Chapter 13: Refactoring, Reuse & Reality

by Bill Opdyke

Lucent Technologies/ Bell Labs

(wopdyke@lucent.com; william.opdyke@bell-labs.com)

Introduction

Martin Fowler and I first met in Vancouver during OOPSLA '92. A few months earlier, I had completed my doctoral thesis [1] at the University of Illinois on refactoring object-oriented frameworks. While I was considering doing follow-on research into refactoring, I was also exploring other options such as medical informatics. Martin was working in a medical informatics application at the time, which is what brought us together to chat over breakfast in Vancouver. As Martin related earlier in this book, we spent a few minutes discussing my refactoring research. He had limited interest in the topic at the time but, as you are now aware, his interest in the topic has grown.

As Martin has noted earlier, this book isn't the first written work on refactoring, but (I hope) it will expose a broadening audience to the concepts and benefits of refactoring. While my doctoral thesis was the first major written work on the topic, most readers interested in exploring the early foundational work on refactoring probably should look first at several papers [2, 3, 4, 5] and conference tutorials [6, 7]. For those with an interest in both design patterns and refactoring, the paper “Lifecycle and Refactoring Patterns That Support Evolution and Reuse” [5], which Brian Foote and I presented at PLoP '94 and which appears in the first volume of Addison-Wesley's “Pattern Languages

of Program Design" series, is a good place to start. My refactoring research was largely built upon work by Brian and Ralph Johnson regarding object-oriented application frameworks and designing reusable classes [8]. Subsequent refactoring research by John Brant, Don Roberts and Ralph Johnson at the University of Illinois has focused on refactoring Smalltalk programs [9, 18]. Their web site (http://st-www.cs.uiuc.edu) includes some of their most recent work. Interest in refactoring has grown within the object-oriented research community; several related papers were presented at the OOPSLA '96 conference in a session entitled "Refactoring and Reuse" [15].

When Martin offered me the opportunity to author a chapter in this book, several ideas came to mind. I could describe the early refactoring research, where Ralph Johnson and I came together from very different technical backgrounds to focus on support for change in object oriented software. I could discuss how to provide automated support for refactoring, an area of my research quite different from the focus of this book. I could share some of the lessons I have learned about how refactoring relates to the day-to-day concerns of software professionals, especially those who work on large projects in industry. Many of the insights I gained during my refactoring research have been useful in a wide range of areas - in assessing software technologies and formulating product evolution strategies, in developing prototypes and products in the telecommunications industry, and in training and consulting with product development groups.

I decided to focus briefly on many of these issues. As the title of this chapter implies, many of the insights regarding refactoring apply more generally to issues like software reuse, product evolution and platform selection. While parts of this chapter will briefly touch upon some of the more interesting theoretical aspects of refactoring, the primary focus is on practical, "real world" concerns and how they can be addressed.

A Reality Check

I worked at Bell Labs for several years before I decided to pursue my doctoral studies. Most of that time was spent working in a part of the company that developed electronic switching systems. Such products

have very tight constraints both with respect to reliability and the speed with which they handle phone calls. Thousands of staff-years have been invested in developing and evolving such systems. Product lifetimes have spanned decades. Most of the cost of developing these systems comes not in developing the initial release, but in changing and adapting them over time. If ways could be found to make such changes easier and less costly, it could be a big win for the company.

Since Bell Labs was funding my doctoral studies, I wanted a research area that was not only technically interesting but also related to a practical business need. In the late 1980s, object-oriented technology was just beginning to emerge from the research labs. When Ralph Johnson proposed a research topic that focused both on object-oriented technology and on supporting the process of change and software evolution, I grabbed it!

I've been told that when people finish their PhDs, they are rarely neutral about their thesis topic. Some are sick of the topic, and quickly move on to something else. Others remain enthusiastic about their topic. I was in the latter camp.

When I returned to Bell Labs after finishing my degree, a strange thing happened. The people around me were not nearly as excited about refactoring as I was.

