Project Report(CS3410)

Pranali Yawalkar(CS11B046)
Varun Gangal(CS11B038)
Shantanu Gupta(CS11B047)
Aritra Ghosh(CS11B062)
Aayush Agarwal(CS11B002)
IIT Madras

May 5, 2014

Contents

1 Acknowledgements

We would like to express our gratitude to Dr P. Sreenivasa Kumar for the useful comments, inputs and engagement through the learning process during the project. Additionally, we are thankful to Dr D .Janaki Ram for the "Principles of Software Engineering course" which introduced us to several good practices in designing scalable and maintainable software. We also thank the Teaching Assistants in general and particularly Manikantan S,Rajashekar V, Krishna Chaitanya T for their continuous guidance and support in our work all along the project.Their suggestions were extremely helpful in progressing in the project. We would also like to thank Sriram Kailasam and Tushar Sharma for introducing us to inFusion and cppDepends, one of which we finally ended up using as our primary code analysis tool. We are also thankful to all those who directly or indirectly helped and guided us in the project in any way. We are also thankful to the Neovim developer community for their whole hearted support.

2 Vim

Vim is a popular text editor, available on multiple platforms. Based on the vi editor common to Unix-like systems, Vim is designed for use both from a command line interface and as a standalone application in a graphical user interface.Initially Vim was released for Amiga ,it has since been developed to be cross-platform, supporting many other platforms. Its C89 source code (around 300,000 lines) has built up over almost 20 years, and has not been easy to maintain, in part because the project has just one maintainer.Adding patches has been difficult or even impossible at times.

Evolution of C standards C standards development has been a conservative process with great care taken to preserve the spirit of the original C language, and an emphasis on ratifying experiments in existing compilers rather than inventing new features. The first standard proposed was known commonly as C89 or C90.A very minor revision of C89, known as Amendment 1, AM1, or C93, was floated in 1993. It added more support for wide characters and Unicode.Revision of the C89 standard began in 1993. In 1999, ISO/IEC 9899 (generally known as C99) was adopted by ISO.It added a great many minor features. Perhaps the most significant one for most programmers is the C++-like ability to declare variables at any point in a block, rather than just at the beginning. Macros with a variable number of arguments were also added.C11 is an informal name for ISO/IEC 9899:2011,[1] the current standard for the C programming language. The standard has incorporated many changes like multi-threading support, alignment specification, bounds checking interfaces etc.[2]

3 Overview of the System

3.1 Neovim

Neovim is an attempt to refactor the codebase, to be able to add plugins and patches more easily, and to have a more decentralised maintenance structure. Neovim seeks to refactor vim source code with the the following aims in mind [3]:

- Simplify maintenance to improve the speed that bug fixes and features get merged.
- Enable the implementation of new/modern user interfaces without any modifications to the core source.
- Improve the extensibility power with a new plugin architecture based on coprocesses.

These form the broad aims of the project.There are many other aims which are not mentioned above.

3.2 Code Overview

The code is arranged in the form of a single directory named *src* which contains most of the core source modules. The modules are mostly of the two types

1. **Source files**
   These files are .c files. They contain most of the function definitions. Each .c file deals with a certain well-defined functionality of Vim

2. **Header files**
   These files are .h files. They contain most of the declarations.

Here is a brief description of the modules contained in the codebase.

File	Functionality
buffer.c	Deals with the manipulation of buffers.
diff.c	Deals with the diff mode
eval.c	Expression evaluation
fileio.c	Reading and writing files
fold.c	Deals with the folding functionality of vim
getchar.c	getting characters and key mapping
mark.c	Deal with marks
mbyte.c	Deals with multi byte character handling
buffer.c	Deals with the manipulation of buffers.
diff.c	Deals with the diff mode
eval.c	Expression evaluation
fileio.c	Reading and writing files
fold.c	Deals with the folding functionality of vim
mark.c	Deal with marks
menu.c	Menu related functionality
quickfix.c	Deals with the handling of quickfix commands (:make, :cn)
regexp.c	Deals with pattern matching machinery
spell.c	Spell checking
syntax.c	Syntax and other highlighting
window.c	Handling split windows

3.2.1 Important variables

The current mode is stored in State. The values it can have are NORMAL, INSERT, CMDLINE, and a few others.
The current window is curwin. The current buffer is curbuf. These point to structures with the cursor position in the window, option values, the file name, etc.
All the global variables are declared in globals.h.

