Refactoring: a technology for managing change

1. Definition of refactoring

Refactoring is a set of techniques for making simple improvements to existing software code. A refactoring session can be done by a single software developer or a small group: identifying places in the existing code base that need to be "cleaned up." After a refactoring session is over, the code continues to have the same functionality as before, but the internal structure of the code has been simplified or clarified, which may allow developers to make subsequent changes more easily (and those changes could be either bug fixes or new features).

The concept of periodic "code cleanup" is an old software practice, but there wasn't much "theory" to explain the best ways to fix old code. The term "Refactoring" was coined in the early 1990s to refer to a disciplined, behavior-conserving code cleanup process. Several early-1990s software researchers, including Bill Opdyke at University of Illinois, created the first generation of tools to automate refactoring changes. In the 1990s, refactoring became especially valuable in parts of the software industry wherever long-lived systems required periodic restructuring – for example, in telecom software. Large and complex telecom switching systems needed occasional refactoring work to reduce internal complexity.

Over time, refactoring became a mature code improvement process in "agile and iterative software development." The agile community established a number of useful refactoring practices: "code smells" to determine where to refactor, catalogs of standard refactorings to explain how to refactor, automated development-tool-level support for many simple refactorings, and a set of clean code principles to act as a philosophical underpinning of the refactoring process.

Beyond the world of agile software development, refactoring has also been applied to software design and architecture. There are some good guides to using design patterns as a focus for design-level refactoring. The cloud computing and microservices communities have applied refactoring to the job of breaking up large applications into small configurable services. Finally, refactoring continues to be applied as a code analysis tool – developers who are faced with unfamiliar legacy code will perform selective "refactoring probes" in an attempt to explore the internal behavior of a large software system.

2. Refactoring as part of the software maintenance process

Refactoring has become an important skill for maintenance programmers: software developers charged with fixing bugs, porting systems to new platforms, and adding customer-requested features to an existing base of "legacy code." If a software change is made to a module that already been patched up multiple times, maintenance programmers like to do some cleanup work first. They try to do refactoring that preserves the behavior of the patched code and they adjust the code to make it look like it should have been written in the first place. The cleanup/refactoring step includes extensive testing to ensure that the refactoring changes does not introduce any new defects into the code base – because if new bugs are found, it is easier to unwind the failed modifications and do the restructuring right. Refactoring is safer than uncontrolled hacking without frequent retesting – undisciplined changes could result in an uglier code base that would be impossible to debug.

In maintenance programming, most of the simple refactoring changes are focused on two kinds of cleanup: reducing complexity and improving the understandability of the code. In the first category, refactoring changes include splitting up large functions, simplifying the code in key decision points in the software's algorithms, and introducing calls to standard libraries to replace custom code within the system. The refactoring changes to improve understanding include renaming variables and functions, simplifying data structures, introducing new classes in object oriented code, and reformatting the source files to make the code structure more obvious.

In the process of "refactoring legacy systems," it is rare to attempt to refactor an entire application. Developers prefer to make "focused" refactorings: making changes to a small subset of the code, perhaps only to software modules that were going to be modified to fix a bug or add a new feature. If a module has been stable, it is better not to do any refactoring work on it.

In some cases, legacy refactoring work includes the creation of new internal interfaces in the code. The "Facade pattern" is one of the best ways to restructure legacy code modules so that new functionality can be added. The new functionality is partitioned into in new functions or classes, and that new code calls on special "wrapper classes" are added that make direct calls to unmodified legacy modules. This refactoring practice is a good way to reduce the amount of refactoring work in stable legacy modules.

Developers in large projects need to perform refactoring work in a way that is consistent with the project's quality management practices. Each refactoring change may need to follow some bureaucratic change tracking process. Many large projects require that each code change be associated with a "change request" – which could be derived from a customer bug report, a management request for adding or modifying an application feature, or an internal change request to do some restructuring (such as updating an algorithm, improving application performance, or any other architecture or design priority). Refactoring work can be applied in such an environment using minor or major internal change requests. Of course, the management process for prioritizing and assigning development tasks will need to put these requests into the project workflow in a suitable way.

3. Refactoring as an agile practice

Refactoring as a practice to support software evolution important part of "agile development" beginning in the late 1990s. Agile development – a set of small-team development methods that emphasize rapid iteration, building very small features over a period of a few weeks, and responding quickly to customer feedback – started to gain a significant following in the mid-2000s. The early exponents of Extreme Programming (XP) were strong advocates of two important code quality practices: automated unit testing and code refactoring.

The majority of the books and articles on refactoring are aimed at "agile refactoring" practices: frequent, small-scale code cleanup that is performed by agile team members, either working alone or in small groups. The goal of refactoring is to simplify the code base and "keep it clean" so that future features will be easier to add.

