

# Software Development is Program Transformation

Ralph E. Johnson  
Department of Computer Science  
U. of Illinois at Urbana-Champaign  
rjohnson@illinois.edu

## ABSTRACT

Since most programmers are working on software that they did not start, their view of programming is that it is the process of converting one version of software to the next. In other words, software development is program transformation. This paper describes some of the implications of this point of view.

## Categories and Subject Descriptors

D.2 [Software Engineering]

## General Terms

Design

## 1. INTRODUCTION

Software design is usually discussed as if the system is being created “de novo”, but most systems are modifications of earlier systems. Most software systems are created by modifying earlier source code; For example, Word 2003 was derived from Word 2002, which was derived from Word 2000, which was derived from Word97, which was derived from Word95, which was derived from Word 6, which was derived from Word 2, which was derived from Word 1. Other times, the source code is not reused, but the new system is clearly derived from the old one. For example, Linux is derived from Unix, but its source code was created independently by people who either knew Unix or who read Unix documentation.

Most documented software development processes, such as the Rational Unified Process, take requirements as input and then create various design models, code, tests, and documentation. They assume that the normal case is for a system to be created from nothing. Changes to an existing system are called “software maintenance” or “software reengineering”, and are treated as an exception to the normal case. Software engineering textbooks first describe how to gather requirements and how to specify systems, assuming that new systems are being build, and only discuss maintenance in a later chapter. However, most of the work (and cost) of software is after the first release, i.e. during maintenance. Redesign is normal, and software design “de novo” is the exception.

Even in the non-software world, what we call “design” is the process of changing an existing design. This is not only true with complex objects like the Space Shuttle or a skyscraper, it is also

true with simpler objects. Petroski describes the history of the design of a number of different objects, such as paper clips, pencils, and zippers [1], and shows how designs change gradually due to changing needs and new technology.

What are the implications of thinking that design is redesign? One implication is that we need tools to understand existing software systems, and tools to manage the history of software. But there has already been a lot of work on these tools. Another implication is that we need tools for transforming existing software. There has been much less work on these tools. I have been working on refactoring tools for the last decade, which are one example of tools for transforming existing software. This position paper is about thinking of software development as a series of transformations of working software. What transformations occur in practice? How can we carry them out more effectively?

## 2. A PERSONAL HISTORY OF REFACTORING

In the late 80s, I was trying to learn how to develop object-oriented frameworks. I developed some on my own and talked to people in industry who had developed some. The people who developed frameworks told stories about rewriting the framework many times. They’d make an application, discover something wrong with the framework, fix the framework, and then update all their applications to use the new version of the framework. So, I paid attention to our own experience, and noticed that we had to rewrite the framework many times as well, and consequently rewrote the applications. I also noticed that the changes to the code were not random, but fell into a fairly small set of categories. The developers at Tektronix called this process “refactoring”, and so Bill Opdyke and I made a list of refactorings that were common in developing frameworks in C++[2]. This was the first time the word was used in a published paper, though of course there is a long history of program transformation, and most of the refactorings are similar to previous work.

It was obvious to me that these refactorings should be automated, because refactorings that could be explained to a student in a few minutes often took a few days to carry out. Refactoring was not intellectually difficult, but it required keeping track of details and was error prone. C++ seemed too complicated for us, and the C++ programmers that we talked to did not seem to appreciate refactoring. They said that you could not change an interface once you published it because customers might have programs that depended on the interface. In contrast, Smalltalk was a much simpler language, there was already a standard representation for programs that was easy for us to manipulate, and many people in the Smalltalk community not only practiced refactoring, but thought it was important and wanted tool support. They knew that interfaces would change and wanted help in dealing with that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*FoSER 2010*, November 7-8, 2010, Santa Fe, New Mexico, USA.

Copyright 2010 ACM 978-1-4503-0427-6/10/11...\$10.00.

change. So, we built a refactoring tool for Smalltalk (the Smalltalk Refactoring Browser[3] which got released in 1994 and continued to be polished for four or five years. It was originally developed for VisualWorks and then also for IBM's VisualAge (two brands of Smalltalk) and eventually got ported to nearly all the Smalltalk environments. John Brant and Don Roberts did most of the work, and they eventually had a contract with Cincom that made it be a core part of VisualWorks.

In spite of our success, few people outside the (very small) Smalltalk community seemed to care about refactoring. Things started to change with the coming of Kent Beck's "Extreme Programming", because refactoring was one of its 12 key practices[4]. XP is also a counterexample to my claim that most software development processes focus on making new software, because all XP iterations except the first start with working software, and one of the key rules is that the system must never be allowed to stop working. XP treats all software development like maintenance. But from the point of view of this story, the important thing about XP was that it spread the word about refactoring, and that it lead Martin Fowler to write a book "Refactoring"[5]. This led to refactoring becoming an accepted practice in the Java community, and to the creation of more than a dozen refactoring tools for Java.