3.2.2 The main loop

This is conveniently called mainloop(). It updates a few things and then calls normalcmd() to process a command. This returns when the command is finished.
The basic idea is that Vim waits for the user to type a character and processes it until another character is needed. Thus there are several places where Vim waits for a character to be typed. The vgetc() function is used for this. It also handles mapping.
Updating the screen is mostly postponed until a command or a sequence of commands has finished. The work is done by updatescreen(), which calls winupdate() for every window, which calls winline() for every line.

3.3 Domain

The Neovim codebase is very vast, since Neovim deals with the editing, searching and manipulation of text within files, execution of shell commands, storing undo history, managing its own clipboard, dealing with highlighting of text amongst other things. Apart from modules which deal with direct functionality, there are other modules which deal with managing system calls, managing the various buffers and swap areas used by Neovim. However, since the duration of the project was quite limited, we had to fix on a small, manageable subset of these modules in order to analyze and refactor them. We decided to work on text modules for the following reasons.

- One does not have to deal with system calls, and other OS specific services, since text modules deal with high level functionality such as searching and tagging.
- One can directly relate to the functionality, since it is visible to us as a user. We can directly map user actions to functions

3.3.1 The Undo Module

The undo tree
Vim maintains a tree of changes, rather than a stack as most editors do. Therefore, it is possible to navigate to versions of a file that would otherwise be lost if opened in other editors. For example, if there is the following sequence of changes [4]:

1. Write 'one'.
2. Add the word 'two' to make it 'one two'.
3. Remove the word 'two' to get back to 'one'.
4. Add the word 'too' to make it 'one too'.

In most editors, it is impossible to get to the 'one two' stage now. But because of the undo tree, `vim` makes it possible to navigate to it.

3.3.2 The Tags Module

When editing programs, there is often a need to jump to another location, for example, to see how a function is defined. To help, Vim uses a tags file that lists each word you are likely to want, and their locations (file path and line number). Each wanted word is known as a "tag", for example, each function name or global variable may be a tag. The tags file has to be created by a utility (such as ctags), and has to be updated after significant editing has occurred. Vim maintains a tagstack , with the locations and tags to which the user has jumped. Jumping to a tagname using the $:tag < tagname >$ command takes the cursor to the declaration of the identifier

4 Previous work

Neovim has a growing set of developers who have been working on a number of issues.A list of issues are available here[3].Each of the issues are flagged as

- Backlog
- Done
- Ready
- In Progress

Each of the issues are discussed in a separate thread. Some of the issues that have been solved are Removing dead code, clean up some list append functions, handling null bytes in strings and various others. We also write some of the other things that have been done till now.[3]

- Cleaned up source tree, leaving only core files
- Removed support for legacy systems and moved to C99
- Removed tons of FEAT* macros with unifdef
- Reduced C code from 300k lines to 170k
- Enabled modern compiler features and optimizations
- Formatted entire source with uncrustify

- Replaced autotools build system with CMake
- Implemented continuous integration and test coverage
- Wrote 100+ new unit tests
- Split large, monolithic files (misc1.c) into logical units (path.c, indent.c, garray.c, keymap.c, ...)
- Implemented job control ("async")
- Reworked out-of-memory handling resulting in greatly simplified control flow
- Merged 50+ upstream patches (nearly caught up with upstream)
- Removed 8.3 filename support
- Changed to portable format specifiers (first step towards building on Windows)

5 Design aspects

One issue relates one of the major goals of writing maintainable code, which is eliminating unexpected side effects. Global state variables are the reason that these effects occur (they introduce common coupling), and therefore, minimizing their use helps a lot in being able to understand the program through understanding each part of the program in isolation. The plan is to collect all global variables (around 1000 in number) into a struct holding the editor state, which can be passed to functions that need the global state.
The vim codebase appears to make use of a large number of goto statements.Since goto statements are both archaic and they complicate the flow, we felt that it was better to factor out goto statements wherever possible during refactoring. In his seminal paper - " Goto Statement Considered Harmful" , Djikstra proposed that the goto statement must be abolished from any structured programming construct [1].
It is generally considered a good rule of thumb to have a function fit into the size of a window. Long functions often have a tendency to deter maintainabilty and readability. Hence, a common aspect of any refactoring process is to make a conscious attempt to break large functions into smaller, more comprehensible parts.