I can vividly recall presenting a talk in early 1993 at a technology exchange forum for staff at AT&T Bell Labs and NCR (we were all part of the same company at the time). I was given 45 minutes to speak on refactoring. At first, the talk seemed to go well. My enthusiasm for the topic came across. But, at the end of the talk, there were very few questions. One of the attendees came up afterward to learn more - he was beginning his graduate work and was fishing around for a research topic. But, I had hoped to see some members of development projects show eagerness in applying refactoring to their jobs. If they were eager, they didn't express it at the time.

People just didn't seem to “get it”.

Ralph Johnson taught me an important lesson about doing research: if someone (a reviewer of a paper, or attendee at a talk) comments “I don't understand” or just doesn't “get it”, it's *our* fault not theirs. It is

our responsibility to work hard to more clearly develop and communicate our ideas.

Over the next couple years, I had numerous opportunities to talk about refactoring at AT&T/Bell Labs internal forums and at outside conferences and workshops. As I talked more with developers "in the trenches", I started to understand why my earlier messages didn't come across clearly. The disconnect was partly caused by the newness of object-oriented technology; those who had worked with it had rarely progressed beyond the initial release and hence had not yet faced the tough evolution problems where refactoring can help. This was the typical "researcher's dilemma" - the state of the art was beyond the state of common practice. However, there was another, troubling cause for the disconnect: there were several "common sense" reasons why developers, even if they bought into the benefits of refactoring, were reluctant to refactor *their* programs. These concerns needed to be addressed before refactoring could be embraced by the development community.

Why are Developers Reluctant to Refactor *Their* Programs?

Suppose you are a software developer. If your project is a "fresh start" (with no backward compatibility concerns) AND if you understand the problem your system is intended to solve AND if your funder is willing to pay until *you* are satisfied with the results - consider yourself very fortunate! While such a scenario may be ideal for applying object-oriented techniques, for most of us such a scenario is only a dream.

More often, you are asked to extend an existing piece of software. You have a less-than-complete understanding of what you are doing. You are under schedule pressure to produce. What can you do?

- You could re-write the program. You could leverage your design experience, and correct the ills of the past - and be creative and have fun! However, who will foot the bill? How can you be sure that the new system does everything the old system used to do?
- You could copy and modify parts of the existing system to extend its capabilities. This may seem expedient, and may even be viewed

as a way to demonstrate reuse - without understanding what you are reusing! However, over time, errors propagate, programs get bloated, program design gets corrupted, and the incremental cost of change escalates.

Refactoring is a middle ground between these two extremes. It is a way to restructure software to make design insights more explicit, to develop frameworks and extract reusable components, to clarify the software architecture, and prepare to make additions easier. Refactoring can help you leverage your past investment, reduce duplication and streamline a program.

Suppose you (as a developer) buy into these advantages. You agree with Fred Brooks that dealing with change is one of the "essential complexities" of developing software [10]. You agree that, in the abstract, refactoring could provide the stated advantages.

Why might you still not refactor *your* programs? Here are four possible reasons:

- You might not understand how to refactor.
- If the benefits are long-term, why exert the effort now? In the long term, you might not be with the project to reap the benefits!
- Refactoring code is an overhead activity; you're paid to write *new* features.
- Refactoring might break the existing program.

These are all valid concerns. I have heard them expressed by staff at telecommunications and (more generally) at high technology companies. Some of these are technical concerns; others are management concerns. All must be addressed before developers will consider refactoring their software.

Let's deal with each of these issues in turn.

Understanding How & Where to Refactor

How can you learn how to refactor? What are the tools and techniques, how can they be combined to accomplish something useful, and when should we apply them?

This book defines several dozen refactorings that Martin found useful in his work, with examples of how they can be applied to support significant changes to programs.

In the Software Refactory project at the University of Illinois, we chose a more minimalist approach. We defined a smaller set of refactorings (in [1] and [5]), and showed how they could be applied. We based our collection of refactorings on own programming experiences, by evaluating the structural evolution of several object-oriented frameworks (mostly in C++), and by talking with and reading the retrospectives of several experienced Smalltalk developers. Most of our refactorings are low-level, e.g., creating or deleting a class, variable or function; changing attributes of variables and functions such as their access permissions (e.g. public or protected) and function arguments, or moving variables and functions between classes. A smaller set of high-level refactorings perform operations like creating an abstract superclass, simplifying a class by subclassing and simplifying conditionals, or splitting off part of an existing class to create a new, reusable component class (often converting between inheritance and delegation/aggregation). The more complex refactorings are defined in terms of the low-level refactorings. Our approach was motivated by concern for automated support and safely, which I will discuss later.