In an agile project, refactoring can be done by anyone on the development team because XP mandates that there is no code ownership: anyone is free to modify anything in the project's code base. However, developers are required to verify that the project's unit tests continue to pass.

Agile developers are constantly working in a "prototyping mode," where they are often writing very incomplete small features in early iterations with a rough plan to extend the code in later iterations, depending on customer feedback. Developers might struggle to implement small features on a tight schedule, so the initial code quality of their feature code may be poor. Code modules might need to be restructured or replaced: at times, developers will tear out a module and completely rewrite it, but it is more common that they make minor step-by-step improvements.

Even a mature module in an agile development project may benefit from some refactoring effort. In some parts of the code base, a series of chaotic changes may add a lot of unnecessary complexity – and those poorly organized modules could become an obstacle to adding new customer features in future iterations. Agile teams often invest in a few days of refactoring from time to time, with the goal of improving the ability of the code base to be easily extended to support new features.

4. Agile refactoring and Code Smells

In an agile project, what is the best approach to decide where to refactor? In a large conventional project, maintenance programmers focus on where the next set of feature changes will be needed, or on the modules where there are the highest number of bug reports. The conventional strategy is to anticipate the location of future changes, then focus the refactoring work in those modules and subsystems.

On the other hand, agile projects move so fast that it is impossible to accurately anticipate the right places to refactor. Instead, the agile community has developed an approach based on looking around for Code Smells – a set of code characteristics that are often associated with troublesome code. Code is usually considered "smelly" if there are elements of the system that are large, complex, or confusing. Every time you detect a code smell, you have found a place in the code where the original developer wasn't thinking about keeping the code "clean."

The concept of Code Smells was introduced by Kent Beck and Martin Fowler in the 1999 book *Refactoring: Improving the Design of Existing Code*, and in the following years, many agilists have written about other code problems that they have discovered. Code smells are relatively easy to find, but they aren't always easy to fix.

Refactoring based on Code Smells is a more aggressive strategy than merely refactoring modules where you anticipate the next changes. But the Code Smells strategy can also generate a lot of unnecessary work, depending on how you use your "code smell" catalog. A list of code smells should never be considered as a strict management rule – a mandate to remove anything and everything that might be smelly. On the other hand, developers must be ready to "use their best judgment" about which smells require their attention.

One popular code smell is **Large Function**, but most code smell catalogs are silent about what should be the upper limit of the size of a function. This is always a matter of judgment – because to implement a key complex part of a big algorithm, large functions are often necessary. On the other

hand, more common internal functions can be meaningfully simplified by breaking off low-level helper functions for computations or input/output. **Excessively short identifier** is also a smell that calls for flexibility: although it is best to always use meaningful words to build good variable names, it is OK to have very brief and artificial "loop index" or "pointer index" variables (like i, j, or p) in a localized code block, because most developers will find this kind of code to be less cryptic than if you used a longer and more complex loop index name.

Some agile developers try to use a set of coding principles called Clean Code – first documented in Robert C. Martin's book *Clean Code*. Martin puts a big emphasis on readable code and short functions, and many developers like the idea of restructuring existing code modules to follow Martin's guidelines. However, it is a bad practice to be too picky about "following an arbitrary set" of clean code practices – especially when a project team is importing a set of legacy modules from another team who had different standards. It is just arbitrary busy work to apply dozens of format conversions to newly imported code. Worse, developers might disrupt the original design intent of a module or subsystem: by making big structural changes, the updated code base might obscure important clues and reduce developers' ability to understand the design choices made in the original code. Light refactoring of legacy code is OK, heavyweight refactoring can increase readability problems.

Refactoring makes it possible for all team members to "navigate" through the existing code more smoothly – to find the places where the next features might be added to the code base. For this reason, refactoring to address many of the code smells that are related to the "readability" of code are most worthwhile.

5. Refactoring to increase the use of standard libraries

The use of standard libraries for common computational tasks is advocated by most software engineering experts because libraries can improve the correctness and efficiency of the code, and library use can reduce long-term maintenance costs. Developers sometimes skip using some of the standard libraries in an agile implementation task – time pressure, unfamiliarity with the library, and a tendency to write narrowly-focused code are often responsible for writing inefficient ad hoc code that neglects to take advantage of well-designed library code.

This is one area where collaborative work across an agile team can make a difference. Even if the original developer doesn't have the time or knowledge to keep the code base clean and simple by using standard libraries, the developer's teammates are in a position to help make the code better. A positive impact of refactoring – particularly the kinds of modifications that use standard libraries, reduce duplication, and eliminate dead code – is a reduction in the number of source lines in the application. A smaller code base often leads to a reduction in long-term maintenance costs. Refactoring in the direction of using libraries instead of custom code is a good practice.