6 Modifications

Inorder to make the above changes we made use of some tools to analyse the code. Some of them are summarised below:

- **Doxygen(to generate the call graphs:)**
- **Generating Control Flow Graphs**
- **Infusion**
- **ctags**
- **Undo and Gundo Vim**

Call Graph

A call graph is a directed graph that represents calling relationships between functions in a computer program. Specifically, each node represents a procedure and each edge (f, g) indicates that procedure f calls procedure g. A cycle in the call graph indicates recursive function calls.
The call graph of a function f gives the subgraph which is rooted at f. For instance, consider the following piece of code.

```
int f()
{
    g();
    k();
}
int k()
{

}
int g()
{
    h();
}
```

The call graph for the above example will be as follows

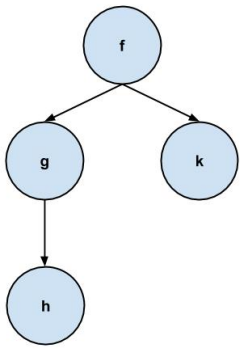

Figure 1: Example call graph

Caller Graph

A caller graph is a directed graph such that each edge (f, g) indicates that procedure f is called by procedure g. The caller graph of a function f gives the subgraph such that any vertex v in it has a path to f. For instance, consider the following piece of code

```
int f()
{
    g();
}

int h()
{
    f();
}

int k()
{
    g();
}
```

The caller graph for the above example will be as follows

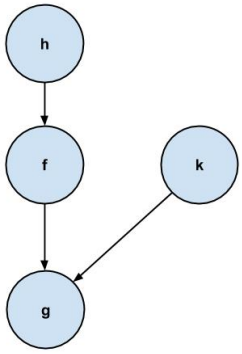

Figure 2: Example caller graph

Importance of call graphs

Call graphs play an essential role in understanding any codebase. They are specifically important in the context of refactoring since refactoring requires a exhaustive overview of the inherent structure of a codebase. The call graph of a function gives us the role of the function in the larger context of the module which contains it.

Using call graphs for function splitting

One of the major forms of refactoring we attempted carrying out was to split excessively large/noncohesive functions into smaller, more cohesive functions. Though the call graph does not give us any idea about the cohesion of the code, it helps us appreciate the dependency relationship between functions. Splitting a function can be thought of as deleting edges to the function being split, removing the corresponding node and adding new nodes for the split functions, and adding these edges. Thus, splitting a function f may be visualized as rewiring the neighbourhood of f in the call graph. From the

point of view of function splitting, the caller graphs and callee graphs of a function f are especially important.

- The callee graph gives information about the entire set of functions which are called by f. In other words, it is a union of all possible call trees. Callee graphs have limited utility for function refactoring when the function under consideration has a high nesting level. For functions with low nesting level(say 1 or 2) , however generating the callee graph would be overkill, since one would rather follow the dependencies from the syntax

- The caller graph gives information about every function whose call tree may eventually contain f. The caller graph is generally more useful for function refactoring than the callee graph, since most codebases do not have much nesting level, but there are several points from which a function is called. When we refactor a function, we might change the interface for the function. In that case, we need to modify the various calls made to the concerned function. Even if we do not change the interface, the semantic interpretation of the function requires us to appreciate the context from which it is called.

Doxygen

Doxygen is the de facto standard tool for generating documentation from annotated C sources. The capabilities offered by Doxygen are as follows:-

- It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.

- One can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find one's way in large source distributions. Doxygen can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

- One can also use doxygen for creating normal documentation.

Generating Call Graphs using Doxygen

We use Doxygen in order to generate call graphs as well as callee graphs. For instance, consider the dotag function in the file tag.c

