# Chapter 2: Principles in Refactoring

The preceding example should have given you a good feel for what refactoring is all about. Now it's time to step back and look at the key principles of refactoring, and look at some of the issues you need to think about in using refactoring.

## A Definition of Refactoring

I'm always a little leery of definitions, for everyone has their own, but when you write a book you get to choose your own definition. So here goes

> ***Refactoring*** *is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable behavior of that software component.*

You use the word refactoring both for large changes: "I need to spend some time refactoring the video billing software in order to make it support HTML bills". You also use it for the small steps, such as *Move Method (160)*. They are both refactorings, but a different scale.

I've been asked "is refactoring just cleaning up code". In a way the answer is yes, but I think refactoring goes further because it provides a technique for doing so in a more efficient and controlled manner. Since

I've been using refactoring, I've noticed that clean code far more effectively than I did before. This is because I know what refactorings to do, I know how to do them in a manner that minimizes bugs, and I test at every possible opportunity.

I should amplify a couple of points in my definition. Firstly the purpose of refactoring is all about making the software easier to understand and modify. There are many changes you could do to software that make little or no change to the observable behavior. But only those that are made in order to make the software easier to understand are refactorings. A good contrast is performance optimization. Like refactoring, performance optimization does not usually change the behavior of a component (other than its speed), it only alters the internal structure. However it's purpose is different. Usually performance optimizations make code harder to understand, but you need to do it to get the performance you need. (I'll talk more about performance optimization later.)

The second thing I want to highlight is that it does not change the observable behavior of the software component. What I mean by that is that the software still carries out the same function that it did before. Any user, whether an end user or another programmer, cannot tell that things have changed.

## Semantics-Preserving Refactorings

So good refactorings are *intention-clarifying* and *behavior-preserving* (at least to the outside). There is a particular category of refactorings that are especially interesting: those that are *semantics-preserving*. Semantics-preserving refactorings go beyond behavior-preserving in that not just is the observable behavior not changed, but you can *guarantee* that there is no change to the external behavior of the component. A simple example of a semantics-preserving change is the renaming of a method. If you rename a method, and you catch all the references to that method, then you can be absolutely certain that the software component works the same way. You can do this with more complex changes. It is possible to move a method from one class to another in a way that is semantics-preserving.

Semantics-preserving refactorings are important for a couple of reasons. Firstly they are a strong basis for manual refactorings. If you fol-

low a semantics-preserving refactoring, then you know that providing you don't make a mistake in following the refactoring, then you are not going to introduce a bug into the software. Therefore it is important to find semantics preserving refactorings and to use them as much as possible. There is still room for non-semantics-preserving refactorings, but they introduce a higher risk of error.

Semantics-preserving refactorings are particularly important for tool builders. If you can make the transformation in a tool, and prove that the transformation is semantics-preserving, then a user of the tool can use the refactoring freely and be confident that it won't introduce a bug.

## Why Should You Refactor?

I don't want to proclaim refactoring as the cure for all of software ills. It is no 'silver bullet'. Yet it is a valuable tool, a pair of silver pliers that helps you keep a good grip on your code. A tool that can, and should, be used for several purposes.

The fundamental reason is that **refactoring will help you develop software more quickly**. It does this because it fixes those things that otherwise slow you down. Working around a bad design slows you down. Failing to understand the code slows you down. In several projects where refactoring was used they reacted to reduced productivity by a burst of more intensive refactoring. They noticed that that always improved productivity.

The most obvious reason for refactoring is the one I've alluded to above: **refactoring improves the structure of existing code**. Without refactoring the program's design will decay. As people make changes to the code — changes done to realize short term goals, or changes made without a full comprehension of the design of the code — that code will lose its structure. It becomes harder to see the design by reading the code. Refactoring is rather like tidying up the code. Working to remove bits that aren't really in the right place. Loss of structure to code has a cumulative effect. The harder it is to see the design in the code, the harder it is to preserve it, and the more rapidly it decays. Regular refactoring helps it retain its shape.

Poorly designed code usually takes more code to do the same things, often because the code quite literally does the same thing in several places. **Refactoring eliminates duplicate code**. The importance of this lies in future modifications to the code. Reducing the amount of code won't make the system run any faster, its affect on the programs footprint is rarely significant, but it makes a big difference to the modification of the code. The more code there is, the harder it is to modify correctly. There's more code to understand. You change this bit of code here but the system doesn't do what you expect because you didn't change that bit over there that does much the same thing in a slightly different context. By eliminating the duplicates you ensure that the code says everything once, which is really the essence of good design. Refactoring reduces the size of your code, that's a good thing.

