Refactoring

Contents

Introduction to Refactoring

- 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.
- The word “refactoring” in modern programming grew out of the word “factoring” in structured programming, which referred to decomposing a program into its constituent parts as much as possible.

Reasons to Refactor

- Sometimes code degenerates under maintenance, and sometimes the code just wasn't very good in the first place.
- In either case, here are some warning signs that indicate where refactoring are needed:
  - Code is duplicated.
  - A routine is too long.
  - A loop is too long or too deeply nested.

Reasons to Refactor

- A parameter list has too many parameters.
- Changes require parallel modifications to multiple classes.
- Inheritance hierarchies have to be modified in parallel.
- Related data items that are used together are not organized into classes.
- A routine uses more features of another class than of its own class.

Reasons to Refactor

- A class doesn't do very much.
- A chain of routines passes tramp data.
- A middleman object isn't doing anything.
- A routine has a poor name.
- Data members are public.
- A subclass uses only a small percentage of its parents' routines.

Reasons to Refactor

- Comments are used to explain difficult code.
- Global variables are used.
- A program contains code that seems like it might be needed someday.

Specific Refactoring

- Some refactorings that are most useful in enhancing code quality are
  - Data level refactoring
  - Statement level refactoring
  - Routine level refactoring
  - Class implementation refactoring
  - Class interface refactoring
  - System level refactoring

Data-Level Refactoring

- ***Replace a magic number with a named constant*** If you're using a numeric or string literal like *3.14*, replace that literal with a named constant like *PI*.
- ***Rename a variable with a clearer or more informative name*** If a variable's name isn't clear, change it to a better name. The same advice applies to renaming constants, classes, and routines, of course.

Data-Level Refactoring

- ***Introduce an intermediate variable*** Assign an expression to an intermediate variable whose name summarizes the purpose of the expression.
- ***Convert a single multiuse variable to multiple single-use variables*** If a variable is used for more than one purpose, common culprits are *i*, *j*, *temp*, and *x*, create separate variables for each usage, each of which has a more specific name.

Data-Level Refactoring

- ***Change an array to an object*** If you're using an array in which different elements are different types, create an object that has a field for each former element of the array.

Statement-Level Refactoring

- ***Decompose a Boolean expression*** Simplify a Boolean expression by introducing well named intermediate variables that help document the meaning of the expression.
- ***Move a complex Boolean expression into a well-named Boolean function*** If the expression is complicated enough, this refactoring can improve readability. If the expression is used more than once, it eliminates the need for parallel modifications and reduces the chance of error in using the expression.

Statement-Level Refactoring

- ***Consolidate fragments that are duplicated within different parts of a conditional*** If you have the same lines of code repeated at the end of an *else* block that you have at the end of the *if* block, move those lines of code so that they occur after the entire *if then else* block.
- ***Use* break *or* return *instead of a loop control variable*** If you have a variable within a loop like *done* that's used to control the loop, use *break* or *return* to exit the loop instead.

Statement-Level Refactoring

- ***Return as soon as you know the answer instead of assigning a return value within nested* if-then-else *statements*** Code is often easiest to read and least error-prone if you exit a routine as soon as you know the return value. The alternative of setting a return value and then unwinding your way through a lot of logic can be harder to follow.

Routine-Level Refactoring

- ***Convert a long routine to a class*** If a routine is too long, sometimes turning it into a class and then further factoring the former routine into multiple routines will improve readability.
- ***Substitute a simple algorithm for a complex algorithm*** Replace a complicated algorithm with a simpler algorithm.

Routine-Level Refactoring

- ***Add a parameter*** If a routine needs more information from its caller, add a parameter so that that information can be provided.
- ***Remove a parameter*** If a routine no longer uses a parameter, remove it.

Routine-Level Refactoring

- ***Combine similar routines by parameterizing them*** Two similar routines might differ only with respect to a constant value that's used within the routine. Combine the routines into one routine, and pass in the value to be used as a parameter.
- ***Separate routines whose behavior depends on parameters passed in*** If a routine executes different code depending on the value of an input parameter, consider breaking the routine into separate routines that can be called separately, without passing in that particular input parameter.

