JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 1, January-February 2005

Refactoring as Meta Programming?

Dave Thomas,
Bedarra Corp., Carleton University and University of Queensland

1 REFACTORING – A BEST PRACTICE IN SOFTWARE DEVELOPMENT

Refactoring [1] is widely acknowledged as one of the best practices of OO programming, and has been practiced in the functional and procedural community in one form or other for many years. Refactoring is a process that takes an existing program and improves it by transforming the program into a new program that is an improved version of the initial program.

The improvements typically eliminate redundancy, improve maintainability and may improve performance and reduce space. In most cases, a refactored program has less code bulk than the initial program. Ideally, one would like to be able to take a working, but complex, application and refactor it to clearly show the various aspects that have been accidentally interwoven by the developers. I look forward with great anticipation to a true aspect refactoring browser.

2 LANGUAGE AND TOOL IMPACT ON REFACTORING

Refactoring has always been much more widely practiced in high-level languages such as Scheme and Smalltalk. These so called dynamic languages feature incremental programming support, minimal syntactic baggage, and simple compile time-type checking and access to the internal representation. Methods are also typically short and simple, relative to those written in a procedural style. Hence, they are much easier to refactor.

Language technologies such as C++ or even Java or C#, which lack incremental support and refactoring browsers, present challenges for refactoring. They require much greater discipline and care when refactoring large application frameworks. Strong skills with tools such as Emacs are considered essential, as is pair programming and comprehensive test suites.

Unfortunately, most IDEs/compilers do not like to see anything but well-formed programs. This often forces developers and/or tools to artificially introduce syntax and semantics to keep the compiler happy during the refactoring process. Further, IDEs must manipulate source, binary and memory resident representations so even simple operations such as renaming can be expensive.

Modern IDEs such as http://www.eclipse.org further support the refactoring activity by providing tools such as Refactory Browser [2]. The refactoring browser makes it easier to apply, track and undo refactorings. Indeed, many claim they would not consider frequent refactoring without access to such a tool!

3 REFACTORING LARGE APPLICATIONS IS WIZARDS' WORK

Refactoring is a manual process that applies a series of non-equivalence preserving transformations [1] to the program being refactored. It is well known that refactoring large, complex frameworks is a high-risk activity. For this reason many large frameworks are not refactored as often as they should be, further increasing the risk associated with refactoring them later. Evidence suggests that while most modern developers are trained to refactor, in practice, it still remains wizards' work due to the risks associated with it and the large amount of context that must be carried in the head of the developers.

A major refactoring is almost always best done as a pair programming activity to reduce risk and manage the complexity. Test driven development with comprehensive test cases substantially mitigates the risks. These, too, need to be refactored, but can also be a source of errors.

While incremental IDEs and refactoring browsers help, false steps are still very common. Even in an incremental environment, simple mistakes can cause frustrating recompilations or-worse-result in the need to back out multiple changes.

Finally, while refactoring tools address programs, they seldom address persistent information associated with the current and transformed program. This makes refactoring in the context of an executing image of a non-stop system or a database even more challenging.

4 PROGRAMMING TO UNDERSTAND PROGRAMS

Recently, it has been realized that the popular browsers and debuggers of modern IDEs are inadequate to work with very large bodies of code. Developers need to have much more information about the program, especially a body of code that one is not intimately familiar with, in order to understand the challenges and opportunities for improving it via refactoring.

Researchers have therefore developed tools for understanding large programs, including visualization of static and dynamic structure and behavior and, more recently, IDE-based query tools such as Jquery [3]. While there is little experience as yet with querying programs, largely due to the awkwardness of expressing the queries, it seems that some form of interactive query refinement process holds promise to allow more generic queries to be refined based on inclusion or exclusion of specific contexts. Research with Graphlog [4], for example, allows developers to understand large, complex programs to look at the impact of refactoring.

5 REFACTORING AS META PROGRAMMING

Once we accept that it is useful to write programs (queries) to understand programs, it is a natural progression to think about other meta programs that would be useful. We already have examples in program generators/transformers/weavers such as those being advocated for MDA and AOSD. These tools help to create or recreate a program from higher-level programs/models/concerns. The focus is on getting it right up front, with little support for incremental refinement. Unfortunately, they do not help the developer who must refactor a large application.