An old engineering adage I like is "the more there is, there more there is to go wrong". Duplicate code often leads to bugs, so refactoring remove bugs by removing unnecessary code. It goes deeper than this, however. When you refactor you are forced to look again at your own code, or concentrate on a piece of new code. The best refactoring is often done the first time you work with someone else's code. **Refactoring is a very effective way of spotting bugs**, especially when you are trying to improve its structure. Indeed there is a whole technique of software inspection, with many studies showing that inspection is more effective than testing at finding bugs. Refactoring is a very active and engaged form of inspection, and I've often found bugs when refactoring.

Programming is in many ways a conversation with a computer. You write code that tells the computer what to do, and it responds by doing exactly what you tell it. In time you close the gap between what you want it to do and what you tell it to do. Programming in this mode is all about saying exactly what you *want*. But there is another user of your source code. Someone who will try to read your code in a few months time to make some changes. We easily forget that extra user of the code, yet that user is actually the most important. Who cares if the computer takes a few more cycles to compile something? Yet it really matters if it takes a programmer a week to make a change that would only have taken an hour if she could have understood your code more easily.

The trouble is that when you are trying to get the program to work, you are not thinking about that future developer. It needs a change of rhythm to make those changes that make the code easier to understand. **Refactoring helps you to make your code more readable.** When refactoring you have code that works, but is not ideally structured. A little time spent refactoring can make the code better communicate its purpose. Programming in this mode is all about saying exactly what you *mean*.

Useful code never stays still. You always need to change it do something that was not expected when it was written. When looking at unfamiliar code you have to try to understand what it does. You look at a couple of lines and say to yourself, oh yes that's what this code is doing. With refactoring you don't stop at the mental note. You actually change the code to better reflect your understanding, and then test that understanding by re-running the code to see if it still works. In this case **refactoring helps you understand unfamiliar code**.

Now you have understood the code you need to do something new with it, something that was not envisaged by the original writer. If this is the case the code is not structured to help you, indeed you may have to do a work-round to get around this failing in the original code. So? Don't work around the original code. You can refactor the code to change the way it works so it better supports you. **Refactoring enhances code to support new requirements.** The techniques are all about not changing what it does, you won't break the existing functionality. But now the code will support your new requirements better, allowing you to make the same changes with less new code to write and maintain.

## Problems with Refactoring

When you learn a new technique that greatly improves your productivity, it is hard to see when it does not apply. Usually you learn it within a specific context, often just a single project. It is hard to see what factors cause that technique to be less effective, even harmful. Ten years ago it was like that with Objects. If someone asked me when they shouldn't use objects, it was hard to answer. It wasn't that I didn't think objects had limitations - I'm too cynical for that. It was just that I

didn't know what those limitations were, while I knew what the benefits were.

Refactoring is like that now. We know the benefits of refactoring, we know they can make a palpable difference to our work, but we haven't had the broad enough experience to see where the limitations apply.

So this section is shorter than I would like, and is more tentative. As more people learn about refactoring we will learn more. For you this means that while I certainly believe you should try refactoring for the real gains it can provide, you should also monitor it's progress. Look out for problems that refactoring may be introducing. Let us know about these problems. As we learn more about refactoring we will come up with more solutions to problems, and learn about what problems are difficult to solve.

## Databases

The first problem area for refactoring is with databases. Most business applications are tightly coupled to the database schema that supports them. That's one reason that the database is difficult to change. Another reason is data migration. Even if you have carefully layered your system to minimize the dependencies between the database schema and the object model, changing the database schema forces you to migrate the data - which is a long and fraught task.

With non-object databases the way to deal with this problem is to realize that you do not have to keep the object model and the database model looking similar. The two models can be different, at the cost of making the mapping code more complex. Most of your refactoring will occur in the object model. You then need to update your database mapping code to cope with the data changes produced by the refactoring. On the whole this is not too bad providing you don't couple your mapping code too deeply into the object model. If you give the object model a clear interface for the mapping code to use, then you can retain that interface as you refactor the object model.

You don't have to start with a clear interface to do this. As you notice parts of your object model become volatile, you can create the interface for them then. This way you get the greatest leverage for your changes.

You can also change your database schema, but this is not something that is as suitable to taking small steps due to the migration issues. Certainly database schema design needs more planning than the object model.

One technique is to create dummy classes that look like database tables to help you manage a change. Before you change the table design, create a class that mimics the original table. Redirect all uses of this table through this class. Then when you change the table you only need to change this one class. If using this class introduces inefficiencies you can then remove this class in a step by step manner. Although this adds an overhead to create the class, it allows you change the mapping code in smaller steps.

(I can't currently recommend an eminent book that discusses this approach to layering an information system in the presence of databases. I discuss it in chapter 12 of [Fowler, AP], but really it deserves a whole book.)

Object databases can be more work. With no separation between the object model and the data model - all your refactoring has to work with data migration. In this situation you have to be more wary about changes to the data structure of classes. You can still freely move behavior around, but you have to be more cautious with moving fields. You need to use accessors to give the illusion that the data has moved, even when it hasn't. Then when you are pretty sure you know where the data ought to be, you can move and migrate the data in single move. Only the accessors need to change, reducing the risk of problems with bugs.