Given an existing program, what refactorings should you apply? That depends, of course, on your goals. One common reason, which is the focus of this book, is to restructure a program to make it easier to add (near-term) a new feature. We'll talk more about this in the next section. There are, however, other reasons why you might apply refactorings.

Experienced object-oriented programmers, and those who have been trained in design patterns and (more generally) in good design techniques, have learned that there are several desirable structural qualities and characteristics of programs that have been shown to support extensibility and reuse. Foote and Johnson have written on this [8], as have Rochat [11], Lieberherr and Holland [12] and others. Object-oriented design techniques such as CRC [14] focus on defining classes and their protocols; while their focus is upon up-front design, there are ways to evaluate existing programs against such guidelines.

An automated tool can identify structural weaknesses in a program, such as functions that have an excessively large number of arguments or are excessively long. These are candidates for refactoring. Similarly, an automated tool can identify structural similarities that may indicate redundancies. For example, if two functions are nearly identical (that often happens when a copy-and-modify process was applied to the first function to produce the second), such similarities can be detected and refactorings suggested that can move common code to one place. If two variables in different parts of a program have the same name, they can sometimes be replaced by a single variable that is inherited in both places. These a few very simple examples - there are many other, more complex cases that can be detected (and corrected) by an automated tool. These structural abnormalities or structural similarities don't always mean that you'd want to apply a refactoring, but often they do.

More recently, much of the work on design patterns has focused on good programming style, and on useful patterns of interactions among parts of a program that can mapped be into structural characteristics and to refactoring. For example, the applicability section of the Template Method pattern [13] makes reference to our abstract superclass refactoring described in [3].

In [1] I list some of the heuristics that can identify candidates for refactoring in a C++ program. More recently, John Brant and Don Roberts have created a tool that applies an extensive set of heuristics to automatically analyze Smalltalk programs and suggest what refactorings might improve the program design and where to apply them [9, 18].

Applying such a tool to analyze your program is somewhat analogous to applying “lint” to a C or C++ program. The tool isn't smart enough to understand the meaning of the program. Only some of the suggestions it makes, based on structural program analysis, may be changes you really want to make. As a programmer, you make the call. You decide which recommendations to actually apply to your program. Those changes should improve the structure of your program and, in general, better support changes down the road.

In summary, before programmers can convince themselves that they ought to refactor their code, they need to understand how and where to refactor. There is no substitute for experience. We leveraged the

insights of experienced object-oriented developers in our research, resulting in a set of useful refactorings and insights as to where they ought to be applied. Automated tools can analyze the structure of a program and suggest refactorings that might improve that structure. As with most disciplines, tools and techniques can help, *but only if you use them*. As programmers refactor their code, their understanding grows.

Refactoring to Achieve Near-Term Benefits

It is relatively easy to describe the mid-to-long range benefits of refactoring. However, many organizations are increasingly judged (by the investment community and by others) on their near-term performance. Can refactoring make a difference in the near term?

Refactoring has been successfully applied for over ten years by experienced object-oriented developers. Many of these programmers "cut their teeth" in a Smalltalk culture that valued clarity and simplicity of code, and embraced reuse. In such a culture, programmers would invest time to refactor because it was "the right thing to do". The Smalltalk language and its implementations made refactoring possible in ways that hadn't been true for most prior languages and software development environments. Much of the early Smalltalk programming was done in research groups such as Xerox PARC, or in small programming teams at leading-edge companies and consulting firms. The values of these groups were somewhat different from the values of many industrial software groups.

Martin Fowler and I are both aware that, for refactoring to be embraced by the more mainstream software development community, at least some of its benefits must be near-term. Martin, in the extended examples provided in this book, describes how refactoring is applied to support near-term extensions to a program.

In [2, 3, 4, 5, 6, 7] our research team describes several examples of how refactorings can be interleaved with extensions to a program, in a way that achieves both near-term and longer-term benefits. Those examples are somewhat less powerful than Martin's, but for many readers they may be easier to follow.

