# Refactoring

Philipp Trucksäß

Eberhard-Karls-Universität Tübingen

**Abstract.** Since code changes dramatically during development, refactoring is necessary to improve the quality of software. This paper provides a systematic approach to efficient refactoring, in specific cases as well as in general. Strategies are discussed, focusing on efficiency and safety. The goal is to show how a schematic approach to refactoring effectively helps to improve code quality.

## 1 Need to Refactor

It is easy to mistake computer programming for a straightforward translation of a finished design into code. Of course every programmer who has ever worked on a software project knows that this is as far from the truth as it could be. The following paper is based on "Code Complete" by Steve McConnell[1]. In the chapter "Refactoring" he points out that 30-60% of time put into the development of a piece of software is spent on actual coding, proving that this stage is the major part of the process. Especially with contemporary coding practises like scrumming as an industry standard there is no denying that adaptation to upcoming challenges on the way to finished software and changing designs on the go is an integral part of what makes a program successful. "All successful software gets changed."(Fred Brooks)
Programming is an incremental process. There is no linear way from empty IDE to finished program. Fortunately, already written code is not set in stone but rather changes all the time during development. To embrace these changes as a chance to improve the overall quality programmers need a method to react to them. This method is refactoring.
It is not clear to everyone what exactly constitutes a refactoring. The term is derived from the concept of factoring, which refers to the idea of breaking a program's code up into small, manageable parts in order to make it more easily understandable and manageable. Factoring is seen at its prime in object-oriented programming.
Analogous, refactoring is "a change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior."[2]

## 2 Detecting "Code Smells"

The cardinal rule of software evolution is that internal quality should improve with code evolution[1]. To achieve this it is essential to identify parts of the

program that need improvement. Parts of code that have degraded over time or simply started out badly are commonly known as "code smells"[1]. These can be as simple as a variable whose use changed over the course of development, so that its name does no longer represent its responsibility. Changing a variable name is pretty straightforward, so it does not need to be discussed in detail.
Almost as straightforward is recognizing that a routine has grown too long as it has been further specified. As a rule of thumb, a routine is too long when it no longer fits into a single screen. If that is the case, possible refactorings may include extracting complex boolean expressions into their own routines or turning whole parts into their own subroutines with clearly defined parameters, optimally following the single-responsibility pattern.
Unfortunately you can not always as easily identify which refactorings are necessary. Most times there is not only one way to code a certain functionality. In such cases, the key to succesfully improving your code is sticking to a coherent abstraction. Abstraction is what makes a program comprehensible for humans. There are different levels of abstraction which are ordered hierarchically. On each level, the responsibilities or tasks of a structure unit (e.g. a class, routine, method or variable) are summarized in a simple sentence. This one sentence also contains the maximum amount of information other structure units are allowed to have, according to the concept of encapsulation. The decision to refactor in a certain way should be based on how it improves the conformity with the relevant abstraction. For instance, methods added to a class during development which do not agree with the class's abstract concept require extraction into their own specific class.
Another clue that a refactoring is necessary is repetition. If multiple branches of a conditional construct repeat the same statements, they need to be reformulated, so that every if/else only contains exclusive statements. Series of similar statements in different parts of the code call for a designated method encapsulating their execution. A C++ example for this:

```
Vector3D normalizedA=a/sqrt(a.x*a.x+a.y*a.y+a.z*a.z);
Vector3D normalizedB=b/sqrt(b.x*b.x+b.y*b.y+b.z*b.z);
```

Two 3D vectors are normalized with similar code, which is hard to read due to its mathematic nature. A better alternative is:

```
Vector3D normalizedA=a.normalized();
Vector3D normalizedB=b.normalized();

                                    ⋮

Vector3D Vector3D::normalized(){
return this/sqrt(x*x+y*y+z*z);
}
```

Not only does this refactoring increase the readability, but it also minimizes the number of places where errors occur. While this example is dumbed down and

virtually all 3D vector classes come with a "normalized"-method, this similarly applies to more complex, multiple line pieces of code, where the benefits are even bigger.
On a greater scale, classes having similar methods is an indicator of a relation, which is best represented by a common superclass.
Another "smell" which indicates bad design and is hard to read, is the use of setup and takedown code. Here is a C++ example of a withdrawal from a bank account, which requires parameters to be set up and taken down again in case they are changed in the process:

```
WithdrawalTransaction withdrawal;
withdrawal.SetCustomerId( customerId );
withdrawal.SetBalance( balance );
withdrawal.SetWithdrawalAmount( withdrawalAmount );
withdrawal.SetWithdrawalDate( withdrawalDate );

ProcessWithdrawal( withdrawal );

customerId = withdrawal.GetCustomerId();
balance = withdrawal.GetBalance();
withdrawalAmount = withdrawal.GetWithdrawalAmount();
WithdrawalDate = withdrawal.GetWithdrawalDate();
```

[1] There is a one line alternative for this if the arguments are passed on by reference:

```
ProcessWithdrawal( customerId, balance, withdrawalAmount
 , withdrawalDate);
```

[1] In languages which do not support the passing of parameters by reference, like Java, the withdrawal is best modeled as a method offered by an account object:

```
account.ProcessWithdrawal( withdrawalAmount, withdrawalDate);
```

That also requires less parameters, which is always a hint at an improvement of the code.
A reason to still prefer the first version is the use of an undo-function. That benefits of distinct operations modeled as individual objects. But in that case the better option (in Java) would still be:

```
Withdrawal withdrawal = new Withdrawal( withdrawalAmount
  , withdrawalDate);
account.ProcessWithdrawal( withdrawal );
account.undoLastWithdrawal();
```

This extensive example demonstrates how there is not one general refactoring for one "code smell". Choosing the right one requires being clear about the design used. There are cases where detecting a "code smell" brings attention to an

oversight in the way things are designed. Thus, refactoring is not limited to the actual code but can be extended to the "paper state" of programming.
For an exhaustive list of different specific refactorings, refer to "Code Complete"[1].