We conjecture that it may be fruitful to look at refactoring as a domain-specific programming language for making specific program transformations. Further, a programming environment that readily supported such a language would most certainly support a wide variety of tools for program understanding and development.

A refactoring language should allow the developer to express complex queries and program transformations. This would considerably facilitate continuous program improvement. What would such a language look like? Clearly, it would need to allow one to express current refactorings [7]. We need to be able to manipulate package, class, interface, method and variable definition and use sites. We also need to be able to split and combine program fragments. If one looks at recent research in tools dealing with components [5] and aspects [6] transforming programs at load time or runtime, we see similar vocabulary in use.

The ability to treat refactorings as programs would allow one to clearly understand what was done at each refactoring session. It would allow a refactoring to be edited, applied, undone, etc. without going through the often tedious WYSIWYG process supported by a refactoring browser. This would substantially reduce the risk associated with a major refactoring effort. Refactoring programs could be validated by refactoring compilers to determine the impact of changes and ensure correctness of resulting programs.

6 REFACTORING AND PERSISTENT INSTANCES

In addition to applying transformations to the program there should be operations for dealing with persistent representations of class and instances. To support persistent data, there needs to be operations on memory and/or disk-based instances. In the case of object-relational application, for example, this would require invoking a relational database restructuring tool to change the schema and tuples. In the case of serialized objects, it would require that these objects be mutated on disk or when they were materialized in memory. The latter approach was used in many Smalltalk systems such as ENVY/Developer where the class/instance serializers would automatically mutate instances to match the current shape of the class and execute fix-up methods at load/runtime.

7 COMPLEX AND NON-EQUIVALENCE PRESERVING REFACTORINGS

While in principle one would like to have all refactorings equivalence *preserving* (i.e. the test cases still run correctly) the reality is that many refactorings *require*. Assuming the test cases use Junit http://www.junit.org and Fitnesse http://www.fitnesse.org, one would expect to be able to apply specific refactorings to test cases appropriately when a non-equivalence preserving refactoring is applied. The reality is that many large refactorings require major code restructuring. This restructuring takes the form of a sequence of refactorings that will take the code through states where it can't even be compiled correctly, but after a sequence of refactorings the code is returned to a stable, compilable state. In order to support this common practice, the refactoring system must be able to deal with broken programs until a transformation is complete. This *transactional refactoring* will typically require locking the code base, and turning off, recompilation and deferring test case execution until a complex refactorying is committed.

8 SUMMARY

Given the importance of refactoring in OO development and the need to manage the evolution of large software systems, it seems worth exploring a concise domain-specific language for such program transformations. Given the transformations share much in common with component integration [5] and load/runtime AOP [6], it should be possible to share a common infrastructure and language to define and apply the transformations.

REFERENCES

[1] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley. http://www.refactoring.com/

[2] Refactoring Browser, John Brant, http://www.refactory.com/RefactoringBrowser/Refactorings.html

[3] Doug Janzen and Kris De Volder, “Navigating and Querying Code Without Getting Lost”, *Proceedings AOSD 2003* http://www.cs.ubc.ca/labs/spl/projects/jquery/

[4] Mariano Consens, Alberto Mendelzon, and Arthur Ryman, “Visualizing and querying software structures”, *Intl. Conference on Software Engineering*, pages 138 156, 1992.

[5] Ralph Keller, Urs Hölzle, “Binary Component Adaptation”, Lecture Notes in Computer Science, 1998.

[6] Shigeru Chiba and Muga Nishizawa, “An Easy-to-Use Toolkit for Efficient Java Bytecode Translators”, *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, pp.364-376, Springer-Verlag, 2003. http://www.csg.is.titech.ac.jp/~chiba/javassist/

[7] Martin Fowler, Catalog of Refactorings, http://www.refactoring.com/catalog/index.html

About the author

Dave Thomas is CEO of Bedarra Corp., Adjunct Professor at Carleton University, Canada and University of Queensland, Australia, founding Director of AgileAlliance.com, and founder of Object Technology International. Bedarra works with research labs and commercial partners to transition innovations into products and practices.