## Changing Interfaces

One of the important things about objects is that they allow you to change the implementation of a software module separately from changing the interface. You can safely change the internals of an object without anyone else worrying about it, but the interface is important - change that and anything can happen

Something that is disturbing about refactoring is that many of the refactorings do change an interface. Something as simple as *Rename*

*Method (234)* is all about changing an interface. So what does this do the treasured notion of encapsulation?

There is no problem changing a method name if you have access to all the code that calls that method. Even if the method is public - as long as you can reach and change all the callers, you can rename the method. There is only problem if the interface is being used by code that you cannot find and change. When this happens I say that the interface becomes a *published interface* (a step beyond a public interface). Once you publish an interface you can no longer safely change it and just edit the callers. You need a somewhat more complicated process.

This notion changes the question. Now the problem is: what do you do about refactorings that change published interfaces?

In short, if a refactoring changes a published interface you have to retain both the old interface and the new one, at least until your users have had a chance to react to the change. Fortunately this is not too awkward to do. You can usually arrange things so the old interface still works. Try to do this so that the old interface calls the new interface. In this way when you change the name of a method, keep the old one an just let it call the new one. Don't copy the method body - that leaves you down the path to damnation by duplicated code. You should also use the deprecation facility in Java to mark the code as deprecated. That way your callers will know that something is up.

All this is usually doable, but it is a pain. You have to build and maintain these extra methods, at least for a time. The methods complicate the interface, making it harder to use. There is an alternative - don't publish the interface. Now I'm not talking about a total ban here, clearly you have to have published interfaces. If you are building APIs for outside consumption, like Sun does, then you have to have published interfaces. The reason I say this is because so often I see development groups using published interfaces far too much. I've seen a team of three people operate where each person published their interfaces to the other two. This lead to all sorts of gyrations to maintain interfaces when it would have been easier to go into the code base and make the edits. Organizations with an over-strong notion of code ownership tend to get like this. Using published interfaces is useful, but it comes with a cost. So don't use it unless you really need to. This may

mean modifying your code ownership rules to allow people to change other people's code in order to support an interface change. Often it is a good idea to do this with pair programming.

> *Don't publish interfaces prematurely. Modify your code ownership policies to smooth refactoring.*

There is one particular area with problems in changing interfaces in Java. This is if you add an exception to the throws clause. This is not a change in signature, so you cannot use delegation to cover for it. The compiler will not let it compile, however. It is tough to deal with this. You can choose a new name for the method, let the old method call it, and convert the checked into an unchecked exception. You can also just throw an unchecked exception, although then you lose the checking ability. When you do this you can alert your callers that it will become a checked exception at a future date. They then have some time to put the handlers into their code. For this reason I prefer to define a superclass exception for a whole package (such as SQLException for java.sql) and ensure that public methods only declare this exception in their throws clause. That way I can define subclass exceptions if I want to, but this won't affect a caller that only knows about the general case.

## Design Changes that are Difficult to Refactor

Can you refactor your way out of any design mistake, or are some design decisions so central that you cannot count on refactoring to change your mind later? This is an area where we have very incomplete data. Certainly we have often been surprised by situations where we can refactor efficiently. But there are places where this is difficult. One project found it very hard to refactor a system built with no security requirements into one with good security.

At this stage my approach is to imagine the refactoring. As I consider design alternatives I ask myself how difficult it would be to refactor from one design into another. If it seems easy, I don't worry too much about the choice, and I pick the simplest design - even if it does not cover all the potential requirements. However if I cannot see a simple

## Why Refactoring Works

*- - Kent Beck*

Programs have two kinds of value: what they can do for you today and what they can do for you tomorrow. Most times when we are programming, we are focused on what we want the program to do today. Whether we are fixing a bug or adding a feature, we are making today's program more valuable by making it more capable.

You can't program long without realizing that what the system does today is only a part of the story. If you can get today's work done today, but you do it in such a way that you can't possibly get tomorrow's work done tomorrow, then you lose. Notice, though, that you know what you need to do today, but you're not quite sure about tomorrow. Maybe you'll do this, maybe that, maybe something you haven't imagined yet.

I know enough to do today's work. I don't know enough to do tomorrow's. But if I only work for today I won't be able to work tomorrow at all.

Refactoring is one way out of the bind. When you find that yesterday's decision doesn't make sense today, you change the decision. Now you can do today's work. Tomorrow, some of your understanding as of today will seem naive, so you'll change that, too.

What is that makes programs hard to work with? Four things that I can think of as I am typing this are:

- programs that are hard to read are hard to modify
- programs that have duplicated logic are hard to modify
- programs where additional behavior requires you to change running code are hard to modify
- programs with complex conditional logic are hard to modify

So, we want programs that are easy to read, that have all logic specified in one and only one place, where changes do not endanger existing behavior, and where conditional logic is expressed as simply as possible.

