Code Refactoring

"Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior."

Martin Fowler, www.refactoring.com

Code has a tendency to stray from clean design as it evolves through enhancements and fixes. This degradation increases maintenance cost and decreases performance and stability. In the past, the solution for degraded code has been "the next big revision", aimed at a redesign and rewrite of the system. Quite often, such projects have ended up being too ambitious, missing their deadlines and budgets, and resulting in systems with reduced functionality. Refactoring approaches the need for improving the code by spreading out a series of small changes over time. Each change preserves the system functionality, and can be verified immediately without having to wait for a major release and QA effort.

How do we find candidates for refactoring? Source checks point us to code that might be acceptable to compilers but are generally considered bad practice. Examples of these are type casts, arithmetic precision concerns, or missing default cases in switch statements.

Software metrics hint at areas that need attention. Control flow complexity like McCabe Cyclomatic tell us which functions, classes, or files are particularly difficult to understand. Coupling and cohesion metrics for object oriented systems provide insight into classes that have outgrown their clean designs. Evolved metrics like Halstead Program Difficulty combine several aspects to find complex code. Architectural metrics are particularly helpful when getting started. Violations of architectural hierarchies and the stability metric point to subsystems that need attention; architectural drill down lets us find the underlying code pieces that are causing concerns.

Introducing explicit typing to increase the readability and robustness of programs is another refactoring activity. Instead of using int for everything, a typedef or a struct containing just one member can be introduced used so that the variables that belong to specific units of measurement are clearly distinguished. To effectively insure that the type is introduced consistently, we need to see the variables and their interdependencies to other variables and parameters.

Another evolutionary downgrade of software results from copied code pieces. Over time, some of copies get changed for fixes but others are forgotten. A facility to recognize similar functions and code pieces can be very helpful to refactor the copies and bring them together.

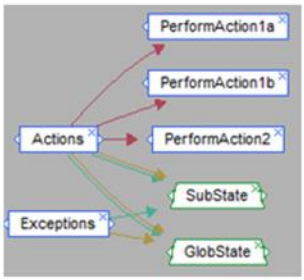
A typical refactoring exercise is to encapsulate data, restricting the number of global variables in the system. This exercise can start by applying simple metrics to review the number of reads and sets of global and static variables. Better would be graphs showing the variable and all the functions directly and indirectly using it. Especially when refactoring C code to C++ code, this view provides insights into whether this is a candidate for a class. More tool support is needed when pointers to these variables are involved; in these cases, data flow analysis is required to see the variable dependencies and actual uses when going through pointers.

Even when there is no intent to create classes, the refactoring of global variables is important as their use spreads over the lifetime of the program. Quite often, newly introduced variables are coupled to existing variables due to time pressures and concerns about unwanted side effects of changing the behavior of existing variables. To reign this in, refactoring needs to find coupled variables, all their uses and their interdependencies. Figure 1 shows a simple example of this refactoring

```
...
if (GlobState == 1) {
    if (SubState == 1) {
            PerformAction1a();
            SubState = 2;
    }else {
            PerformAction1b();
      GlobState++;
            SubState = 3;
    }
}else if (GlobState == 2) {
    PerformAction2();
    GlobState++;
}

...

if (GlobState > 2) {
    Globstate = 1;
    SubState = 1;
}
...
```




```
...
if (GlobState == 1) {
            PerformAction1a();
            GlobState = 2;
}else if (GlobState == 2) {
            PerformAction1b();
            GlobState = 3;
    }
}else if (GlobState == 3) {
    PerformAction2();
    GlobState = 4;
}


...

if (GlobState > 3) {
    Globstate = 1;
}
...
```

Figure 1. Refactoring coupled variables: mapping SubState into GlobState with help of complete view of uses

A variation of global variable refactoring is found in multi-tasking or multi-threading programs. Here, excessive use of shared variables leads to performance degradation due to the need to protect access in critical regions. Other concerns are increased risk of race conditions caused by missing protection, and deadlocks caused by the large number of critical regions. To refactor these, the coupling between variables needs to be identified. Distribution of variable access over the tasks comes into play as well.

To support refactoring campaigns, the tool Imagix 4D from the Californian company Imagix Corp. very helpful. The tool is also suitable to get a better understanding of your own and third-party code.

Further information is available at https://www.verifysoft.com/fr_imagix4d.html.

Translation of a blog post, source:
https://www.imagix.com/blog/coderefactoring/

Verifysoft Technology, www.verifysoft.com-2020