Vim Text editor inside and out

Cleo Kesidis
260448644
cleoniki.kesidis@mail.mcgill.ca

Mangala gowri Krishnamoorthy
260511307
mangala.krishnamoorthy@mail.mcgill.ca

Kamran Naghiyev
260362012
kamran.nagiyev@mail.mcgill.ca

1. INTRODUCTION

Vim is a text editor initially developed to imitate and improve vi, a standard editor in UNIX systems. ("Vim" initially meant "Vi IMitation", but in 1992 the name was changed to "Vi IMprovement" [6].) Vim was first developed in 1988 on Amiga by Bram Moolenaar, and first released in 1991 [6]. It's an open source project and a charityware; Moolenaar helped to establish a charity called ICCF Holland which supports a children's centre in Uganda and to which he encourages users of Vim to donate [9].

The original version of Vim, 1.14, was simply an extension of vi for use on Amiga. In 1992, version 1.22 was released, containing new features and a UNIX port. Version 3.0, which allowed for multiple buffers, came out in 1994; version 4.0 in 1996 with a graphical user interface (previously Vim had only been used from terminals); version 5.0 with syntax highlighting in 1998; and version 6.0 came out in 2001 with vertical window splits and a plug-in system for script loading. The most recent major update to Vim, version 7, was completed in 2006 and included spellchecking, auto-completion, and undo branches. In 2010, a minor update (7.3) came out including persistent undo features and support for Python 3. [6]

One of the main design goals of Vim was to keep it vi compatible while improving on vi by adding new features. Vim was designed to be portable and maintainable, with a flexible interface and well-documented features to make it easily customizable and understandable by users [5]. Additionally, Vim was built to be fast and small in size, with a quick start-up time and minimal communications overhead [5]. The developers specifically chose to have a simple graphical user interface because they considered consistency over many platforms to be more important than fancy graphics [5]. Some of Vim's features which make it still appealing to users today are its ability to have multiple documents open at once (or multiple views of the same document), its system of registers that allow users to "copy" a piece of text to a certain register and retrieve it later (perhaps after copying other pieces of text, or after quitting and re-entering the editor), its powerful search and replace abilities, and its extensibility [6].

Starting from the 7th version, Vim includes a spell check. When spell checking was going to be added to Vim a survey was done over the available spell checking libraries and programs. Unfortunately, the result was that none of them provided sufficient capabilities to be used as the spell checking engine in Vim, for various reasons such as unsupported encodings of the documents and most of all due to unavoidable slowdown in performance. Thus it was decided to develop a new compatible spell check for the Vim that also supported suggestions for the misspelled words. To avoid overloading of the memory by the suggestion mechanism the spell check was designed such way that the users with sufficient memory could get very good suggestions while a user who is short of memory could just use spell check that did not consume a lot of memory. [5]

One of the biggest design decisions for Vim was its extensibility and customizability. Users of Vim – many of whom are programmers – should be able to easily write scripts to make Vim become whatever they want it to be. Every developer can submit their contributions if they think that it could be useful to other users. However like in any other software project lots of developers working on the same project can turn the code into a mess. To prevent this every developer is encouraged to stick up to the Vim's coding style and document their contributions. [5] Vim is also customizable in its interface, its ability to display files in many different views [6], its many different syntax highlighters, its ability to create new commands and shortcuts, and the fact that it allows users to disable certain features [5]. Currently, Vim is still being actively developed. Moolenaar remains the chief maintainer of the project [6].

2. EXISTING CONTEST

Vim's architecture is described most frequently as model-view-controller or repository style. Model-view-controller is the reference architecture for graphical editors. In these situations, the model is a data structure representing the document being edited. (In the case of Vim, this is called the buffer.) The view is the section of the program that shows the model to the user and updates itself. The controller causes actions to happen based on the user's command. As in all model-view-controller architectures, this means that the controller and the view subsystems both depend on the model and each other, and the model does not depend on either of them. [1, 8]