6. More agile practices that call for refactoring support

Many agile projects do not have enough unit test coverage, so adding new unit test cases in each iteration is a useful strategy. There are many good guides to introducing new unit tests, but one of the best is the 2004 book *Working Effectively With Legacy Code* by Michael Feathers. The book discusses a number of good strategies for "getting legacy code under unit test control."

The best unit test strategy is to design very small tests that are focused on exercising each individual function in isolation. Some tests are even smaller: there can be unit tests merely exercise a small block of code within a function.

Developing a unit test is an exercise in "understanding the collaborations" for the function under test and writing clever "fakes" based on that analysis. Much of the "test harness" is a set of small functions or classes that are test-specific substitutes for real system functionality. The developer will usually create small fake functions that feed test data into the function under test, plus more fake functions that simulate the functions that the function under test invokes. This kind of unit test design is a perfect way to make each unit test small and fast – so that a developer can run a large number of unit tests in a few minutes.

Some agile teams use decided to use a Test Driven Development [TDD] approach: a development style that takes advantage of a good unit test writing environment. A developer will initially write a unit test that defines a specific behavior that is desired, but the test will fail. Then the developer writes "just enough code" to make the test pass. TDD is a slow iterative approach, so the development of a small application-level feature might require dozens of unit tests. But when the feature is finished, the unit test suite is also finished, so those new unit tests can be added to the project-level testing database.

Test Driven Development can be a good way to develop new features that work "correctly," but the code in the initial version of a feature might be relatively ugly and confusing. So the last step in the TDD process is to do some code refactoring – to convert the working code into a better form for long-term maintenance.

If a development team intends to use TDD in middle of their agile development process, it will almost certainly be necessary to add more unit tests – to get the existing legacy code "under unit test control."

7. Summary: agile refactoring practices

In the agile development world, refactoring as a software development practice has gone much farther than the initial vision of "focused refactoring to support legacy code evolution." Traditional project might refactor periodically, especially when trying to revise software modules that have gone through many bug fixes and modifications to add new features.

But in agile refactoring, instead of a narrow focus on the preparation for a small set of code changes, agile refactoring practices are used frequently across the entire product code base – to keep all of the code clean and flexible. An agile development team may invest in refactoring work every week. And in the extreme case, agile teams using Test Driven Development will certainly be doing refactoring even more frequently – as part of every new code addition, multiple times per day.

Here is another way to think about the difference between the traditional approach to refactoring and the agile approach. Traditional software development views each module and each subsystem as an "asset" – most of the assets will have a long and stable lifetime but some of them will require periodic maintenance (to fix bugs and make simple extensions). Refactoring effort can be focused on a subset of modules that are most likely to be changed. But in agile software development, most

modules are not really fixed assets, they are ongoing "experiments" that will continue to evolve as new requirements evolve. The individual software components may not have a big intrinsic value, because they are smaller and they have been built more quickly. But the main value is the flexibility of the entire system – the knowledge that the development team has accumulated in building the early system and the skills of the team members to add new capabilities. For that reason, it pays to do frequent refactoring of the entire base to address code smell issues and keep the total software quality as high as possible.

But it is important to note that the total amount of refactoring will probably be similar in the traditional and agile styles. When working with legacy software with many changes and patches, a refactoring session may take days to restructure a large block of ugly code. Frequent agile refactoring based on code smells will help conscientious agile developers keep most of the code very clean, so most of the refactoring efforts will be short and agile. And the refactorings in a TDD environment are also relatively simple, and the presence of a good unit test suite helps to make the refactoring effort safe and painless.

8. Beyond code refactoring: Design-level restructuring

Refactoring is a useful code cleanup practice, but refactoring can also be used to clean up and improve the design structure of a software application. This concept of higher-level refactoring work was introduced in the book *Refactoring to Patterns* by Joshua Kerievsky. Joshua thinks that developers sometimes need to aim for more than just "clean" code. In many cases in a rapidly evolving software application, it becomes clear that the application is suffering from a rigid design structure. A good way to add more flexibility is to restructure the design to use one or more Design Patterns.

A Design Pattern is a recurring design solution to common problems. There are thousands of good design patterns that have been discovered and documented across the software industry, but the best-known patterns are the design ideas in the classic 1994 book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Joshua's book explains some of the simple refactoring techniques that can be applied to "refactor to" (or "refactor towards") a pattern that might improve a software design – making an application easier to extend without a lot of rework, *and* making the design structure clearer to future maintenance developers.

The Refactoring to Patterns book introduces some new code smells, as well as some design solutions for some of the standard code smells. Most of the target patterns require the developers to put key functionality into well-encapsulated classes, and the target pattern explains how to add some variant behavior in a new subclass or in a special class that "delegates" to the main design classes.

