Refactoring: The Neglected Art of Agile Development

Robert Lee, Sr. Software Engineer
FamilySearch.org
leerw@familysearch.org

Refactoring is defined by Martin Fowler as “improving the design after it has been written.” But if code works already, why change it? This presentation covers the why’s, what’s, how’s, and when’s of refactoring.

WHY?

A correct perception of software development is required to understand refactoring. Software development is often mischaracterized as a sequential process that can be managed by an assembly line. A much better comparison is to an engineering process, which is full of discovery, reworking, and analysis; all before any production work is done. And what is the production process of software? It is the compiling process. Ours is a unique discipline where the product can be made over and over and over again incurring minimal cost. The code is not the product; the code is the engineering design. The working application is the product.

Too often software projects are completed late and over budget. And often the product is highly fragile, ready to break at any change. Another common occurrence is the code becomes “monolithic,” where all pieces are intertwined and not maintainable.

Because software is so easy to build, customers expect regular changes. Normally changes would not be a problem. If the code is flexible, then the change can be made, and the code can be quickly built. But when the code is not flexible, maintenance becomes a nightmare.

But how does an organization prevent their code from becoming fragile or monolithic (often termed “code rot”)? The answer is refactoring. Engineers know they cannot see all the answers from the beginning. That is why there is an engineering process, to find the problems before production. But where we build our product several times a day, we have an advantage over most other forms of engineering. We can engineer, build, experiment, build, test, and build again. We can use the working product regularly in our engineering process. We just need to know how to use that fact to our advantage.

It is impossible to write code right the first time, so don’t waste time trying. The common phrase is “make it work; then make it right.” Divide and conquer! Our job requires us to make code work. That working code can take the form of a first try, and ugly as it may be, any working code is valuable.

But the engineering process doesn’t stop with making the code work. We must make the code right. Otherwise the poor code will fester and propagate as others try to maintain it.

Why should we refactor? To avoid code rot. To keep technical debt low. To make maintenance easier. And to better serve our customers.
WHAT?
Martin Fowler calls refactoring “a process of changing a software system in such a way that it
does not alter the external behavior of the code yet improves its internal structure.” Basically,
refactoring is what gets you in trouble with your product manager. But we know our trade, and
we know what must be done. Refactoring is coding for the long run.

We must take care to consider our purpose in refactoring. Many people with CDO (that’s OCD,
Obsessive Compulsion Disorder, but alphabetized, the way it should be) gravitate to engineering
fields. Sometimes we want to make code right to our definition of right. But said behavior is
counterproductive. Primarily, it can lead to the unintentional introduction of defects into the
code. Further, it can really upset your coworkers. We need to decide before refactoring if the
change is syntactical, or pragmatic. Will the change make maintenance easier? If so, proceed.
If not, then we must resist.

A common problem that requires refactoring is unnecessary complexity. Functions that are too
long, classes that do too much, and code duplication are all examples of unnecessary complexity.
Keep the design and code simple. Avoid being clever. Use a straightforward approach to
solving problems. And further, please remove dead code. If we’ve killed the rat, we can’t leave
it on the table. We must throw it away. Venkat Subramaniam has a great quote, “It only takes a
few fingers to write some code, but it takes real courage to delete some.”

Software engineers should know their craft. Part of the craft of software development is
realizing that it changes constantly, and if we don’t change with it, we become a burden. Too
often engineers get in their own style, not recognizing that things can be much better. Software
engineers should know their craft. We must read and follow the books, articles, and conference
presentations, constantly improving our style.

WHEN?
When should we refactor? At two times, before making a change and after making a change.
We never refactor in the middle of a change. We complete the change, check it in to source
control, and then do the refactoring. Martin Fowler adds, “When you find you have to add a
feature to a program, and the program’s code is not structured in a convenient way to add the
feature, first refactor the program to make it easy to add the feature, [and] then add the feature.”

When should we not refactor? Again, just because we think we need to change the code, doesn’t
mean we should. Consider the cost and the impact of the change. And then get a second
opinion. Never soldier alone. Once it is clear the change is needed, then proceed.

HOW?
The first rule of refactoring is unit tests. We rely heavily on unit tests. The tests will tell us if
the code functionality has changed. If the code does not already have unit tests, and is not
testable, we consider loosing the code. Lost code is better than bad code. We can rewrite lost
code, but we have to maintain bad code.

We follow refactoring up with code reviews. But line-by-line walkthroughs are not effective.
Instead, we send an e-mail to another developer asking him or her to review the code at his or her
convenience. Then the reviewer can take quality time and understand the context of the change.
What do I tell my manager when I decide a refactoring needs to take place? Nothing! We know our craft, and we do not leave it up to managers to tell us how to do it.

We take baby steps with refactoring. We make small changes, test them, then check it into version control. We keep each step clear and concise through the process. We do not require a code review with every check-in. Version control is not about code reviews. Version control is about keeping track of code changes.

Code should have an evolving architecture. Again, we don’t know everything up front. But with regular refactoring and revisiting, we can keep a quality working product that can continue to be used in the future. Don’t rewrite every couple of years. Instead, refactor regularly!