One of our examples is the Choices file system framework. Initially, the framework implemented the BSD UNIX file system format. Later, it was extended to support System V UNIX, MS/DOS, persistent and distributed file systems. System V file systems bear many similarities to BSD UNIX file systems. The approach taken by the framework developer was first to clone parts of BSD UNIX implementation, then modify the clone to support System V. The resultant implementation worked, but there was lots of duplicate code hanging around. After adding the new code, the framework developer refactored the code, creating abstract superclasses to contain the behavior common to the two UNIX file system implementations. Common variables and functions were moved to superclasses. In cases where corresponding functions were nearly but not entirely identical for the two file system implementations, new functions were defined in each subclass to contain the differences, and in the original functions those code segments were replaced with calls to the new functions. Code was incrementally made more similar in the two subclasses. When the functions were identical, they were moved to a common superclass.

These refactorings provide several near-term and mid-term benefits. Near-term, during testing, errors found in the common code base needed to only be modified *in one place*. The overall code size was smaller. The behavior specific to a particular file system format was cleanly separated from the code common to the two file system formats, making it easier to track down and fix behaviors specific to that file system format. Mid-term, the abstractions that resulted from refactoring were often useful in defining subsequent file systems. Granted, the behavior common to the two existing file system formats might not be entirely common for a third format, but the existing base of common code was a valuable starting point, and subsequent refactorings could be applied to clarify what was really common. The framework development team found that over time, it took less effort to incrementally add support for a new file system format, even though the newer formats were more complex developments was done by less experienced staff.

I could site other examples where near-term and longer-term benefits are realized using refactoring, but Martin has already done this. Rather than add to his list, let me draw an analogy to something that is near and dear to many of us: our physical health.

In many ways, refactoring is like exercise and eating a proper diet. Many of us know that we ought to exercise more and eat a more balanced diet. Some of us live in cultures that highly encourage this. Some of us can get by, for a while, without doing this, perhaps even without visible effects. We can always make excuses for not doing this. But, we are only fooling ourselves if we continue to ignore it.

Some of us are motivated by near term benefits of exercise and eating a proper diet, such as high(er) energy levels, greater flexibility, higher self-esteem and other benefits. Nearly all of us know that these near term benefits are *very* real. Many - but not all - of us make at least sporadic efforts in these areas. Others, however, aren't sufficiently motivated to "do something" until they reach a crisis point.

Yes, there are cautions that need to be applied - people should consult with an expert before embarking on a program. In the case of exercise and dieting, they should consult with their physician. In the case of refactoring, they should seek out resources such as this book and the papers cited above; staff experienced in refactoring should be able to provide more focused assistance.

Several people I've met are role models with respect to fitness and/or refactoring. I admire their energy and their productivity. Other, negative "role models" show the visible signs of neglect. Their future - and the future of the software systems they produce - may not be nearly so rosy.

In summary, refactoring can achieve near term benefits, as well as making the software easier to modify and maintain down the road. Refactoring is a means rather than an end. It is part of a broader context of how programmers or programming teams develop and maintain their software [5].

Reducing the Overhead of Refactoring

"Refactoring is an overhead activity - I'm paid to write new, revenue generating features."

My response, in summary is this:

- Tools/technologies are now available to allow refactoring to be done quickly and relatively painlessly.

- Experiences reported by some object-oriented programmers suggest that the overhead of refactoring is more than compensated by reduced efforts and intervals in other phases of program development.
- While refactoring may seem a bit awkward and an overhead at first, as it becomes part of a software development regimen, it stops feeling like overhead and starts feeling like an essential.

As noted above, perhaps the most mature tool for doing automated refactoring has been developed (for Smalltalk) by the Software Refactory team at the University of Illinois, and is freely available at their web site. While refactoring tools for other languages are not so readily available, many of the techniques described in our papers and in this book can be applied in relatively straightforward manner using a text editor (and, better yet, a browser). Software development environments and browsers have progressed substantially in recent years. We hope to see a growing set of refactoring tools available in the future.