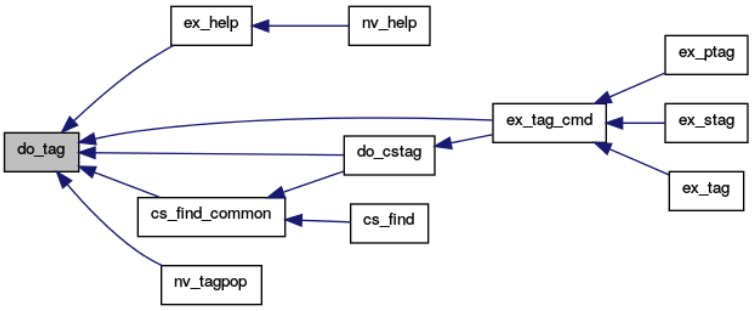

Figure 3: Caller graph for dotag.c

Generating Control Flow Graphs

Running gcc with the fdump-tree-all-blocks.c flag on, gives us a .cfg file. This file represents the control flow graph of the file compiled. We then convert the .cfg file to a .dot file using a bash script we write. We then use graphviz in order to render the generated .dot file as a .png file. For instance, consider the following piece of code

```
#include<stdio.h>
int check_prime(int num);
int main(){
    int n1,n2,i,flag;
    printf("Enter two numbers(intervals): ");
    scanf("%d %d",&n1, &n2);
    printf("Prime numbers between %d and %d are: ", n1, n2);
    for(i=n1+1;i<n2;++i)
    {
        flag=check_prime(i);
        if(flag==0)
            printf("%d ",i);
    }
    return 0;
}
```

```
int check_prime(int num) /* User-defined function to check prime number
{
    int j,flag=0;
    for(j=2;j<=num/2;++j)
    {
        if(num \mod j==0){
            flag=1;
            break;
        }
    }
    return flag;
}
```

The generated control flow graph is as follows. Every node of the graph is a basic block.

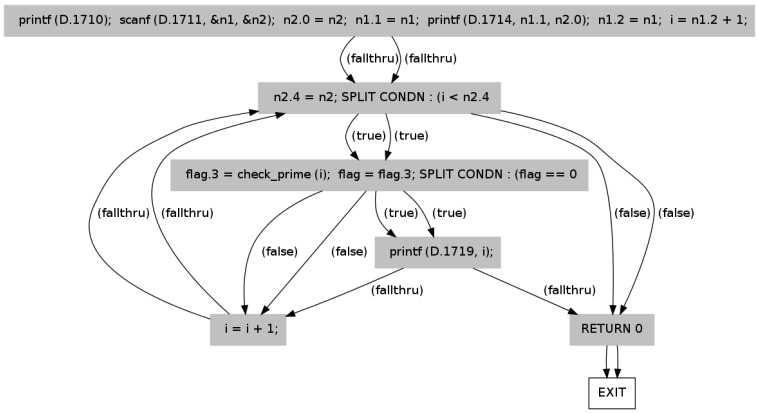

Figure 4: Control flow graph

InFusion

$InFusion^{TM}$ is a commercial tool for code analysis, which claims to perform indepth design assessment. We used a temporary 20 day licence that the software provides. We used the following features of $InFusion^{T}M$ for analyzing the codebase

- Finding the Cohesion Deficit at both Module and Function Level
- Detecting blob modules and blob operations

- Qualitative interpretation of function level cohesion

Ctags

Ctags is a programming tool that generates an index (or tag) file of names found in source and header files of various programming languages. Depending on the language, functions, variables, class members, macros and so on may be indexed. These tags allow definitions to be quickly and easily located by a text editor or other utility. When used in combination with tags in vim, ctags makes a powerful combination capable of exploring an unknown codebase.

Undo and Gundo.vim

In order to better appreciate the undo functionality of vim, we decided to use the gundo.vim plugin, which allows us to navigate the undo tree maintained by vim using a GUI like navigation over it.

Changes in the code

We broadly made three types of changes. Each of them has been shown with some examples of the code:

- **Strategic grouping of Global variables:**The motivation of this was explained above. Basic idea follows from here :

```
struct G {
int g_a ;   // was a global earlier.
int g_b ;
} vars = { 1 , 2 };

G * const_g ;
void f (G     const pointer ) {
//old use :
// x = fprime(ga) ;

//new use :
x = fprime( pointer >ga);
}
```

- **Removing gotos:**
  The vim codebase appears to make use of a large number of goto statements. Since goto statements are both archaic and they complicate the flow, we felt that it was better to factor out goto statements wherever possible during refac toring.For instance, we consider the following piece of code from the tag.c file of neovim.

```
if((tagstack[tagstackidx].tagname =vimstrsave(tag)) ==NULL)
{
curwin >wtagstacklen = tagstacklen  1 ;
goto enddotag;
}
```

  As we discussed above, such a piece of code presents issues from the point of view of understandability and maintainability. The issues that arise are briefly discussed below:-

  - The control flow cannot be inferred directly by putting the text in context. We do not know whether end do tag exits the function, or returns back to some earlier point in the function
  - The user would need to scroll down or search for the section labelled enddotag in order to understand this code. This would lead to significant time expenditure, and loss of attention on the part of the programmer to alternate between the two pieces of code when understanding them. This would also pose additional challenges as it may lead to mistakes when adding to the code.