Vim's architecture is described by other authors [2, 3, 7] as a repository style system. Their argument is that all of Vim's subsystems and actions revolve around a central data structure, the buffers which represent the files being edited. The data structures implementing these buffers are all global variables accessed and used by essentially every part of Vim. Figure 1 shows one author's interpretation of Vim as a repository style architecture [7]. The boxes represent subsystems they have broken Vim into: Command to process user commands; File to read, write, and manipulate buffers; Global to contain all the global data structures; GUI to manage the graphical user interface; OS Interface to interact with the many OS supports; Terminal to work with the keyboard and mouse; and Utility for regular expressions and other miscellaneous activities. The arrows are dependencies between these modules (Global is in black to show that everything depends on it). On the top is the repository style architecture they believe Vim was meant to have, and on the bottom is Vim's actual architecture (for version 5.3). The black arrowheads in Figure 1b show dependencies that are permitted by Vim's intended architecture, and the white arrowheads are extraneous dependencies. The numbers annotated on the arrows indicate how many dependencies are between those two modules. The size of the arrowheads is proportional to the annotated number.

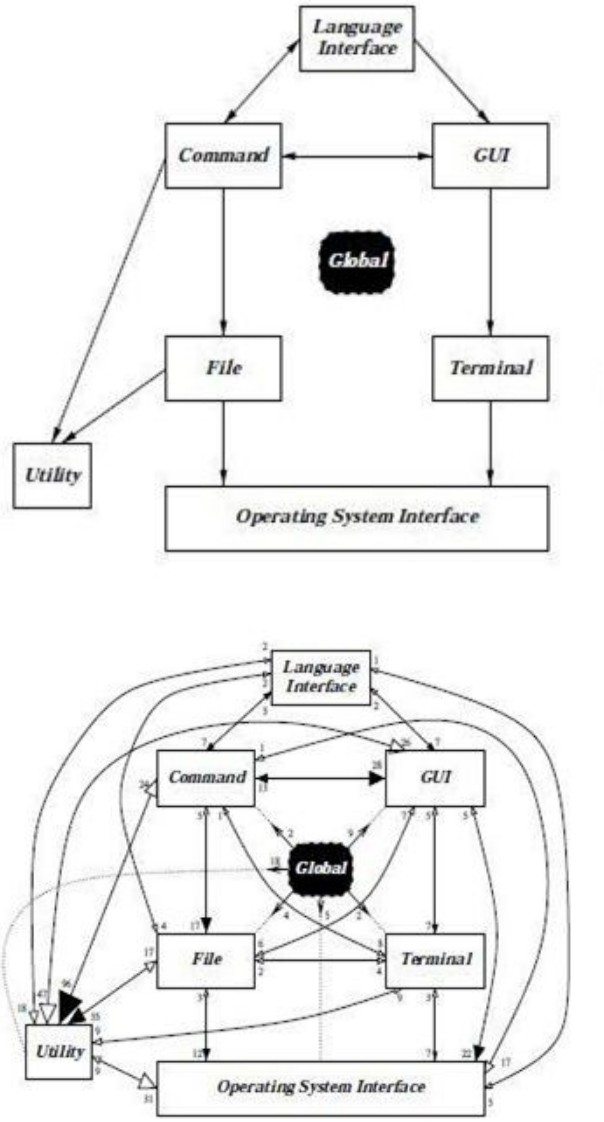

Figure 1: Conceptual and concrete architecture of Vim [Source: 7]

One of the observations made possible by Figure 1 is that Vim's actual architecture does not match its conceptual architecture. In reality, Vim is not a perfect example of either model-view-controller or repository style architectures. Because Vim is quite old and its open-source nature means that many developers have worked on it, its architecture has started to decay [3]. One of the most striking proofs of this is the existence of two files called misc1.c and misc2.c. These are both extremely large files (238 and 149 KB, making them two of the largest files in Vim's source code) and are filled with random functions that do not belong together but that have not been moved to the modules or classes where they do belong. Issues such as these have caused Vim's architecture to decay from its original style. However, Vim is still a growing and functional software system, and this is because [3] it was built as a collection of mostly-independent subsystems all oriented around one global data structure. That repository-style-like architecture makes it possible for Vim to survive despite its decay.

