Refactoring

Articles

Posted by:

Posted on : 2007/9/17 23:34:47

Refactoring is a powerful technique for improving existing software. Having source code that is understandable helps ensure a system is maintainable and extensible. This paper describes the refactoring process in general and some of the benefits of using automated tools to reliably enhance code quality by safely performing refactoring tasks.

Author: Oliver Whiler, Agris Software - www.refactorit.com

Overview

When a system's source code is easily understandable, the system is more maintable, leading to reduced costs and allowing precious development resources to be used elsewhere. At the same time, if the code is well structured, new requirements can be introduced more efficiently and with less problems. These two development tasks, maintenance and enhancement, often conflict since new features, especially those that do not fit cleanly within the original design, result in an increased maintenance effort. The refactoring process aims to reduce this conflict, by aiding non destructive changes to the structure of the source code, in order to increase code clarity and maintainability.

However many developers and managers are hesitant to use refactoring. The most obvious reasons for this is the amount of effort required for even a minor change, and a fear of introducing bugs. Both of these problems can be solved by using an automated refactoring tool

Refactoring – an emerging software development activity

Who needs refactoring?

Software starts on a small scale, and most of it well-designed. Over time, software size and complexity increases, with that bugs creep in, and thus code reliability decreases. Software developers, particularly when they are not the original authors, are finding it increasingly difficult to maintain the code, and even harder to extend. The code base, which in any software company should be a valuable asset, at some point may become a liability.

What is needed to prevent the software from ageing prematurely? Strategically, attention of management and software developers is the most important factor. On the practical side, application of sound development methods will slow this ageing down. However refactoring can reverse this ageing when applied properly, preferably with good software tools that aid in the detection, analysis, and characterisation of the problems, and ultimately allow fixing them.

Well trained software developers who are intimately familiar with their code, are often acutely aware of the lurking code ageing problems. However, most developers would be reluctant to make changes to the structure of the code, especially if the changes may take some time. If developers find it easy

to apply refactoring operations to their code, they will show less resistance to such restructuring work.

Rewriting a component is often seen as easier by the developer, or at least less confusing. The current source code may have changed over time from the original design, and may not be immediately clear to a developer who is seeing the code for the first time. Alternatively, the original developer may regret certain design decisions, and now believes there is a better way. However this ignores the fact that the source code has a lot of hidden value. The bug fixes contained in the source code, may not all be documented, however they are very valuable. The component has been previously tested thoroughly in the production environment, and this is not something that should be thrown away. Refactoring retains this hidden value, ensuring the behaviour of the system does not change.

Management are often unwilling to allow changes that will not give any immediate visible benefit, "If it ain't broke, don't fix it". As well, management may worry about the problem of introducing bugs to a system which has previously been thoroughly tested. This could happen, for example, when manually renaming a method where similarly named methods can be changed as well or methods overridden in a subclass are left with the original name. However, if refactoring operations do not pose a threat of introducing such bugs, management will be less reluctant towards letting refactoring proceed.

What exactly is refactoring?

Refactoring simply means "improving the design of existing code without changing its observable behaviour". Originally conceived in the Smalltalk community, it has now become a mainstream development technique. While refactoring tools make it possible to apply refactorings very easily, its important that the developer understand what the refactoring does and why it will help in this situation eg allow reuse of a repetitive block of code.

Each refactoring is a simple process which makes one logical change to the structure of the code. When changing a lot of the code at one time it is possible that bugs were introduced. But when and where these bug were created is no longer reproducible. If, however, a change is implemented in small steps with tests running after each step, the bug likely surfaces in the test run immediately after introducing it into the system. Then the step could be examined or, after undoing the step, it could be split in even smaller steps which can be applied afterwards.

This is the benefit of comprehensive unit tests in a system, something advocated by Extreme Programming techniques. These tests, give the developers and management confidence that the refactoring has not broken the system, the code behaves the same way as it behaved before.