  Thus we make cleaner functions instead of the goto like shown below. The only change that would be required is to change the goto to function call instead of the jump.Each function that we discussed above had an end section, from which there were goto jumps from various points in the code. In most cases, this piece of code had a behaviour equivalent to garbage collection methods or destructors in modern terminology. Instead of jumping to end do tag in the function body, we can instead call a function which contains the code of the end do tag section.We have labelled this function as do _tag _cleaner.

```
int do_tag_cleaner(int *use_tagstack_ptr,int *tagstackidx_ptr,
int *jumped_to_tag_ptr)
{
```

```
    if(*use_tagstack_ptr &&*tagstackidx_ptr<=
    curwin->w_tagstacklen)
      curwin->w_tagstackidx =*tagstackidx_ptr;
    postponed_split = 0; /* don't split next time */

    return *jumped_to_tag_ptr;
}
```

This makes it easier to understand the code and also add to the code.This change was done at several places in the module tag.c

- **General refactoring of functions:**This was done for functions which were very long. It is safe to assume that a function more than 500 lines of code would not perform a single operation.This was seen in the function do _tags of tag.c. The function was seen to have logical cohesion. Thus it was split to several functions both to improve the readability and maintainability.We should also note that removing logical cohesion introduces control coupling.The changes corresponding to these are not shown here since they are too spreadout.

7 Results

We have tested the changes made on the unit tests given as a part of the Neovim source code. **All the tests pass on making the changes**. We check on correctness for the refactoring changes in the steps briefly described below

- We build the project/codebase using the makefile provided. An error at this stage indicates that there is a compilation error in one of the source files touched.
- We then run all the tests in the Neovim test suite at once. If any of the tests fails, then a runtime error is thrown. If not, we get list of all the tests which pass successfully
- In addition to this, we also run neovim and try running the functionality for which we had refactored the concerned modules. For instance, we test the tags functionality of neovim if we refactor anything in the tag.c module.

All the changes mentioned below have cleared the three stage process mentioned above successfully.

7.1 Refactoring do _tags

On initial inspection, the file tag.c was classified as a blob module by $InFusion^{TM}$. This was because it contained two functions - namely do _tags and find _tags which were classified as blob operations. We refactored out the do _tags function, which had a high degree of logical cohesion and very large LOC in to smaller functions, which together exhibited equivalent functionality.
On inspecting this modified codebase, we find that the do _tags module is not classified as a blob operation anymore. Also, there is a decrease in the cohesion deficit of the module tag.c as a whole. We see that cohesion deficit is negligible for most of the functions that have been factored out of the former do _tags. Also, the qualitative remarks on the functions have in general improved greatly. This indicates that the refactoring carried out in this case was largely successful.

Functionname	Lines of Code	Cohesion Deficit	Data Access	Operation Calls
dotag	374	Negligible	Tight	Tight
function1	4	Negligible	Weak	Weak
function2	9	Negligible	Weak	Weak
function3	47	Negligible	Weak	Weak
function4	50	Negligible	Average	Average
function5	31	Negligible	Average	Weak
function6	30	Negligible	Weak	Weak
function7	155	Negligible	Tight	Average
function8	139	Negligible	Tight	Average

The average lines of code for functions which were formerly a part of dotags is now reduced to 93.22, which is much less compared to the single function present originally, which was over 500 lines long.

7.2 Strategic Grouping of Global Variables

Since this change was primarily aimed at enhancing the readability of the code, it is difficult to provide quantitative metrics in order to validate the same. However, on running the refactored code with global variables grouped into structures, we observe that none of the essential metrics such as Complexity, Encapsulation , Cohesion or Coupling are affected. This indicates that the grouping of global variables into structures has not had an adverse impact on the metrics of the codebase, though it has enhanced readability. This validates our claim that the change is a positive change.

7.3 Adding Cleaner Methods instead of Goto

This change was also aimed at enhancing readability, and making the code more modular.

7.4 Refactoring undo.c

The saved lines in undo.c are stored as a list of lists, one for each buffer. Each change is stored as a list of elements, with one element corresponding to a text block under change. Multiple redo branches are stored as a linked list (using the `uh_alt` fields), the head of which is stored in the parent change (the undo higher in the tree).

7.5 Function(s) under consideration

7.5.1 undo _time

This function goes back and forth across the timeline in response to commands like `earlier f`. We chose to refactor this function firstly because of its length (nearly 300 lines with comments), and also because of its somewhat roundabout code, which provides opportunities for refactoring.

The function takes 4 arguments:

- `step` : the number of steps to take in the undo tree

The following arguments are boolean flags to convey what units to take steps in.

- `sec` : step back/forward in time.
- `file` : in terms of file writes
- `absolute` : each change also has an associated number, which can be referred to directly.