The authors of one source [7] described how they would redesign Vim to truly follow a repository style and to remove some of the dependencies. They divided misc1.c and misc2.c into pieces and assigned each of these pieces to the module where it actually belongs (the fact that this has not been done to Vim in practice suggests heavily that misc1.c and misc2.c exist simply because of the laziness of the developers; it is easier to keep everything in two huge files, but the authors of source [7] showed that it would be possible and more architecturally sound to distribute the pieces of those two files among the proper modules). By splitting some of the other modules and doing some shuffling (many files were moved to their Global module) the authors were able to create a theoretical Vim that does follow repository style architecture [7].

It is also worth mentioning that Vim's module decomposition was chosen based on functionality and features. This means that, because Vim was built to support many different platforms, all the platform-specific code is spread throughout all the source files. [2]. This module decomposition caused scattering of the platform concern and tangling, but if they had separated the concerns so that support for the multiple platforms was not scattered then the functionality and features code would be scattered. Thus, the existing module decomposition is probably well-chosen.

3. DOCUMENTATION OF THE ARCHITECTURE

Figure 2 shows the module decomposition of Vim that we discovered by studying Vim's code. There are a number of files - including vim.h, vimio.c, main.c (which contains the main loop), the two miscellaneous files mentioned in section 2 of this paper, and others - that make up a set of base classes, which we put into a module called Vim Base. All other modules depend on and are depended on by this base. Nearly every file in Vim's source code includes vim.h (through #include) and vim.h in turn includes files from almost every other module. This structure seems to have been set up for the convenience of the developers; they can simply include vim.h and then use nearly any part of Vim's code they want to, without having to include particular files. It would be possible to redesign Vim so that all its modules did not have a circular dependency on the base; this would simply require going through all the files and determining which part of vim.h that file actually uses. The base module depends on a small module called Alphabetical Mapping which is used to map ASCII characters to Arabic and Farsi.

The rest of the modules are: Programming Language Interface, which contains files to interface with several programming languages; Commands, which contains several files with code to execute the commands the user gives Vim; Buffer, which manipulates and loads the text files being edited; Screen, which contains the files screen.c as well as other more specific files to manage the terminal (term.c) and many different GUIs; and Operating Systems, which contains code to interface with several different operating systems. Which of these modules use each other is shown in Figure 3, our uses view. We left Vim Base out of the uses view, because it (through the main loop in main.c) uses everything, and everything (through vim.h, misc1.c, misc2.c and other such files) uses it. We also left out Alphabetical Mapping, because it is only used by Vim Base and uses nothing.

Under Vim's src directory there is also a directory called "testdir" which contains several files (test1.in and test1.ok through test73.in and test73.ok, and others) used to regression test Vim.

Developers can run these tests to make sure that their changes haven't broken any of Vim's functionality.