Kent Beck and Ward Cunningham, both experienced Smalltalk programmers, have reported at OOPSLA conferences and other forums that refactoring has enabled them to develop software rapidly in domains such as bond trading. I have also heard similar testimonials from C++ and CLOS developers. In this book, Martin describes the benefits of refactoring with respect to Java programs. We expect to hear more testimonials from those who read this book and apply these principles.

Our experience suggests that, as refactoring becomes part of your routine, it stops feeling like overhead. Admittedly, that is easy to state but hard to substantiate. To the skeptics among you, my advice is: just do it, then decide for yourself. Give it time, though.

Refactoring Safely

Safety is a concern - especially for organizations developing and evolving large systems. In many applications, there are compelling financial, legal and ethical considerations for providing continuous, reliable and error-free service. Many organizations provide extensive

training and attempt to apply disciplined development processes in order to help ensure the safety of their products.

For many programmers, though, safety often seems to be less of a concern. It's more than a little ironic that many of us preach “safety first” to our children, nieces and nephews, while in our roles as programmers we scream for freedom - a hybrid of the wild west gunslinger and teenage driver. Give us freedom, give us the resources, and watch us fly. After all, do we really want our organization to miss out on the fruits of our creativity, merely for the sake of repeatability and conformity?

In this section, we discuss approaches for refactoring safely. I'll focus on an approach that, compared with what Martin Fowler has described earlier in this book, is somewhat more structured / rigorous but which can eliminate many of errors that might be introduced in refactoring.

Safety is a difficult concept to pin down. Intuitively, a safe refactoring is one that doesn't break a program. Since a refactoring is intended to restructure a program without changing its behavior, a program should perform the same after a refactoring as before.

How does one safely refactor? There are several options:

- Trust your coding abilities.
- Trust that your compiler will catch errors that you miss.
- Trust that your test suite will catch errors that you and your compiler missed.
- Trust that code review(s) will catch errors that you, your compiler and your test suite missed.

Martin focuses on the first three options in his refactoring. Mid-to-large-size organizations often supplement these with code reviews.

While compilers, test suites, code reviews and disciplined coding styles are all valuable, there are limits to all of these approaches:

- Programmers are fallible - even you. (I know I am!)
- There are subtle (and some not-so-subtle) errors that compilers can't catch - especially scoping errors related to inheritance [1].

- Perry and Kaiser [16] and others have shown that, while it is (or at least, used to be) common wisdom that the testing task is made simpler when inheritance is used as an implementation technique, in reality an extensive set of tests is often needed to cover all the cases where operations that used to be requested on an instance of class are now requested on instances of its subclass(es). Unless your test designer is omniscient (or pays great attention to detail) there are likely to be cases your test suite won't cover. Testing all possible execution paths in a program is in general a computationally undecidable problem. In other words, in general you can't be guaranteed to have caught all of the cases with your test suite!
- Code reviewers, like programmers, are fallible. Furthermore, reviewers may be too busy with their main job to thoroughly review someone else's code.

Another approach, which I took in my research, was to define and prototype a refactoring tool to check if a refactoring can be safely applied to a program and, if it is, refactor the program. This avoids many of the bugs that may be introduced by “human error”.

Part of my refactoring tool was a “program analyzer” which is a program that analyzes the structure of a another program (in this case, a C++ program to which a refactoring might be applied). That tool could “answer” a series of questions regarding scoping, typing and program semantics (i.e. the “meaning”/ intended operations of a program). Scoping issues related to inheritance make this analysis more complex than with many non-OO programs, but for C++, language features such as static typing makes the analysis easier than for, say, Smalltalk.

Consider, for example, the refactoring to delete a variable from a program. A tool can determine what other parts of a program (if any) reference the variable. If there are any references, then removing the variable would leave dangling references - thus this refactoring would not generally be safe. When a user asks the tool to refactor their program, it would flag this as an error. The user might then decide that the refactoring was a bad idea after all, or the user may decide to change the parts of the program that refer to that variable, then apply the refactoring to remove the variable. There are many other checks, most as simple as this, some more complex.

In my research, I defined safety in terms of program properties (related to scoping, typing, etc.) that need to continue to hold after applying a refactoring. Many of these program properties are similar to integrity constraints that must be maintained when database schemas change [17]. Each refactoring has associated with it a set of necessary preconditions which (if true) would ensure that the program properties are preserved. Only if the tool could determine that everything is "safe" would the tool perform the refactoring.