At a higher level, the function does 3 things:

1. Find the undo sequence number of the change to navigate to.
2. To find a path to the change found above, by marking the nodes to go through.
3. To traverse the path formed above.

Therefore, we propose to split the function into 3 parts, each of which does one of the jobs mentioned above. This is expected to improve readability, and cohesion, at the cost of introducing some tighter coupling.

Another problem with the code is the use of a two-iteration loop [3] to reuse some code in the second step. This style of writing code has been labeled an *anti-pattern* in the Neovim issues page, and other functions containing such code are also being worked on by other volunteers. We also wish to get rid of this loop from the function, and replace it with something cleaner.

8 Software Principles

When one normally writes a piece of code, say an elementary program that implements Djikstra's algorithm, one scarcely cares the extent to which the code is extensible to other shortest path algorithms later, its performance in more generalized settings etc. The emphasis happens to rest on getting the job done, while ensuring the code can be understood enough to debug it.
However when writing a large scale software system, this assumptions get thrown out of the window. The reason is that there are several long term considerations that arise, which we have to consider when we are coding up any component that is to be added to such a system. The reasons for this are as follows

- The system may be maintained across several generations of developers. The code one writes now should be understandable and extensible a decade or two later.
- In addition, the code must be scalable, since the system may be deployed at several different places across a variety of settings. Even if one component of a software system is not scalable, it will pull down the performance of the entire system
- The code must be easily documented, and the interfaces must be clear.
- All developers have to adhere to some common coding style and standards, which have to be decided beforehand for the system

Adherence to software principles become necessary in any large project, since it is difficult to enforce invariants which make the code maintainable and scalable otherwise. A bad design pattern or coding practice may propagate through the system and bring down its performance severely (as we will see in the case of "anti-patterns" below).

Regression Testing

- Regression testing is the process of testing changes to computer programs to make sure that the older programming still works with the new changes.

- The intent of regression testing is to ensure that a change such as those mentioned above has not introduced new faults. One of the main reasons for regression testing is to determine whether a change in one part of the software affects other parts of the software.
- One of the common methods of regression testing include rerunning previously completed tests.
- The Neovim package comes with a suite of tests.By seeing if any of the tests break after making a change, we can infer the correctness of the change to some extent.
- Another possible way of checking whether the changes are valid is to test the functionality of the module which has been changed. For instance, when we make changes to the tags module(tags.c) in Neovim, we inspected the tags functionality.

Anti Patterns

An anti-pattern (or antipattern) is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive. It can be thought of in some sense as a bad pattern. Since it is adopted widely, it can proliferate through a large scale software system , severely damaging its performance on many software metrics. As a consequence, the codebase becomes less scalable and maintainable. Antipatterns can also pop up in a large scale software codebase as a result of changing standards and added features. As languages and frameworks mature, they add more sophisticated programming constructs. Also, newer and more efficient patterns for the language emerge. As a result, what was an accepted/good solution some years earlier may now be a bad solution with much more efficient alternatives available. The authors of [7], who first proposed the term, said there must be atleast two elements present to formally distinguish an actual anti-pattern from a simple bad habit, and practice, or bad idea.

- A commonly used process, structure or pattern of action that despite initially appearing to be an appropriate and effective response to a problem, typically has more bad consequences than beneficial results.
- A good alternative solution exists that is documented, repeatable and proven to be effective.

Examples

We shall elaborate in this section on some anti patterns that we observed in the Neovim/Vim codebase. Some of them are as follows

- Blob Operations and Blob Modules
- Two Iteration Loop
- Spaghetti Code

Blob Operation

A Blob Operation is a very large and complex operation, which tends to centralize too much of the functionality of a class or module. Such an operation usually starts normal and grows over time until it gets out of control. The presence of such an operation hampers readability and maintainability.

Detecting Blob Operations

We use $InFusion^{TM}$ in order to detect blob operations. Detecting blob operations is also used in the blob module detection. The metrics used for detecting a blob operation are as follows

- The number of lines of code is very high
- The operation has many conditional branches.
- The method has deep nesting.
- The method uses deep variables.
- There are many comment lines inside the operation body

Blob Module

The Blob Module design flaw is a common design flaw observed in large codebases, especially those which are procedural. It designates a module that has a large size and a high complexity. These characteristics make the module difficult to comprehend for a developer who is new to the codebase. They also have a negative impact on the maintainability of the affected module, and by extension the entire codebase. A blob module has a tendency to have

low cohesion since it is difficult to maintain cohesion as the number of lines of code shoots up. The metrics which are affected adversely by the presence of a blob module are as follows:-

- Complexity
- Coupling
- Cohesion

Detecting Blob Modules

We use $InFusion^{TM}$ in order to detect blob modules in the codebase analyzed. InFusion detects blob modules by checking for five threshold conditions. A module is classified as a *blob module* if and only if these conditions are satisified. The following metrics are severely affected by the presence of a blob module.