A refactoring operation proceeds roughly in the following phases:
* Detect a problem: Is there a problem? What is the problem?
* Characterise the problem: Why is it necessary to change something? What are the benefits? Are there any risks?
* Design a solution: What should be the "goal state" of the code? Which code transformation(s) will move the code towards the desired state?
* Modify the code: Steps that will carry out the code transformation(s) that leave the code functioning

the same way as it did before.

Example Refactorings:

Rename

A method, variable, class or other java item has a name that is misleading or confusing. This requires all references, and potentially file locations to be updated. The process of renaming a method may include renaming the method in subclasses as well as clients. On the other hand, renaming a package will also involve moving files and directories, and updating the source control system.

Move Class

A Class is in the wrong package, it should therefore be moved to another package where it fits better. All import statements or fully qualified names referring to the given class need to be updated. The file will also have to be moved and updated in the source control system.

Extract Method

A long method needs to be broken up to enhance readability and maintainability. A section of code with a single logical task (e.g. find a record by id) is replaced with an invocation to a new method. This new method is given suitable parameters, return type and exceptions. By giving the method a clear and descriptive name (findRecordById), the original method becomes simpler to understand as it will read like pseudocode. Extracting the method also allows the method to be reused in other places, not possible when it was tangled amongst the larger method. If the extracted section is well chosen, this method may be a natural place to change the behaviour of the class through subclassing, rather than a copy and paste of the existing method before making changes.

Extract Superclass

An existing class provides functionality that needs to be modified in some way. An abstract class is introduced as the parent of the current class, and then common behaviour is "pulled up" into this new parent. Clients of the existing class are changed to reference the new parent class, allowing alternative implementations (polymorphism). Any methods which are common to the concrete classes are "pulled up" with definitions, while those that will vary in subclasses are left abstract. As well as aiding in efficient code re-use, it also allows new subclasses to be created and used without changing the client classes.
]

Replace Conditional with Polymorphism

Methods in a class currently check some value (if or switch statement) in order to decide the right action to perform. One trivial example is a class that draws a shape, which is defined by a width and type (circle or square). The code quickly becomes confusing as the same if or switch statements are repeated throughout the class, i.e. in methods that calculate the area or perimeter of the shape. By using polymorphism, the shape specific behaviour can be offloaded to subclasses, simplifying the code. This has the added benefit of allowing other subclasses, e.g. rectangle or star, to be introduced without extensive code changes.

With each problem above a more or less obvious solution has been stated, too. However, it is clear to every experienced software developer that there are more complicated code problems, for which simple solutions can not so easily be presented. Obviously, a software developer will usually apply refactorings successfully only, if he/she knows how the software should look like in the end. In other words, before trying to refactor some code, one needs to familiarise oneself with the common object oriented design patterns and refactorings (see Gamma et al. 1994; Grand 1998).

When should one consider refactoring?

Ideally, refactoring would be part of a continuing quality improvement process. In other words, refactoring would be seamlessly interwoven with other day-to-day activities of every software developer.

Refactoring may be useful, when a bug has surfaced and the problem needs to be fixed or the code needs to be extended. Refactoring at the same time as maintenance or adding new features, also makes management and developers more likely to allow it, since it will not require an extra phase of testing.

If the developer in charge finds it difficult to understand the code, he/she will (hopefully) ask questions, and begin to document the incomprehensible code. The phrases he/she coins may be a good starting point for the names of new methods or classes.

Often, however, schedule pressures do not permit to implement a clean solution right away. A feature may have to be added in a hurry, a bug patched rather than fixed. In these cases, the code in question should be marked with a FIXME note, in order to be reworked, when time permits. Such circumstances call not for individual refactorings, but for a whole refactoring project. When the time has come to address the accumulated problems, a scan for FIXMEs, TODOs, etc. over the code base will return all the trouble spots for review. They can then be refactored according to priority.

What are the benefits of refactoring?