Fortunately, determining whether a refactoring is safe is often trivial, especially for the low-level refactorings which comprise most of our refactorings. To ensure that the higher-level, more complicated were safe, we defined them in terms of the low-level refactorings. For example, the refactoring to create an abstract superclass is defined in terms of "steps" which are simpler refactorings such as creating and moving variables and methods. By showing that each step of a more complicated refactoring is safe, we can know that (by construction) that that refactoring is safe.

There are some (relatively rare) cases where a refactoring that might actually be safe to apply to a program, but where a tool can't be sure - in which case the tool takes the safe route and disallows the refactoring.

For instance, consider again the case where you want to remove a variable from a program, but there is a reference to it from somewhere else in the program. Perhaps that reference is contained in a code segment that will never actually be executed - ever. For example, the reference many appear inside a conditional (e.g. if/then loop) which will never test true. If you could be sure that the conditional would never test true - ever - the you could remove the conditional test, including the code referring to the variable/ function that you want to delete, and then you could safely remove the variable/ function. In general it isn't possible to know for certain whether that condition will always be false. (Suppose you inherited code that was developed by someone else - how confident would you be in deleting this code?)

In this case, a refactoring tool could flag the reference and alert the user. The user might decide to leave the code alone. Or, if/ when they were sure that the referencing code would never be executed, they could remove that code and then apply the refactoring. The tool makes

the user aware of the implications of what of the reference, rather than blindly applying the change.

Wheh! This may sound like complicated stuff - OK for a doctoral thesis (whose primary audience, the thesis committee, want to see some attention to theoretical issues) but is it practical for “real” refactoring?

All of this safety checking can be implemented “under the hood” of a refactoring tool. The programmer who wants to refactor their program merely needs to ask the tool to check the code and, if safe, perform the refactoring. While my tool was a research prototype, John Brant and Don Roberts have implemented a far more robust and featured tool as part of their (follow on) research into refactoring Smalltalk programs [9].

In summary, there are many levels of safety that can be applied to refactoring. Some are easy to apply but don't guarantee a high level of safety. Using a refactoring tool can provide many benefits - there are many simple but tedious checks that it can make, flagging in advance problems that if left unchecked would cause the program to break as a result of refactoring.

While applying a tool such as this avoids introducing many of the errors that you otherwise hope will be flagged during compilation, testing and code review, these other techniques are still of value, particularly when developing or evolving real-time systems. Often, programs don't execute in isolation - they are parts of a larger system. Some refactorings not only clean up the code but can make a program run more quickly. Speeding up one program might result in performance bottlenecks elsewhere. This is similar to the effects of upgrading microprocessors that speed up parts of a system, and require similar approaches to tune and test overall system performance. Conversely, some refactorings may slow down the overall performance a bit, but in general such performance impacts are minimal.

These approaches are intended to guarantee that refactoring does not introduce *new* errors into a program. These approaches don't detect and fix bugs that were in the program before it was refactored. However, refactoring may make it easier to spot such bugs and subsequently correct them.

A Reality Check (Revisited)

- Making refactoring real requires addressing the "real world" concerns of software professionals. Four commonly expressed concerns are:
- They might not understand how to refactor.
- If the benefits are long-term, why exert the effort now? In the long term, you might not be with the project to reap the benefits!
- Refactoring code is an overhead activity; they are paid to write *new* features.
- Refactoring might break the existing program.

In this chapter, I have briefly addressed each of these concerns, providing pointers for those who want to delve further into these topics.

There are other issues that are of concern to some projects:

- What if the code to be refactored is collectively "owned" by several people? In some cases, many of the traditional change management mechanisms are relevant. In other cases, if the software has been well designed and refactored, sub-systems will be sufficiently de-coupled that many refactorings will only affect a small subset of the code base.
- What if there are multiple versions/ code lines from a code base? In some cases, refactorings may be relevant for all of the versions, in which case all need to be checked for safety before applying the refactoring. In other cases, the refactorings may only be relevant for some versions, which simplifies the process of checking and refactoring the code. Managing changes to multiple versions often requires applying many of the traditional version management techniques. Refactoring can be useful in merging variants/ versions into an updated code base, which may simplify version management downstream.