Refactoring is the process of taking a running program and adding to its value, not by changing its behavior, but by giving it more of these qualities that enable us to continue developing at speed.

### Indirection and Refactoring

*"Computer Science is the discipline that believes all problems can be solved with one more layer of indirection." --DennisDeBruler*

Given software engineers infatuation with indirection, it may not surprise you to learn that most refactoring introduces more indirection into a program. Refactoring tends to break big objects into several smaller ones and big methods into several smaller ones.

Indirection is a two-edged sword, however. Every time you break one thing into two pieces, you have more things to manage. It also can make a program harder to read, as an object delegates to an object delegating to an object. So you'd like to minimize indirection.

Not so fast, buddy. Indirection can also pay for itself. Here are some of the ways:

- **To enable sharing of logic.** For example, a sub-method invoked in two different places or a method in a superclass shared by all subclasses.
- **To explain intention and implementation separately.** Choosing the name of each class and the name of each method gives you an opportunity to explain what you intend. The internals of the class or method explains how the intention is realized. If the internals are also written in terms of intention in yet smaller pieces, you can write code that communicates most of the important information about its own structure.
- **To isolate change.** I use an object in two different places. I want to change the behavior in one of the two cases. If I change the object, I risk changing both. So I first make a subclass and refer to it in the case that is changing. Now I can modify the subclass without risking an inadvertent change to the other case.
- **To encode conditional logic.** Objects have a fabulous mechanism, polymorphic messages, to flexibly but clearly express conditional logic. By changing explicit conditionals to messages, you can often reduce duplication, add clarity, and increase flexibility all at the same time.

So, here is the refactoring game — maintaining the current behavior of the system, how can you make your system more valuable, either by increasing its quality or reducing its cost?

The most common variant of the game is to look at your program. Identify a place where it is missing one or more of the benefits of indirection. Put that indirection in without changing the existing behavior. Now you have a more valuable program because it has more qualities that we will appreciate tomorrow.

Contrast this with careful up-front design. Speculative design attempts to put all the good qualities into the system before any code is written. Then the code can just be

hung on the sturdy skeleton. The problem with this process is that it is too easy to guess wrong. With refactoring, you are never in danger of being completely wrong-the program always behaves at the end like it did at the beginning. In addition, you have the opportunity to add valuable qualities to the code.

There is a second, rarer refactoring game. Identify indirection that isn't paying for itself and take it out. Often this takes the form of intermediate methods that used to serve a purpose but no longer do. Or it could be a component that you expected to be shared or polymorphic, but turned out only to be used in one place. When you find parasitic indirection, take it out. Again, you will have a more valuable program, not because there is more of one of the four qualities above, but because it costs less, it costs less indirection to get the same amount of the qualities.

way to refactor, then I put more effort into the design. I do find such situations are in the minority.

## When Shouldn’t you Refactor?

There are times when you should not refactor at all.

The principle example is when you should rewrite from scratch instead. There are times when the existing code is such a mess that, although you could refactor it, it will be easier to start from the beginning. This decision is not an easy one to make, and I’ll admit that I don’t really have good guidelines for it.

A clear sign of this is when the current code just does not work. You may only discover this by trying to test it, and discovering that it is so full of bugs that you cannot get it stable. Remember code has to work mostly correctly before you refactor.

A compromise route is to refactor a large piece of software into components with strong encapsulation. Then you can make a refactor vs. rebuild decision for each component one at a time. This is a promising approach, but I don’t have enough data to write good rules yet for

doing that. With a key legacy system, this would certainly be an appealing direction to take.

The other time you should avoid refactoring is when you are close to a deadline. At this point the productivity gain you would get from refactoring would appear after the deadline, and thus be too late. Ward Cunningham has a good way to think of this. He describes unfinished refactoring as going into debt. Most companies need some debt in order to function efficiently. However with debt comes interest payments: i.e. the extra cost of maintenance and extension caused by overcomplex code. You can bear some interest payments, but if the payments get too great then you will be overwhelmed. Thus it is important to manage your debt, paying parts of it off with refactoring.

Other than when you are very close to a deadline, however, you should not put off refactoring because you haven't got time. Experience from several projects has shown that a bout of refactoring results in increased productivity. So not having enough time usually is a sign that you need to do some refactoring.

## Refactoring and Up Front Design

Refactoring has a special role as a complement to design. When I first learned to program, I just wrote the program and muddled my way through it. In time I learned that thinking about the design in advance helped me avoid costly rework. In time I got more into this style of *up front design*. Many people consider that design is the key piece and that programming is just mechanics – drawing the analogy of the design as an engineering drawing and the code as the construction work. But software is still very different to physical machines. It is much more malleable and it is all about thinking. As Alistair Cockburn puts it "with design I can think very fast, but my thinking is full of little holes".