Carrying out a few refactoring operations before or during code debugging may have immediate benefits. Often it becomes easier to spot the bug location. So time is saved, while at the same time the quality of the code is enhanced. Well-structured code is also less error-prone when it comes to extending it.

Kent Beck [Fowler, p.60] states that refactoring adds to the value of any program that has at least one of the following shortcomings:
* Programs that are hard to read are hard to modify.
* Programs that have duplicate logic are hard to modify
* Programs that require additional behaviour that requires you to change running code are hard to modify.
* Programs with complex conditional logic are hard to modify. Summarising, while sometimes there are immediate benefits to be reaped from refactoring, the real benefits normally come in the long term. They consist in substantially reduced time that developers spend on debugging and maintenance work, as well as in improved extensibility and robustness of the code. In addition, code duplication is reduced, and code re-use fostered. Overall maintenance and development cost should

come down, and the speed of the team to react to changing needs should improve.

What are the concerns, particularly of management?

If there are so many usually undisputed benefits from well-structured, comprehensible code, and if refactoring leads from chaotic, badly structured, error-prone code to well-designed code by performing a serious of small, controlled operations, why then is not every software developer engaged in refactoring?

Software developers are often reluctant, because some refactorings are simply tedious. And there is no visible external benefit for all the labour put in.

Also management may be at fault. As long as management only rewards externally visible code properties such as functionality and performance, but neglects to pay attention to the code’s inner quality, e.g. via code review or by failing to set coding standards, it is only to blame itself, when software developers are reluctant to invest some of their precious time on refactoring operations.

Then there is the serious risk of breaking the code through refactoring operations. And if e.g. file-names change, traceability of modifications is likely to become an issue, too.

Even if the software developer, is keen to refactor some badly structured code. His/her manager, however, may have quite a different perspective, and may oppose any attempt to modify working code. These concerns cannot simply be ignored, but must be suitably addressed by both developers and management.

Why use an automated tool?

When doing refactoring the externally observable behaviour must be guaranteed to stay the same. If refactorings are carried out manually, one needs to frequently rebuild the system and run tests. Manual refactoring is therefore really practical only, when the following conditions hold:
1. The system, of which the refactored code is a part, can be rebuilt quickly.
2. There are automated "regression" tests that can be frequently run. This situation is not very common, meaning that the applications of refactoring is limited. This situation is becoming more common, particularly as more people use XP (Extreme Programming) development methods.

Another hindrance is that many of these refactoring are tedious. Potentially requiring hours of precious development time, as the change is made and then thoroughly checked. Not many programmers would enjoy the task of renaming a method in a large code base. A simple search and replace will potentially find extra results. So each replacement must be examined by the programmer. However there is no great intelligence to the operation, all that is wanted is to rename any use of a method on a given class or its subclasses. A refactoring tool therefore can save hours of work. Even more importantly give confidence that the correct changes were made.

The speed of automated tools has another benefit, which is shown in team development environments. A developer is much less likely to perform refactoring operations if the source code involved is under the responsibility of other developers as well. However by using an automated tool, the refactorings can be completed quickly so that other developers are not held up waiting to make

there changes when the refactoring is completed, or worse making their changes on the old code at the same time as the refactoring operation. This ensures that even when the responsibility for a section of code is shared, developers will not reach a stale mate, where none of the developers make the required changes.

Integration with the developers chosen IDE also bring many benefits. Firstly having the tools at hand, means that developers can more easily refactor. They do not have to switch between development and refactoring modes, and can instead see it as part of their normal development cycle. Secondly IDE features such as Source Control Integration can reduce the effort in refactorings such as move class or rename package.

Conclusion

Refactoring is a well defined process that improves the quality of systems and allows developers to repair code that is becoming hard to maintain, without throwing away the existing source code and starting again. By careful application of refactorings the system's behaviour will remain the same, but return to a well structured design.

The use of automated refactoring tools, makes it more likely that the developer will perform the necessary refactorings, since the tools are much quicker and reduce the chance of introducing bugs.
Originally published in the Spring 2003 issue of Methods & Tools