In the past few years, refactoring has become a popular research topic, with papers at OOPSLA, ICSE, and ICSM. There have been papers (and tool support) on making use of type information, refactoring to generics, refactoring to aspects, using refactoring to support framework evolution, and detecting refactorings. But the scope of refactoring has not changed much. Refactorings are still considered small program transformations that move code around and that change its structure, but that don't change what it does. Because the original Smalltalk Refactoring Browser focused on object-oriented features, most of the other refactoring tools do, too. Thus, Eclipse doesn't have much support for changing package boundaries or for refactoring plugins. This is typical of the state of the art in refactoring tools; they tend to focus on small program features and ignore the gross structure of systems. They focus on coding and not architecture.

Since the Smalltalk Refactoring Browser, my refactoring projects have been CRefactory, Photran, and work with Danny Dig on applying refactoring to component evolution. CRefactory is a refactoring tool for C. Its major novelty is a program representation that integrates C with the C preprocessor and so perfectly handles conditional compilation and macros[6]. Photran is an Eclipse plugin that is a Fortran IDE. Our goal is to provide a wide variety of refactorings for Fortran. In addition to providing a useful tool for Fortran programmers, we have advanced the state of the art in how to implement refactoring tools.

Refactorings are only a tiny fraction of the changes that programmers make. However, they illustrate a general principle: by focusing on a narrow class of change, it is possible to make those changes much more effectively. First, understanding those changes makes them easier to perform manually. Second, it might be possible to automate some or all of them.

"Program transformation" is often assumed to mean something like a compiler. Like compilers, refactoring tools must understand the syntax and semantics of a programming language and are usually based on a program representation. They differ in that they are interactive, and rely heavily on the user.

Programmers think of them as tools for transforming programs, instead of as systems that transform programs on their own.

### 3. UNDERSTANDING PROGRAM TRANSFORMATIONS

In general, we do not understand the kind of changes programmers make to software. The usual categories such as "bug fix" and "new feature" are too broad. In the past few years, software engineering researchers have been studying specific classes of bugs, such as null pointer violations and copy-and-paste errors, and have been using this as a basis of tools for detecting and fixing bugs. This is important work that fits the theme of this paper well. However, not as much work has been done on adding features. Don Batory's work on features is one good example of the kind of work that should be done[7]. We need to better understand what we mean by "fix a bug" and "add a feature", which means to learn how to categorize them into more precise changes, learn when each kind of change is most appropriate, learn how to detect them when programmers make them, and (finally) learn when and how to automate them.

Studies of existing programming projects have motivated much of our work on refactoring. Bill Opdyke developed his initial catalog of refactorings by looking at the history of Choices, an operating system written in C++. Before Danny Dig started work on automating upgrades to a new version of a software component, he looked at five Java systems and decided that over 80% of the changes to a component that were not backward compatible were changes that could be treated as refactorings that needed to be propagated to the programs using the component. Later, he looked at how Java programmers parallelized programs by introducing classes from the java.concurrent library. There need to be more case studies, and studies of existing programming projects.

One test of a good theory of changes is whether it leads to tools that can categorize changes recorded in a version control system. Can bug fixes and new features be identified? If so, what kind of bug was fixed? How can we distinguish an optimization from a refactoring or a bug fix? No theory will be perfect, but this seems to be a reasonable measure for a theory of program changes.

### 4. AUTOMATING SPECIFIC KINDS OF CHANGES

Given a particular kind of change, can it be automated? We have only worked on automating a few kinds of program transformations, but this is an area that deserves a lot of attention. The following categories should be taken as examples, not as a complete list.

#### 4.1 Security Transformations

Munawar Hafiz and I have been looking at security from the point of view of patterns and program transformations. Many security techniques are program transformations. For example, eliminating buffer overflows or SQL injection attacks are program transformations, as are nearly every technique that targets a specific security vulnerability. More general software design techniques such as compartmentalization or "chroot jail" are also program transformations.

A lot of the recent research on security involves either detecting when a program transformation is needed, or in automating a

particular program transformation. But this is usually done as a solution to a specific problem, not as a part of general environment for program transformation. According to the principle of “defense in depth”, security will come from a set of different techniques, not from any single one. In fact, since new security threats are constantly appearing, we need a way of easily adding a new program transformation.