There is an argument which says that refactoring could be an alternative to up front design. In this scenario you don't do any design at all. You just code the first approach that comes into your head, get it working, and then refactor it into shape. Actually this approach can work. I've seen people do this and come out with a very well designed

piece of software. Those who support Extreme Programming are often portrayed as advocating this approach.

But while doing only refactoring does work, it is not the most efficient way to work. Even the extreme programmers will do some design first. They will try out various ideas with CRC cards or the like, until they have a plausible first solution. Only after generating a plausible first shot will they code and then refactor. The point is that refactoring changes the role of up front design. If you don't refactor then there is a lot of pressure in getting that up front design right. The sense is that any changes to the design later on are going to be expensive. Thus you put more time and effort into the up front design to avoid the need for such changes.

But with refactoring the emphasis changes. You still do up front design, but now you don't try to find *the* solution. Instead all you want is some reasonable solution. You know that as you build the solution, as you understand more about the problem, you will realize that the best solution is different from that you originally came up with. With refactoring this is not a problem, for it is no longer expensive to make the changes.

An important result of this change in emphasis is a greater movement towards simplicity of design. Before I used refactoring I always looking for flexible solutions. With any requirement I would be wondering about how that requirement would change during the life of the system. Since design changes were expensive I would look to build a design that would stand up to the changes that I could foresee. The problem with building a flexible solution like this is that flexibility costs. Flexible solutions are more complex than simple ones. The resulting software is then more difficult to maintain in general, although easier to flex in the direction I had in mind. Although even there you have to understand how to flex the design. For one or two aspects this is no big deal, but changes occur throughout the system. Building this flexibility in all these places makes the overall system a lot more complex and expensive to maintain. The big frustration, of course, is that all this flexibility is not needed. Some of it will be, and it's impossible to predict which pieces of flexibility are needed. So to gain the flexibility you need you are forced to put in a lot more flexibility than you actually need.

With refactoring you approach the risks of change differently. You still think about potential changes, you still consider flexible solutions. But instead of implementing these flexible solutions you ask yourself "how difficult is it going to be to refactor a simple solution into the flexible solution". If, as happens most of the time, the answer is "pretty easy", then you just implement the simple solution.

So refactoring can lead to simpler designs, without sacrificing flexibility. This makes the design process easier, and less stressful. Once you get a broad sense of those things that refactor easily, you don't even think of the flexible solutions. You have the confidence to refactor if the time comes. You build the simplest thing that could possibly work. As for the flexible, complex design: most of the time you aren't going to need it.

## Refactoring and Evolutionary Development

If you've read much that I've written over the years, you'll know that I consider evolutionary development to be essential for developing software systems. Indeed I agree with Alistair Cockburn that not just is incremental development a critical success factor in a project, but that its absence is a critical failure indicator.

Refactoring is important to evolutionary development because it is the mechanism that makes the iterative part of evolutionary development work well. When people talk about this style of development they are usually talking about two adjectives: iterative and incremental. Incremental development implies adding functionality in pieces as fully tested and integrated chunks. A project could be incremental and not iterative if it added the software chunk by chunk and never changed a prior chunk when it added new ones. Iterative development, however, implies reworking existing chunks. A purely iterative development would continuously rework the same functionality until it was satisfactory. Good evolutionary development is both incremental and iterative, with the incremental part taking the lead.

Refactoring is something you do continuously throughout the project. It is driven by the incremental process. Typically you begin a new increment of function to add to the system. You realize the structure of

the existing code makes it harder to add the new function than it might be. You thus refactor the existing code to make it easy to add the new function. Such an incremental driver is important because it provides the justification for refactoring. If you refactor just for the sake of refactoring, you end up suffering from prototypis - where you just iterate and iterate, never delivering, until your customer puts you out of their misery.

So should refactoring always be driven by adding a new increment? My answer is “mostly”. Sometimes you get the situation where some code needs to be refactored but it isn’t really for a specific increment. It wouldn’t be worth doing for just that increment, but as increments follow each other there is a collective weight that does make it worthwhile. A few times I’ve seen a project losing productivity and developers feeling uncomfortable about the state of the code base. Then it is worthwhile to concentrate on refactoring for a couple of weeks, even if it means pushing off some incremental function to a future stage. You shouldn’t do this often (maybe a couple of times a year) and you should keep such bouts short (within a single incremental stage).

## Pair Programming

Pair Programming is a technique that is frequently stressed by extreme programmers. It describes programming done by having developers work on the same code at a single workstation.

The first reaction to pair programming is usually a puzzled opposition.

> *Surely if two programmers are working together on the same machine they can only go half as fast two programmers working separately? This would be true if the hardest part of programming was typing.*

My contention is that the key to productivity is producing software with a high internal quality and keeping that internal quality high. Code that is difficult to understand is hard to change - which slows

you down. Code that has bugs hidden in it needs to be debugged, which means you have to find the bugs - which slows you down. If you have a code base that is well structured and free of bugs you can quickly add new features - and productivity is all about the speed with which you add new features.

