Rubicon...and Beyond!* Companion of the 19th annual ACM conference on Object-oriented programming, systems, languages, and applications (Oct 2004)
- [2] Boekhoudt, C. *The Big Bang Theory of IDEs* ACM Queue vol. 1 no. 7 (Oct 2003)
- [3] Janzen, D. and De Volder, K. *Programs as Information* Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange (Oct 2003)
- [4] Fowler, M. *Refactoring* Addison-Wesley, Reading, MA, 1999
- [5] Salinger, S. and Jekutsch, S. *Did the availability of refactoring functionality of IDEs result in refactorings being performed more frequently?* <http://www.inf.fu-berlin.de/inst/ag-se/teaching/survey/04-01/fragebogen.php>