1. The module contains two or more blob operations
2. The total size of functions in the module is very high. In other words, the total lines of code, or the average lines of code are very high
3. The functional complexity of the module is very high.
4. The cohesion of the module is low.

Blob Modules in Neovim

We consider first consider the tags module in the neovim codebase. This module is found to contain two blob operations, namely the do_tag and the find _tags functions. The module is itself found to contain high cyclomatic complexity and nesting level.

Two Iteration Loop

The two iteration loop pattern attempts to reuse code by having a two iteration loop corresponding to the if and $else$ parts of an $ifelse$ construct. The original motivation for writing such code may have been code reuse. However, using static functions is a much cleaner and less tricky way of reusing blocks of code.

```
/*
 * attempt == 0: try match with '<', match at start of word
 * attempt == 1: try match without '<', match anywhere
 */
for (attempt = 0; attempt <= 1; ++attempt) {
  if (attempt > 0 && patc == pat)
    break; /* there was no anchor, no need to try again */

  //...

  /*
   * round == 1: Count the matches.
   * round == 2: Build the array to keep the matches.
   */
  for (round = 1; round <= 2; ++round) {
    // ...
```

Spaghetti Code

Spaghetti Code appears as a program or system that contains very little software structure. Coding and progressive extensions compromise the software structure to such an extent that the structure lacks clarity, even to the original developer. One common form in which spaghetti code is often found in procedural programs, especially dated ones, is in the form of goto statements which themselves contain goto statements that jump back into the code. A code snippet which would qualify as spaghetti code is as follows

```
{
    L2:
        x = 3;
        x+= y;
        goto L1;

    L1:
        goto L2;

}
```

9 Learning

A project helped us a lot in learning a lot about Open source Software. Some of the salient aspects are described below

- **Cohesion vs Coupling Tradeoff**
  Ideally, one would like to achieve a high cohesion and a low coupling. However, one can achieve this only to a certain degree, since cohesion and coupling happen to be inversely related (meaning that increasing the cohesion leads to increase in coupling). Beyond a certain point, decoupling any component(whether a function or a file) leads to a decrease in its cohesion. This is also known as the Cohesion vs Coupling tradeoff.

- **Importance of Interaction with the Developer Community**
  We started the project with the intent of working with Vim. We made this decision since some of us are vim users, and we found that touching the vim codebase would help us gain better appreciation of vim functionality, while gaining a better insight of how vim works.
  We found that every version of vim carries with it a todo list/ wishlist of features that have to be added to it. We selected some of these features as potential candidates for the final project, and then decided to contact the developer community.
  Most open source communities have their own IRC channel. We first went on to the vim IRC channel. However, on conversing with some of the developers, we were informed that the vim codebase is very archaic and that it would be very difficult for a novice to make any modifications to the same. They asked to us to look at Neovim, which is a modern effort at refactoring Neovim.
  We then went onto the neovim IRC channel. However, since we did not get any response from developers on the channel, we asked for guidance from some of the developers on the vim channel(since some of them would also be neovim developers). They asked us to get onto the neovim mailing lists. It was here that we finally managed to get a response from Thiago Arruda, a developer from Brazil who is the founder of the Neovim project.

- **Organization of Open Source Software**

Free and Open Source Softwares(FOSS) have a different model of development from what one would expect in a traditional classroom setup. The following components are usually always associated with an open source project.

- **Repository**
  The entire codebase is usually hosted on a repository. The repositories are mostly located on standard repository hosting service such as GitHub or Mercurial. The repository helps in maintaining the entire codebase in a version control setup.
- **Issues Pages**
  Every project usually has an issues/bugs page associated with it. This page contains a log of all the issues raised, the responses by developers, the acceptance or rejection of an issue by the community, the assigning of the issue to a developer/group of developers, discussion of problems related to the same. An issue is finally closed when it reaches some logical conclusion, which may be some of the below

  1. Accepted and solved by some developer. The relevant patch is commited and then merged by the veteran developers who have merging rights on the repository
  2. The issue is rejected. This may be either because it is invalid or is not of priority to the aims of the developer community as of now
  3. The issue is assigned to some developer, but it is later found that it is not possible to solve it as of now, and it is closed or left open for anyone who has a valid solution.