Having two programmers work together improves internal quality and productivity for several reasons.

- Often you lose time by working your way down a dead end. One developer, head deep in the code, is more likely to get stuck like this. A second head more than often spots this and stops it.
- Two programmers can freely exchange design ideas and as such keep feeding better designs into the software.
- A second programmer helps spot design mistakes by acting as a real time code inspector.
- Two programmers are less susceptible to interruption. While one is fielding a phone call the other is making progress. With the partner there the interrupted developer comes back up to speed more quickly.
- If two programmers work on every piece of code, you have always have two people who know about every piece of code. This leaves you less vulnerable if one is unavailable due to sickness, vacation, or departure.
- Working with a partner allows them to teach each other development techniques, improving both their practices.
- Also working with a partner is more fun, increases team spirit, which increases motivation. And motivation is the key to productivity.

As in many areas of software development, there is not much objective evidence to back up my assessment. However [Nosek] reported an experiment with experienced system programmers that showed that pair programmers produced more functional and readable solutions, expressed higher confidence in their work, and enjoyed their work more than lone programmers.

You may have experienced this yourself. Many developers have done a complex piece of work jointly with another developer, and remem-

ber how that particular piece of development went very well. Or they remember being coached by someone more experienced.

Well maybe you're not convinced about the benefits of pair programming in general development. But in refactoring pair programming has a particular virtue. Often refactoring occurs when a programmer needs to add a new feature to existing code and that existing code was written by someone else. In these cases you should have both the original author (who knows the code) and the one adding the function (who knows what they want to do with the code) work together.

## Refactoring and Performance

A common concern with refactoring is the effect it has on the performance of a program. In order to make the software easier to understand, you often make changes that will cause the program to run more slowly. This is an important issue. I'm not one of the school of thought that ignores performance in favor of design purity or in hope of faster hardware. Software has been rejected for being too slow, and faster machines merely move the goalposts. Refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning. And the secret to fast software, in all but hard-real time contexts, is to write tunable software first, and then to tune it to get sufficient speed.

In general I've seen three approaches to writing fast software. The most serious of these is time budgeting, used often in hard real time systems. This situation as you decompose the design you give each component a budget for resources: time and footprint. That component must not exceed it's budget, although some mechanism for exchanging budgeted times is allowed. Such a mechanism focuses hard attention on hard performance times. It is essential for such systems as heart pacemakers when late data is always bad data. But this technique is overkill for other kinds of systems, such as the corporate information systems that I usually work in.

The second approach is the constant attention approach. Here every programmer, all the time, does whatever they can to keep performance high. This is a common approach, has intuitive attraction, but

## It takes a while to create nothing

*- - Ron Jeffries*

The Chrysler Comprehensive Compensation pay process was running too slowly. Although we were still in development, it began to bother us, since it was slowing down the tests.

Kent Beck, Martin and I decided we'd fix it up. While I waited for us to get together, I was speculating, based on my extensive knowledge of the system, about what was probably slowing it down. I thought of several possibilities and chatted with folks about the changes that were probably necessary. We came up with some really good ideas about what would make the system go faster.

Then we measured performance using Kent's profiler. None of the possibilities I had thought of had anything to do with the problem. Instead, we found that the system was spending half its time creating instances of Date. Even more interesting was that all the instances had the same couple of values.

When we looked at the Date creation logic, we saw some opportunities for optimizing how these dates were created: they were all going through a string conversion even though there were no external inputs involved. The code was just using string conversion for convenience of typing. Maybe we could optimize that.

Then we looked at how these dates were being used. It turned out that the huge bulk of them were all creating instances of Date Range, an object with a from-date and a to-date. Looking around little more, we realized that most of these Date Ranges were empty!

As we worked with date range we used the convention that any date range that ended before it started was empty. It's a good convention which fits in well with how the class works. Soon after we started doing this we realized that just creating a date range that starts after it ends wasn't clear code, so we extracted that behavior into a factory method for empty date ranges.

We had made that change to make the code clearer, but now we got an unexpected payoff. We created a constant empty date range and adjusted the factory method to return that object instead of creating it every time. That change doubled the speed of the system, enough for the tests to be bearable. It took us about five minutes.

I had speculated with various members of the team (Kent and Martin deny participating in the speculation) on what was likely wrong with code we knew very well. We had even sketched some designs for improvements, without first measuring what was going on.

We were completely wrong. Aside from having a really interesting conversation, we were doing no good at all.

The lesson is: even if you know exactly what is going on in your system, measure performance, don't speculate. You'll learn something, and nine times out of ten, it won't be that you were right!

does not work very well. Changes that improve performance usually make the program harder to work with. This slows down development speed. This would be a cost worth paying if the resulting software was quicker, but usually it is not. The performance improvements are spread all around the program, and each one is done with a narrow perspective of the program's behavior.