## 3 Strategies

Now that it has been made clear when refactorings are necessary, it is time to discuss the best strategies to get started. As to most kinds of work, the 80/20 rule applies to refactoring[1]. It states that 20% of the work provide 80% of the benefit. Therefore it is essential to identify the 20% of the possible refactorings which are the most rewarding.
Inherent to the nature of newly written or recently fixed code is its high level of entropy. The first priority when writing code is to make it work, not so much to structure it as well as possible. This leads to recently added passages in the code being comparatively unordered. It is helpful to spend the little extra time to refactor code that has just been tested and proven to work, while you still know what each statement does, and tidy up, so you can go back to it later and seamlessly put it to work with all the other parts.
The tricky part is finding regular intervals to switch from hacking code to refactoring it. Refactoring should not be mistaken for the last step necessary before a program is ready for release. As discussed above, it might expose the need to further specify the design. Regular refactoring sessions helps recognizing "dead ends" in development early on. Scrumming offers a viable rule of thumb for subdividing the project. Just as every sprint should leave you with a fully functional increment of your software that could be given to a user, every such increment is a good breakpoint to go from coding to refactoring.
Since the number of possible refactorings is still rather big, there are parts that deserve more focus than others. Refactoring targets which are especially rewarding are those parts which are most complex or error-prone. If a routine requires more comment than actual statements to make it understandable, it is a good idea to refactor it, so it speaks for itself. Even though it might be an intimidating task to refactor code that you do not understand intuitively, the merit of increased readability is an easier understanding at future reference by you as well as others.
But it is important to keep in mind, that refactoring does not change a program's outward behavior, so there is no point refactoring broken code in order to make it work. However, in some cases refactoring code that does not work increases the understanding of what is going wrong, and is therefore helpful in its own right. If so, it does still not replace the final refactoring as soon as the errors are fixed.
If potential errors derive from user inputs or hardware readings, it pays off to refactor the relevant parts of the code in such a way that possible exceptions are isolated in an easily manageable segment, where they can be treated appropriately. Concentrating the refactoring effort on these situations keeps you from

getting lost in the refactoring process, while still maximizing the advantage it offers.
On the other hand there are cases when refactoring is a waste of effort. As stated earlier, it does not change the outward behavior of a program so it will not make broken code work and should not be used to cover up sloppy code. If the code solutions implemented prove to be flawed, refactoring has to make way for rewriting.

## 4 Safety

"There is no code so big, twisted, or complex that maintenance can't make it worse"[3]
Refactoring can cause more harm than good. But a few precautions can help to prevent possible missteps. A study[3] shows that against expectations the chance of producing additional errors when refactoring does not increase linearly with the lines of code changed. Instead, it increases from a 50% chance of error with one line of changed code to 80% with five lines and then decreases again to 30% with 20 changed lines. The obvious conclusion that can be drawn from that data is that programmers tend to treat smaller changes less carefully, and the additional preparation before bigger refactorings pays off in the long run since it reduces the time and effort that has to be put into debugging afterwards. As a consequence it is advised to treat any small change with the same attention as a big one, for instance, discussing it with colleagues, planning it on paper etc. So which precautions should be taken in particular? At first, backup is essential, so you can always go back to a working version of the code. If you work with SVN or Git, this means commiting before you start changing anything.
Then break up any change into small, clear parts, each of which leaves you with a working piece of software at completion. If they are now executed one at a time, any occurring error is easily traceable, which greatly reduces the necessary work put into it.
During refactoring is often the first time code is reviewed after it was initially written, so it is very likely that you come across further refactorings you might want to make. In order not to get sidetracked, put these on the backburner. Use a "parking lot"[1], a list containing possible future changes, so you do not forget about them, but finish the alternation you are currently working on before you move on to the next one.
After each of these atomic changes, retest and review the relevant code and add new test cases if necessary.

## 5 Benefits of Refactoring

As we can see, refactoring is not only necessary, but also provides a structured method to improve the overall code quality. In a contemporary work environment it is the obvious systematic way to deal with the requirements of an evolving

code project.
However, refactoring should not be mistaken for an excuse to write sloppy code with the intention to fix it later on. On the contrary, a systematic approach to refactoring increases the sensibility towards the subtle differences between alternative ways to implement a certain concept. In the long run that awareness serves to reduce the number of changes needed.
For the number of changes which are unavoidable, they are not a flaw of the programmer and are better dealt with pragmatically in the ways previously discussed. That does not only improve the overall code quality, but also helps to make the software design rock solid.
Keep in mind that "Refactoring during development is the best chance you'll get to improve your program, to make all the changes you'll wish you'd made the first time"[1]. It is better to put in the extra effort during the development stage instead of postponing it until maintenance and having to deal with angry customers on top of the work on the code.

## References


1. McConnel, S.: *Code Complete: A Practical Handbook of Software Construction: A Practical Handbook of Software Costruction*
   Microsoft Press, 2nd Edition(7.7.2004)
   ISBN: 978-0735619678
2. Fowler, M.: *Refactoring: Improving the Design of Existing Code (Object Technology Series)*
   Addison Wesley(28.6.1999)
   ISBN: 978-0201485677
3. Weinberg, G.: *Kill That Code!*
   Infosystems (August 1983)