- **Tools used**
  The project helped us appreciate the importance of being acquainted with various tools. We also appreciated the role of docpages, blogs and forums, which allow any developer to quickly get a tool up and running in a manageable amount of time without facing significant hassles. The tools that we came across can broadly fall into two categories

  - **Analysis**

These tools were used by us to analyze the structure of the code, and also to evaluate the goodness of the codebase both qualitatively and quantitatively. The tools we came across were

  * Doxygen
  * Cscouts
  * InFusion
  * CodeSurfer

- **Coding**
  One needs to be familiar with a plethora of tools in order to perform programming efficiently. One need s to have a good hang of the associated plugins, command line utilities etc. The plugins and the utilities we came across were as follows

  * Ctags
  * Folding Functionality of Vim

- **Team Work:** Through this project we realized the potential of synchronizing efforts of each member and combining them in a meaningful way. We learnt how to maintain the code on Github and commit and push changes introduced without leading to any inconsistencies.Since each one did a certain part of the work and all of our work was interrelated, it was necessary for each one to understand each other's work and coordinate efficiently.

- **The difference code size makes:** Neovim has around 200K lines of code. The huge code size necessitates a different way of looking at code. We had to choose a specific part of the codebase to understand, leaving us with next to no knowledge about how the rest of it operates, as of now.

- **Builds:**Building a large-scale software requires specialised build configuration. Neovim uses CMake, which handles builds across platforms.

10 Hurdles

As we learnt a lot in the project we also faced some impediments in the process.We eventually overcame some of them and managed to circumvent others in the process:

- **Code Analysis tools**
  Some of the tools posed problems in terms of using them for our project. Though we were able to get them running for small programs, we were unable to get them running for the entire project. Some instances of such tools were
  - Cscouts
  - CodeSurfer
  - CppDepends
- **Understanding the semantics**:Many of the modules require a clear understanding of the exact functionality and context, the changes cannot be automated.Understanding the context requires a careful study of the module.
- **Documentation unclear for some of the tools:**Since some of the tools are quite recent and still being improved, the amount of documentation available is less.Running the tools in Linux also holds throws in a greater challenge in terms of some of the settings to be made.

11 Conclusion

There is an extensive scope for refactoring the vim codebase. Since the codebase was written both in an older standard of C, there are many archaic features and constructs of C that it employs. Hence, there are many changes waiting to be made in terms of migration to the newer C standard. Apart from this, the design methodology and patterns employed are also those which were around a decade or two back. Better solutions and newer patterns for some of these are now around and the older patterns found in the codebase may now even be classified as antipatterns. There is wide potential for the detection, removal and replacement of these antipatterns. This direction of investigation holds special promise from the point of view of refactoring the codebase.

12 Appendix

Using Vim

Vim is a powerful text editor with a wide range of features. Throughout this document, we have referred to these features quite liberally, without providing explanation on the nature of the feature. This section attempts to address any confusion the user may have in that context

Installing Vim

If you are working on a Linux machine, there is a good chance that you may have the vi editor already installed. Vim can be installed using the apt-get commmand like any other package. In case vim is not available on your local repository, you can clone the Vim project which is hosted on Mercurial.

Vimtutor

Vim is a powerful editor that has many commands. Vimtutor is a tutorial which offers a 25-30 minute long tutorial to get oneself acquainted with basic Vim functionality. For a new user of vim, this tutorial is a good starting point. For the sake of illustration, we will demonstrate vimtutor briefly. One thing that one must keep in mind while using vimtutor is that it is necessary to actually type out the instructions specified in order to get a feel of Vim.

- Type in vimtutor into your terminal, in order to fire up the vimtutor tutorial.
- Use the h and l keys to move left and right respectively. The k and j keys are used to move up and down. Note that you may also use the normal cursor keys to navigate, the shortcuts are just recommended for faster navigation. This finishes the first lesson
- Follow the remaining lessons similarly. They deal with deletion commands in vim, an introduction to dealing with modes etc.

References

[1] Edgar W Djikstra *Goto Statement Considered Harmful*

[2] Eric Steven Raymond *The Art of Unix Programming* Chapter-17 Portability

[3] *Neovim Issues page*

[4] The vim user help file : *husr*32.*txt*

[5] Erich Gamma et al *Design Patterns (The GoF book)*

[6] http://sjl.bitbucket.org/gundo.vim *The Gundo home page*

[7] Brown, Malveau et al *Refactoring Software, Architecture and Projects in Crisis*