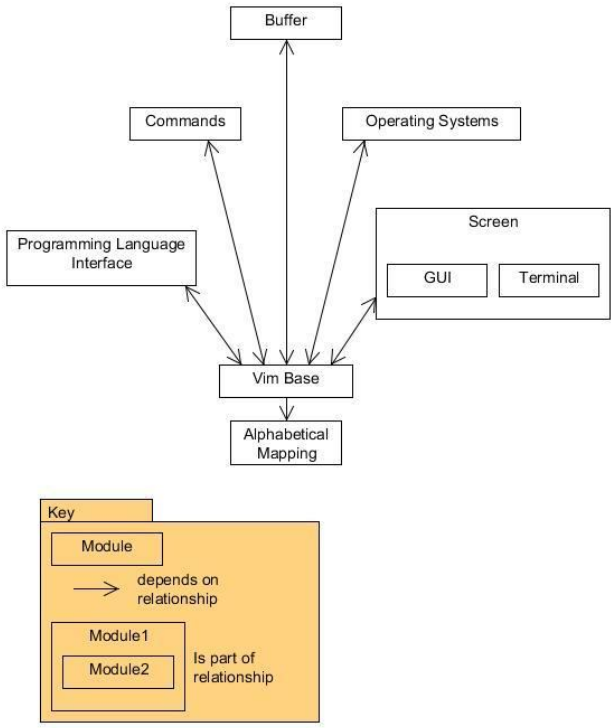

Figure 2: Module Decomposition View of Vim

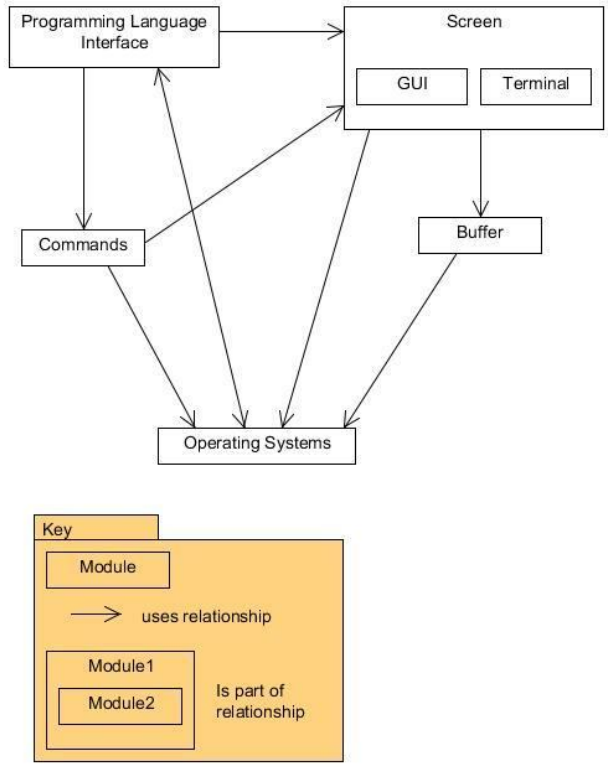

Figure 3: Uses View of Vim

3.1 Component and connector diagrams

Editors are one of the most commonly used tools on the web today. From editing documents to sending emails to instant messaging, they have a ubiquitous presence everywhere. Some web applications such as email clients define their own editors tweaked to their needs. But this is cumbersome as writing a full-fledged editor encompassing all the rich functionality and efficiency of mature editors like VIM and Emacs is not always feasible [4]. A better solution would be for the web application to invoke VIM or some other pre-existing editor to take care of editing tasks as and when required instead of defining a new editor altogether. For this to work, the client has to have VIM or some other editor preinstalled in his system so that the web application can invoke and use it. Managing editing sessions can be done in a generic way using the session id, but the actual execution of editor instances including formatting the data and text in each instance has to be editor specific using the functions provided by the editor. The main trick here is to separate the actual editor instances from the management of these editing sessions. Apart from enabling the application to invoke multiple instances of a particular editor, it also enables one to open multiple editors simultaneously. For this purpose, a Multiplexer is implemented in a generic way and a connector is provided for each of the supported editor. The function of the multiplexer is to enable the application to invoke multiple instances of the editor. The connector knows the editor specific details which are used as a communication channel between the generic multiplexer and the editor instance. Each editor session has a session id that is passed as a parameter when it is referenced. The Vim connector is implemented partially in C and partially in Vim's scripting language. The C functions implement the text editor interface. Commands from the MULTIPLEXER to the editor are sent using Vim's remote scripting feature. The second component-connector diagram shows details of the VIM runtime component which is executed once per editor instance.