For other discussions regarding the value and practical utility of refactoring, please see [6, 7].

In summary, convincing software professionals of the practical value of refactoring is quite different from convincing a doctoral committee that refactoring research is worthy of a Ph.D. It took me some time,

after completing my graduate studies, to fully appreciate these differences.

Hopefully, by this point in the book, you are planning to apply refactoring techniques in you work and/or will be encouraging others in your organization to do so. If you are still undecided, you may want to refer to the references I have provided or possibly contact Martin, myself or others who are experienced in refactoring.

Implications Regarding Software Reuse

The “real world” concerns addressed above don't only apply to refactoring - they apply more broadly to software evolution and reuse.

For much of the past several years, I have focused on issues related to software reuse, platforms, frameworks, patterns and the evolution of legacy systems - often involving software that was not “object oriented”. In addition to working with projects within Lucent/ Bell Labs, I have also participated in forums with staff at other organizations who have been grappling with similar issues [19, 20, 21, 22].

The “real world” concerns regarding a reuse program are very similar to those related to refactoring:

- Technical staff may not understand what to reuse and how to reuse it.
- Technical staff may not be motivated to apply a reuse approach unless short term benefits can be achieved.
- Overhead, learning curve and discovery cost issues must be addressed for a reuse approach to be successfully adopted.
- Adopting a reuse approach should not be disruptive to a project; there may be strong pressures to leverage existing “assets”/ implementation albeit with legacy constraints. New implementations should interwork/ be backward compatible with existing systems.

In [23] Geoffrey Moore describes the technology adoption process in terms of a bell-shaped curve where the front tail includes “innovators” and “early adopters”, the large middle hump includes “early majority” and “late majority”, and the trailing tail including “laggards”. For

an idea and product to succeed, it must ultimately be adopted by the early and late majorities. Put another way, many ideas that appeal to the innovators and early adopters ultimately fail because they never make it across the chasm to the early and late majorities. The disconnect mainly lies in the differing motivators of these customer groups. Innovators and early adopters are attracted by new technologies, visions of paradigms shifts and breakthroughs, whereas the early and late majorities are primarily concerned with maturity, cost, support, and in seeing if the new idea or product has already been successfully applied by others with needs similar to theirs.

As I noted earlier, software development professionals are impressed / convinced in very different ways than software researchers. Software researchers are most often what Moore refers to as innovators, while software developers and especially software managers are often part of the early and late majorities. Recognizing these differences is important in reaching each of these groups. With software reuse, as with refactoring, it is important to reach software development professionals on their terms.

Within Bell Labs / Lucent I found that encouraging the application of reuse and platforms required reaching a variety of stake holders: formulating strategy with executives, organizing leadership team meetings among middle managers, consulting with development projects, and publicizing the benefits of these technologies to broad R&D audiences through seminars and publications. Throughout, it was important to train staff in the principles, address near term benefits, provide ways to reduce the overhead and address how these techniques could be introduced safely - insights that I had gained from my refactoring research.

A Final Note

Thanks for taking the time to read this chapter. I've tried to address many of the concerns that you might have about refactoring, and tried to show many of the “real world” concerns regarding refactoring apply more broadly to software evolution and reuse. I hope that you came away enthusiastic about applying these ideas in your work.

At first glance, it might appear that refactoring began in academic research labs. In reality, it began in the software development trenches, where object-oriented programmers (then using Smalltalk) encountered situations where techniques were needed to better support the process of framework development - or, more generally, to support the process of change. This spawned research which has matured to the point where we feel that it is “ready for prime time” - where a broader set of software professionals will experience the benefits of refactoring.

Best wishes as you move forward in your software development tasks!

References:

[1] William F. Opdyke, "Refactoring Object-Oriented Frameworks". PhD Thesis, University of Illinois at Urbana-Champaign. Also available as Technical Report UIUCDCS-R-92-1759, Department of Computer Science, University of Illinois at Urbana-Champaign.

[2] William F. Opdyke and Ralph E. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems". In "Proceedings of SOOPPA '90: Symposium on Object-Oriented Programming Emphasizing Practical Applications". September 1990.