The interesting thing about performance is that if you analyze most programs, you find that they most of their time in a small fraction of the code. So if you optimize all the code equally, you end up with 90% of the optimizations wasted, because you are optimizing code that isn't run much. The time spent making it fast, the time lost because of the lack of clarity, is all wasted time.

The third approach to performance improvement takes advantage of this statistic. In this approach you build your program in a well-factored manner, without paying attention to performance, until you begin a performance optimization stage (usually fairly late in development). During the performance optimization stage you follow a specific process to tune the program.

You begin by running the program under a profiler that monitors the program and tells you where it is consuming time and space. This way you can find that small part of the program where the performance hot spots lie. Then you focus on those performance hot spots and use the same optimizations that you would use if you were using the constant attention approach. But since you are focusing your attention on a hot spot, you are having much more effect for less work. Even so you remain cautious. Like refactoring you make the changes in small steps. After each step you compile, test, and re-run the profiler. If you haven't improved performance, you back out the change. You continue the process of finding and removing hot spots until you get the performance that your users are satisfied with.

Having a well-factored program helps you do this in two ways. Firstly it gives you time to spend on the performance tuning. Since you have well factored code, you can add function quicker. This gives you more time to focus on performance. (And profiling ensures you focus that time on the right place.) Secondly with a well-factored program you have a finer granularity for your performance analysis. Your profiler leads you to smaller parts of the code, which are easier to tune. Since the code is clearer, you have a better understanding of your options, and of what kind of tuning will work.

So I've found that refactoring helps me write fast software. It slows the software down in the short term while I'm refactoring; but it makes the software easier to tune during optimization. I end up well ahead.

## Where did refactoring come from?

I've not succeeded in pinning down the real birth of the term refactoring. Certainly good programmers have spent at least some time cleaning up their code. They do this because they have learned that clean code is easier to change than complex and messy code, and good programmers know that they rarely write clean code first time around.

Refactoring goes beyond this. In this book I'm advocating refactoring as a key element in the whole process of software development. Two of the first people to recognize this were Ward Cunningham and Kent Beck, who worked with Smalltalk from the 80's onwards. Smalltalk is an environment which, even then, was particularly hospitable to refactoring. It is a very dynamic environment which allows you to quickly write very functional software. It has a very short compile-link-execute cycle, which makes it easy to change things quickly. It is also object-oriented, and thus gives you powerful tools to minimize the impact of change behind well-defined interfaces. Ward and Kent worked hard at developing a software development process which was geared to working with this kind of environment. (Kent currently refers to this style as *Extreme Programming*.) They realized that refactoring was a big factor in improving their productivity and ever since have been working with refactoring, applying it to serious software projects and refining the process.

## Optimizing a Payroll System

*- - Rich Garzaniti*

We had been developing Chrysler Comprehensive Compensation System for quite a while before we started to move it to GemStone. Naturally, when we did that, we found that the program wasn't fast enough. We brought in Jim Haungs, a master GemSmith, to help us optimize the system.

After a little time with the team to learn how the system worked, Jim used GemStone's ProfMonitor feature to write a profiling tool that plugged into our functional tests. The tool displayed the numbers of objects that were being created and where they were being created

To our surprise, the biggest offender turned out to be the creation of strings. The biggest of the big was repeated creation of 12,000-byte strings. This was a particular problem because the string was so big that GemStone's usual garbage collection facilities wouldn't deal with it. Because of its size, GemStone was paging the string to disk every time it was created. It turned out the strings were being built way down in our IO framework, and they were being built three at a time, for every output record!

Our first fix was to cache a single 12,000-byte string, which solved most of the problem. Later, we changed the framework to write directly to a file stream, which eliminated the creation of even the one string.

Once the huge string was out of the way, Jim's profiler found similar problems with some smaller strings: 800 bytes, 500 bytes and so on. Converting these to use the file stream facility solved them as well.

With these techniques we steadily improved the performance of the system. During development it looked like it would take over a thousand hours to run the payroll. When we actually got ready to start it took forty hours. After a month we got it down to around 18, when we launched we were at 12. After a year of running and enhancing the system for a new group of employees it was down to 9.

Our biggest improvement was to run it in multiple threads on a multi-processor machine. The system wasn't designed with threads in mind, but as it was so well factored it took us only three days to run in multiple threads. Now the payroll takes a couple of hours to run.

Before Jim provided a tool that measured the system in actual operation, we had good ideas about what was wrong. But it was a long time before our good ideas were the ones that needed to be implemented. The real measurements pointed in a different direction, and made a much bigger difference. Jim says with performance optimization you have to think like a tailor — "measure twice, cut once"

Their ideas have always been a strong influence on the Smalltalk community, and the notion of refactoring has become a important element in the Smalltalk culture. Another leading figure in the Smalltalk community is Ralph Johnson, a professor at the University of Illinois at Urbana-Champaign who is famous as one of the [Gang of Four]. One of Ralph's biggest interests is in developing software frameworks, and he explored the ideas of how refactoring can help develop an efficient and flexible framework.