Experts on security, performance, reliability, and good user interfaces all say that their specialty is so crucial that it must be considered from the first days of a project. They say “You can’t retrofit security”, or performance, or a good user interface. But people do it all the time! Advocates for a particular software quality insist that it can’t be retrofitted because they want to make people think about it at the beginning of a project. The motive might be noble but the statement is false; since given enough effort, any change is possible. We can put a basement under a two-story house and can add air-conditioning and Ethernet to a 500 year old castle. In the same way, we can refactor a program to make it more secure or more reliable. Sometimes it is cheaper to knock a house down and build a new one, and sometimes it is cheaper to start a new program. But as we make program transformation cheaper, it becomes more cost-effective to improve our old software instead of making something new.

## 4.2 Refactoring for performance and portability

Photran is a refactoring tool for Fortran that we are building. Some of the refactorings are for the classic purposes of refactoring, such as making software easier to understand and making it more reusable. However, Fortran programmers tend to be concerned with performance, and a lot of the changes to Fortran programs are for the purpose of making it faster on a particular platform. Ideally, an optimizing compiler will be enough, but usually Fortran programmers spend a lot of time hand-optimizing.

When programmers port a program from one supercomputer to another, they often have to undo some of the old hand-optimizations before making new ones. Part of the problem is that they don’t distinguish between the portable, unoptimized version, and a version optimized for a particular machine. One way to approach this problem is as a configuration management problem. But if optimizing a Fortran program can be thought of as refactoring it, then making the program portable can be thought of detecting the refactorings and undoing them.

Making more conventional programs portable is an easier refactoring task. Usually most of the system is machine independent, and only part is not portable. So, the system is refactored into a machine dependent and a machine independent part. The machine dependent part becomes the portability layer. The system can be ported to a new machine by rewriting the portability layer.

Similarly, refactoring can be used to move from one library to another. First, all calls on the first library are segregated to a single module. Then that module can be rewritten to use the new library. As with a portability layer, this new design might be slower and harder to understand than the original one, though often it isn’t. If it is, and if only the new library is going to be used, then the calls to the new module can be in-lined to call the new library directly and the module can be deleted.

## 5. INFRASTRUCTURE NEEDS

There are many tools aimed at helping build program transformation tools. For example, Stratego/XT is a transformation language that has been used to build IDEs[8]. Each tool focuses on a few aspects of the entire problem. We’ve focused on the fast reponse time needed by an interactive tool, and ensuring that refactorings do not destroy comments or preprocessor directives, but wrote the transformations in Java. In contrast, Stratego/XT is a declarative language for specifying transformations, but did not pay as much attention to white space. As time goes on, these tools will tend to borrow ideas from each other. But there are some problems that none of the tools have addressed, in particular dealing with programs written in a variety of languages, dealing with a system that is a collection of programs, and studying the effect on other programming tools of thinking of a program as a sequence of program transformations.

### 5.1 Multilingual Transformations

Large systems tend to use more than one language. These include not only conventional programming languages, but specialized languages for defining web pages, database schemas, menu layouts, and so on. Systems like struts and Hibernate embed class names and method names in XML files, making these XML files essentially an extension of Java. Changing a file in one language often requires changing a file in another language. Thus, refactoring tools need to be able to handle changes across language boundaries.

One way of mixing languages is with preprocessors. This includes classic preprocessors like the C preprocessor as well as code generators like yacc or antlr. Some preprocessors are easier to analyze than others. In the worst case, such as for the C preprocessor,

Currently we are trying to write C preprocessor support for Photran, reusing the techniques invented for CRefactory. CRefactory can refactor programs written in a mix of C and the C preprocessor. Although the techniques could be used for other language combinations, each combination will require a separate tool. This is important for Photran because some Fortran programs use the C preprocessor, some use other preprocessors, and others use none. I’ve been working on a large Fortran project called IBEAM that has a custom-made preprocessor written in Python. We want to make it easy to extend Photran to handle different preprocessors, but we haven’t succeeded yet.

Modern software is usually written in a combination of languages. Therefore, if refactoring is universal and if we are going to support it with tools, we need to have multilingual refactoring tools. Further, it must be easy to extend these tools to handle new languages.

### 5.2 Transformations Across Programs

Many program transformations are larger than a single program. For example, qmail is a mail transport agent that was designed to be more secure than sendmail, its chief competitor, and has had a perfect security record since it was released in 1997. One of the reasons that it is so secure is that it is composed of over twenty small C programs, while sendmail is one large C program[8]. The overall behavior of qmail is similar to sendmail, though each has features that the other doesn’t, and qmail was not created by refactoring sendmail, but is a total rewrite. However, sendmail was refactored into several processes to achieve some of the

security benefits of qmail. In particular, the smtpd server was separated from the rest of the program because, as the public interface to the mail transport agent, it is the part of the system most likely to be attacked. Separating the smtpd server into a separate process means that attackers that overcome its defenses will still not gain access to the important features of the mail transport agent. Separating the smtpd server from the rest of the program is a complex refactoring. Not only does it require making two processes where there used to be one, but it requires that all communication between those processes be done by pipes or by shared files, instead of by direct procedure calls and pointers to shared memory.