[3] William F. Opdyke and Ralph E. Johnson, "Creating Abstract Superclasses by Refactoring". In "Proceedings of CSC '93: The ACM 1993 Computer Science Conference". February 1993.

[4] Ralph E. Johnson and William F. Opdyke, "Refactoring and Aggregation". In "Proceedings of ISOTAS '93: International Symposium on Object Technologies for Advanced Software". November 1993.

[5] Brian Foote and William F. Opdyke, "Lifecycle and Refactoring Patterns That Support Evolution and Reuse". Presented at PLoP 94; included in "Pattern Languages of Program Design" (J. Coplien and D. Schmidt, eds.), Addison-Wesley, 1995, pp 239-257.

[6] William Opdyke and Don Roberts, "Refactoring". Tutorial presented at OOPSLA '95: 10th Annual Conference on Object Oriented

Program Systems, Languages and Applications, Austin, Texas, October, 1995.

[7] William Opdyke and Don Roberts, "Refactoring Object-Oriented Software to Support Evolution and Reuse". Tutorial presented at OOPSLA '96: 11th Annual Conference on Object Oriented Program Systems, Languages and Applications, San Jose, California, October, 1996.

[8] Ralph E. Johnson and Brian Foote, "Designing Reusable Classes". In "Journal of Object-Oriented Programming 1:2", 1988, pp 22-35.

[9] Don Roberts, John Brant, Ralph Johnson and William Opdyke, "An Automated Refactoring Tool." In Proceedings of ICAST '96: 12th International Conference on Advanced Science and Technology". Chicago, Illinois. April, 1996.

[10] Fred Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering". Information Processing 1986 - Proceedings of the IFIP Tenth World Computing Conference (H.-L. Kugler, ed.), Elsevier.

[11] Roxanna Rochat, "In Search of Good Smalltalk Programming Style". Technical report CR-86-19, Tektronix, 1986.

[12] Karl J. Lieberherr and Ian M. Holland, "Assuring Good Style For Object-Oriented Programs". In IEEE Software, pp 38-48, September, 1989.

[13] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1985.

[14] Rebecca Wirfs-Brock, Brian Wilkerson and Luaren Wiener, "Design Object-Oriented Software". Prentice-Hall. 1990.

[15] Proceedings of OOPSLA '96: Conference on Object-Oriented Programming Systems, Languages and Applications, San Jose, California. October, 1996.

[16] Dewayne E. Perry and Gail E. Kaiser, "Adequate Testing and Object-Oriented Programming". In "Journal of Object-Oriented Programming", January-February, 1990.

[17] Jay Banerjee and Won Kim, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases". In "Proceedings of the ACM SIGMOD Conference", 1987.

[18] Don Roberts, John Brant and Ralph E. Johnson, "A Refactoring Tool For Smalltalk". TAPOS 3(4); pp. 39-42. 1997.

[19] Report on WISR '97: Eighth Annual Workshop on Software Reuse, Columbus, Ohio, March 1997. In the September, 1997 issue of ACM Software Engineering Notes.

[20] Kent Beck, Grady Booch, Jim Coplien, Ralph Johnson and Bill Opdyke, "Beyond the Hype: Do Patterns and Frameworks Reduce Discovery Costs?". Panel session at OOPSLA '97: 12th Annual Conference on Object Oriented Program Systems, Languages and Applications, Atlanta, Georgia, October, 1997. Position statements appear in the OOPSLA '97 proceedings.

[21] David Kane, William Opdyke and David Dikel, "Managing Change to Reusable Software". Presented at PLoP 97: 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, September, 1997.

[22] Maggie Davis, Martin L. Griss, Luke Hohmann, Ian Hopper, Rebecca Joos and William F. Opdyke, "Software Reuse: Nemesis or Nirvana?". Panel session at OOPSLA '98: 13th Annual Conference on Object Oriented Program Systems, Languages and Applications, Vancouver, BC, Canada, October, 1998. Position statements to appear in the OOPSLA '98 proceedings.

[23] Geoffrey A. Moore, "Cross the Chasm: Marketing and Selling Technology Products to Mainstream Customers". HaperBusiness, 1991.