Bill Opdyke was one of Ralph's doctorate students, particularly interested in frameworks. He saw the potential value of refactoring, and saw that it could be applied to much more than just Smalltalk. His background was in telephone switch development, where over time a lot of complexity accrues, making changes difficult to make. His doctorate research looked at refactoring from a tool builder's perspective. He investigated the refactorings that would be useful for C++ framework development, and researched the necessary semantics-preserving refactorings, how to prove they were semantics preserving, and how a tool could implement these ideas. His doctorate thesis [Opdyke] is the most substantial work on refactoring to date. He has also contributed Chapter 13 to this book.

I remember meeting BIll at OOPSLA 92, we sat in a cafe and discussed some of work I'd done in building a conceptual framework for healthcare. He told me about his research and I remember thinking: "interesting, but not really that important". Boy was I wrong!

John Brant and Don Roberts have taken the tool ideas in refactoring much further to produce Refactory, a refactoring tool for Smalltalk, which I'll discuss below. They also contributed Chapter 14 to this book.

And me? I'd always been inclined to clean code, but I'd never considered it to be *that* important. Then I worked on a project with Kent, and saw they way he used refactoring. I saw how the difference it made to productivity and quality. This convinced me that refactoring was a very important technique. However I was frustrated because there was no book that I could give to a working programmer, and not of the experts above had any plans to write such a book. So, with their help, I did.

## Tools for Refactoring

In this book I put a lot of emphasis on using small steps, frequent compile / test cycles, and strong tests. I do this because people make mistakes and nothing slows down software development more than debugging. Manual refactoring will always require tests to ensure that the external behavior of the software is unchanged.

Many of the refactorings I describe, however, are semantics-preserving refactorings. This means that it is possible to prove that such a refactoring, if done correctly, cannot cause a change to the behavior of the program. If you could make a semantics-preserving change and be absolutely certain you didn't make a mistake, then you wouldn't need to compile and test.

Humans cannot do this, but a computer tool could. With such a tool I could just indicate the refactoring I want, and the tool would carry out the refactoring. I win in two ways. Firstly the tool can do all the steps faster than I can, and secondly I don't have to even compile and test to ensure nothing is broken.

There has been a fair bit of research, both commercial and academic, into such software restructuring tools. Within the object-oriented community a principal such group has been Ralph Johnson's group at the University of Illinois at Urbana-Champaign.

Bill Opdyke's thesis [Opdyke] is all about research into refactorings that could be done by tools. He describes several refactorings and the mathematical proofs required to show the semantics preservation. John Brant and Don Roberts have built on this research to provide a practical tool for refactoring Smalltalk programs: *Refactory* [Refactory].

Strictly a discussion of Refactory is outside the scope of this book. Refactory is a Smalltalk tool and not usable for Java. However it is useful to understand the features of such a tool as I believe such tools will become available for Java over the next few years. Such tools will have a significant effect on software development.

I'll concentrate on a couple of the refactorings that Refactory provides. They are simple to use, and hide a lot that needs to be done to implement them.

Refactory allows you change the signature of any method, and it guarantees that all senders and implementers of that method change appropriately. You can even change the name of such fundamental methods as the new method, which creates new objects in Smalltalk. This is much more intelligent than a global search and replace. In a search and replace you have to be very careful with your search and replace strings (knowing your regular expressions backwards) to avoid an unintended side effects. A search and replace tool just looks at text, so it does not recognize when "amount" is used as a method name, a variable, or part of a print statement.

Refactory does a parse of the program and can thus guarantee that the change does not introduce a bug into the program. It's also much easier to use, no messing around with regular expressions, just select the old method in the browser, type in the new name, and blink.

One of the most impressive refactorings is the one to extract a method. If you take a look at the catalog, you'll see that method extraction can be quite involved due to the presence of local variables. You need to do quite a bit of analysis in order to do this and guarantee semantics preservation in all cases. With Refactory you highlight the code you want to extract. Refactory then analyzes the local variables in the code. Variables only used within the selected code are entirely moved in, local variables read but not written to are passed in as parameters. For more complex cases Refactory cannot do the transformation safely, so it issues a suitable message. When it can, which is most of the time, it prompts you to name the new method and its arguments and performs the transformation. All this takes but a few seconds, and you don't even need to test the result.

I believe such tools will be become common over the next few years. The companies who build them will get a significant competitive advantage over those that don't. If you have the chance it is worth spending some time with Refactory to familiarize yourself with what future software development tools will be like. Not just is this tool quite an amazing herald of the future, it is also free. That's one aspect of this tool that is never going to be improved on.