Class Implementation Refactoring

- ***Pass a whole object rather than specific fields*** If you find yourself passing several values from the same object into a routine, consider changing the routine's interface so that it takes the whole object instead.
- ***Pass specific fields rather than a whole object*** If you find yourself creating an object just so that you can pass it to a routine, consider modifying the routine so that it takes specific fields rather than a whole object.

Class Interface Refactoring

- ***Move a routine to another class*** Create a new routine in the target class, and move the body of the routine from the source class into the target class. You can then call the new routine from the old routine.
- ***Convert one class to two*** If a class has two or more distinct areas of responsibility, break the class into multiple classes, each of which has a clearly defined responsibility.
- ***Eliminate a class*** If a class isn't doing much, move its code into other classes that are more cohesive and eliminate the class.

Class Interface Refactoring

- ***Hide a delegate*** Sometimes Class A calls Class B and Class C, when really Class A should call only Class B and Class B should call Class C. Ask yourself what the right abstraction is for A's interaction with B. If B should be responsible for calling C, have B call C.
- ***Remove a middleman*** If Class A calls Class B and Class B calls Class C, sometimes it works better to have Class A call Class C directly. The question of whether you should delegate to Class B depends on what will best maintain the integrity of Class B's interface.

Class Interface Refactoring

- ***Hide routines that are not intended to be used outside the class*** If the class interface would be more coherent without a routine, hide the routine.
- ***Encapsulate unused routines*** If you find yourself routinely using only a portion of a class's interface, create a new interface to the class that exposes only those necessary routines. Be sure that the new interface provides a coherent abstraction.
- ***Collapse a superclass and subclass if their implementations are very similar*** If the subclass doesn't provide much specialization, combine it into its superclass.

System-Level Refactoring

- ***Replace error codes with exceptions or vice versa*** Depending on your error-handling strategy, make sure the code is using the standard approach.

Refactoring Safely

- Refactoring is a powerful technique for improving code quality. Like all powerful tools, refactoring can cause more harm than good if misused. A few simple guidelines can prevent refactoring missteps.
  - Save the code you start with
  - Keep refactorings small
  - Do refactoring one at a time

Refactoring Safely

- Make a parking lot
- Make frequent check points
- Use your compiler warnings
- Retest
- Review the changes

Refactoring Strategies

- The number of refactorings that would be beneficial to any specific program is essentially infinite.
- The 80/20 rule applies on refactoring. Spend your time on the 20 percent of the refactorings that provide 80 percent of the benefit.

Refactoring Strategies

- Consider the following guidelines when deciding which refactorings are most important:
- ***Refactor when you add a routine*** When you add a routine, check whether related routines are well organized. If not, refactor them.
- ***Refactor when you add a class*** Adding a class often brings issues with existing code to the fore. Use this time as an opportunity to refactor other classes that are closely related to the class you're adding.

Refactoring Strategies

- ***Refactor when you fix a defect*** Use the understanding you gain from fixing a bug to improve other code that might be prone to similar defects.
- ***Target error-prone modules*** Some modules are more error-prone and brittle than others. Is there a section of code that you and everyone else on your team is afraid of? That's probably an error-prone module. Although most people's natural tendency is to avoid these challenging sections of code, targeting these sections for refactoring can be one of the more effective strategies

Refactoring Strategies

- ***Target high-complexity modules*** Another approach is to focus on modules that have the highest complexity ratings. One classic study found that program quality improved dramatically when maintenance programmers focused their improvement efforts on the modules that had the highest complexity (Henry and Kafura 1984).
- ***In a maintenance environment, improve the parts you touch*** Code that is never modified doesn't need to be refactored. But when you do touch a section of code, be sure you leave it better than you found it.

Key Points

- Program changes are a fact of life both during initial development and after initial release.
- Software can either improve or degrade as it's changed. The Cardinal Rule of Software Evolution is that internal quality should improve with code evolution.
- One key to success in refactoring is learning to pay attention to the numerous warning signs or smells that indicate a need to refactor.

Key Points

- Another key to refactoring success is learning numerous specific refactorings.
- A final key to success is having a strategy for refactoring safely. Some refactoring approaches are better than others.
- 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. Take advantage of these opportunities during development!

Readings

- **[Chapter 24]** Code Complete: A Practical Handbook of Software Construction by Steve McConnell, Microsoft Press; 2nd Edition (July 7, 2004). ISBN-10: 0735619670