The refactoring details may be different for different programming languages, but the design approach may be the same. These design-level refactorings are particularly important in the early design and prototyping work. Developers may struggle to "get a small subset working," but they haven't thought enough about design alternatives and design extensions. Pattern-driven design refactoring is an opportunity to introduce a more substantial "architecture" into an early prototype – which may lead to important design insights as the application grows.

Design-level refactoring can also be used to extend the life of a mature development project. Many of the standard design patterns, such as Facade, Proxy, and Adapter, are good examples of "wrapping" legacy functionality in a reusable wrapper class. Many of the refactoring steps will be focused on building a stable interface and reduce coupling within the architecture.

9. Beyond code refactoring: Refactoring to understand

One last application of refactoring is in doing investigation of legacy code. Maintenance developers are often faced with a large code base that is difficult to understand: the internal structure might be unclear, confusing, or mysterious, the system documentation might be inadequate or missing, and the code comments might either be out of date or they might be outright lies. When a maintenance team takes over a large software system, the team needs to do some serious "software archaeology" – digging into the details of the software structure. Refactoring can be a powerful tool to probe the existing system's structure and behavior.

Exploratory refactoring is one of the topics in an excellent book on software reengineering: *Object-oriented Software Reengineering Patterns* by Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz. This book presents an outline of many code discovery techniques – including modeling, testing, code analysis tools, exploration of documents, and refactoring.

Exploratory refactoring is "designed to be thrown away" – so it doesn't require the same level of careful step-by-step code changing and retesting. But this kind of refactoring is usually very lightweight and focused. Adding new "probes" into the middle of existing functions is an easy way to collect intelligence on how a system works, and making minor structural changes to the code base can also be used to verify some initial hypotheses about how the system works internally.

All refactoring work – including agile refactoring and refactoring to patterns – is about more than just making the code cleaner. All refactoring work contributes to greater developer understanding. We all learn more about our code as we do restructuring, and in fact, we may learn about things we didn't really understand about our initial design.

10. The utility of refactoring tools

From the beginning, the pioneers of refactoring have looked for computer tools to give support for refactoring tasks. One reason: In the early days of refactoring, programming languages experts were working on productivity tools to reduce the effort coding work. At the time, everyone was working on syntax-directed program editors (text editors that are aware of programming language syntax, so they could prevent developers from typing in code with syntax errors) and code browsers (tools that build a database of programming primitives in a large code base – all of the functions, structure, attributes, database tables, and so on). The early philosophy of refactoring tools was, "We will make refactoring work easier by using technology to reduce the number of refactoring errors."

Many local refactorings (renaming local variables, splitting off compact small classes and functions from the existing code base) are relatively easy tasks to automate. For languages like Java, the standard integrated development environments all have very good basic refactoring capabilities. For other languages, such as C++ and Javascript, the set of automated refactoring toolkits is much more limited.

It is important to understand that "automated" refactoring is a long way from "automatic" refactoring. The refactoring process requires a lot of developer judgment – a developer can't just point a tool at the code and ask the tool to "make it better." Tools are good for executing individual refactoring steps, but when looking for the code smells or deciding on which modules to run an "exploratory refactoring," human judgment is still important.

One situation where it is especially useful to employ automated refactoring tools is in refactoring a code base without many unit tests. Many of the automated refactorings are pretty safe, which means that the need to retest after each refactoring step isn't so critical. Even so, it is still useful to recompile and run some tests on refactored code – don't trust the automated tool to be perfect. Also, there is some value in using automated refactoring tools to help add more unit tests. This can enable more manual refactoring later – and it contributes to a better level of understanding of the code and design.

11. Learning about refactoring

There are a lot of practical books about refactoring that illustrate the techniques with extended examples, outline some simple guidelines for finding code smells, and share useful ideas for writing clean code. It is good to become familiar with examples in books and online articles, just to learn how refactoring works.

But most software developers don't really understand refactoring until they start using refactoring on code they really care about. A good way to practice refactoring is to find some self-developed code, which could be some old product code no longer in use or some simple small self-contained software utility. Self-developed code is good because the developer will be familiar with the structure and style of the code. Older code is good because the developer will probably have forgotten some of the details of how the code works. The goal of an initial refactoring exercise is to learn something about how to make your own code clearer and easier to read.

Developers will improve their ability to refactor as they get practice, but they should also spend time understanding the theoretical background of understanding and improving legacy code. They should periodically study some of the books and articles about Code Smells, as a way to identify the types of code problems to attack in future refactoring efforts. In addition, every developer who plans to use refactoring will need to improve their skills at writing small unit tests. Unit tests are invaluable in the process of doing safe refactoring. The worst thing that can happen in a refactoring session: the code still runs, but there are hidden errors that the developers didn't detect. The lack of effective testing could cause even more code quality problems in the near future.

Last modified: Feb. 13, 2022