### 5.3 Integration with version control

As part of Danny Dig's dissertation on using refactoring tools to ease the cost of component evolution, he developed a version control system that represented program history as a sequence of refactorings and program edits. [10] This turned out to make merging much easier. It is probably possible to integrate other kind of program transformations with version control. This could make it easier to understand the history of the system. Making merging easier means that it is easier to apply a change done to one branch to another branch. Changes become components that can be moved from one branch of the version history to another.

Programs are often viewed as a set of modules. An alternate view is to think of a program as a sequence of program transformations. Composing sequences to form large sequences becomes a new kind of program composition, one that is well-suited for feature-oriented programming [7].

## 6. ADVANTAGES OF ORGANIZING PROJECTS AROUND PROGRAM TRANSFORMATIONS

Transformations make the development process more repeatable. Transformations have explicit steps to be performed that help in estimating the time it takes to manually perform the transformation. The steps also help to limit defects. Without the explicit steps, different developers will have different processes to make the changes. Some of these processes will be faulty and others will be non-optimal.

Once the steps for a transformation have been defined, it may be possible to automate them. Automated transformations will be faster and more reliable than manual ones. Automated transformations are also easier to merge, so changes can more easily be done in parallel. Even when changes cannot be merged, programmers using automated transformations can work on their own to develop a "transformation script" that can then be replayed on the latest version.

## 7. CONCLUSION

Maintenance has traditionally been considered a less-interesting phase of the software lifecycle. If it is instead considered normal software development, software development becomes a kind of program transformation. Therefore, we should be asking questions such as:

- 1) What kinds of program transformations occur in practice?

- 2) Can they be automated? If so, how?
- 3) How can we design programming environments based on program transformations?

If we face up to the fact that software development is program transformation, we are sure to improve the way we develop software.

## 8. ACKNOWLEDGEMENTS

Thanks for comments on earlier drafts of this paper by Paul Adamczyk, John Brant, Nick Chen, Tanya Crenshaw, Danny Dig, Munawar Hafiz, Joshua Kerievsky, Stas Negara, Jeff Overbey, and Joe Yoder.

## 9. REFERENCES

- [1] Petroski, H. 1996. *Invention by Design: How Engineers Get from Thought to Thing*. Harvard University Press.
- [2] Opdyke, W.F. and Johnson, R.E. 1990. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented System., *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Application*, ACM
- [3] Roberts, D. , Brant, J., and Johnson, R.E. 1997. A Refactoring Tool for Smalltalk, *Theory and Practice of Object Systems*, vol. 3 no. 4, October 1997
- [4] Beck, Kent 1999. *eXtreme Programming Explained: Embrace Change*. Addison-Wesley
- [5] Fowler, M., Beck K, Brant, J., Opdyke, W, and Roberts, D. 1999. *Refactoring: Improving the Design of Existing Code*, Addison Wesley.
- [6] Garrido, A. and Johnson, R.E. 2005. Analyzing Multiple Configurations of a C Program, *Proceedings of the 21<sup>st</sup> International Conference on Software Maintenance*, September 2005
- [7] Delaware, B., Cook, W., and Batory, D. 2008. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition . ACM SIGSOFT FSE, August 2009.
- [8] Bravenboer, M., Kalleberg, K., Vermaas, R. and Visser, E. 2001. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. of Comp. Programming*, 72(1-2):52-70, June 2008.
- [9] Hafiz, M. and Johnson. R.E. 2008. Evolution of the MTA Architecture: An Impact of Security. *Software---Practice and Experience*, 38(15):1569-1599, Dec 2008.
- [10] Hafiz, H, Adamczyk, P and Johnson, RE. 2009. Systematically Eradicating Data Injection Attacks using Security-oriented Program Transformations. *In ESSoS09: Symposium on Engineering Secure Software and Systems*. Leuven, Belgium. Feb, 2009
- [11] Dig, D., Manzoor, K., Johnson, R.E. and Nguyen, T. 2008. : Effective Software Merging in the Presence of Object-Oriented Refactorings, *In IEEE Transactions on Software Engineering (TSE)*, Volume 34, Number 3, pp 321-335, May/June 2008