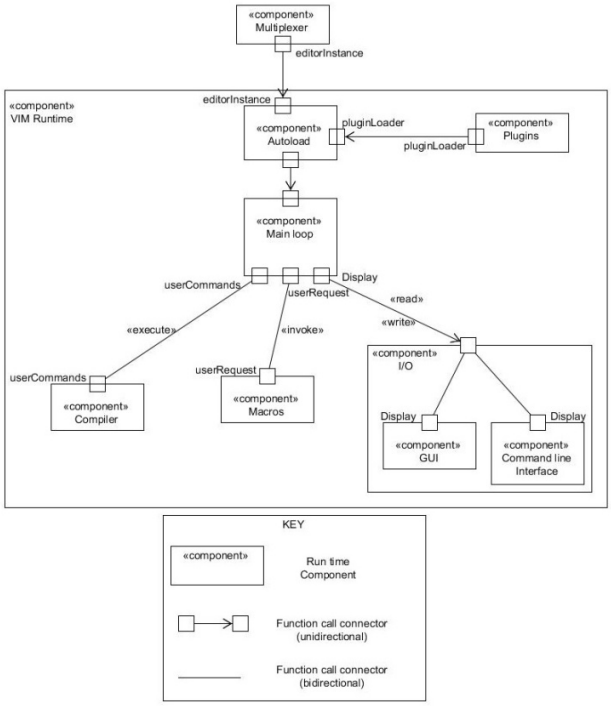

Figure 4: The VIM Runtime component

As can be seen from the figure, the Autoload component is invoked first that sets some configuration settings for the editor. VIM has several options to extend its functionality through plugins. The scripts are usually written in Vim's internal scripting language vimscript. Vim also supports scripting using Perl, Python, Ruby and contains modules to integrate them into the editor. If any plugins are installed, then they are loaded during this time. After this phase, the Main Loop component is started. The Main loop as the name suggests, is an infinite loop that keeps waiting for the user to enter something. The vgetc() function is used for this purpose. When the user enters a command, the Compiler is invoked to process it. After the command processing is done, control returns back to the main loop to await the next input from the user. The user also has the option to use some predefined macros which is stored as a separate component. The macros allow the user to define personalised key mappings like automate sequences of keystrokes, or even call internal or user defined functions. There is a separate component for I/O which provides the user with two options - to run VIM through the terminal (no GUI) or use the GUI version (known as gvim).The GUI is implemented as though it is a clever terminal [9]. Actions such as a mouse click, moving the scroll bar, etc. are all translated into events that are then written into the input buffer that is then read and processed by the Main Loop. The code for these actions is present in gui.c. The I/O component also contains options to refresh the screen using functions defined in a file called screen.c. The communication diagram in Figure 5 presents the message flow between various components of the compiler.

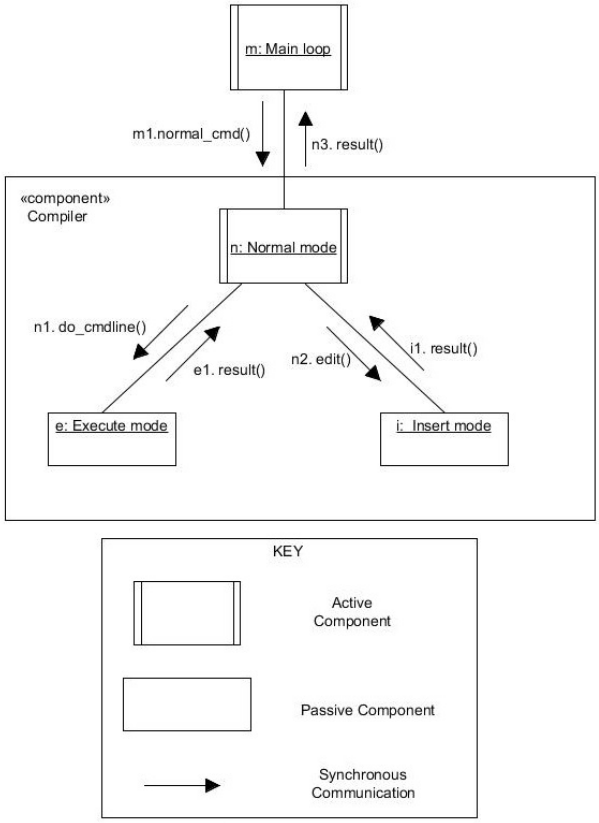

Figure 5: Concurrent Communication diagram

The Main Loop is an active component that invokes the compiler through the normal_cmd() function call when the user needs to process a command. By default, the compiler is the normal mode which is for navigation and manipulation of text. To insert new text, the user needs to move into the insert mode by pressing :i. In the context of the editor, pressing :i causes the edit() command to execute which switches the mode. Similarly, do_cmdline() method is called to move into the execute mode which is used to enter commands to the editor like :help (to invoke the help file), or :q (to quit the editor). The results of all these steps are ultimately passed to the main loop which then determines the flow of further control.

3.2 PROCESS

To build the module view, we first looked to see if Vim's code was already put into packages. It was not, so we then looked at the names of the files. It turned out that many of the file names had prefixes: os, ex, gui, or if (such as "if_python.c" or "os_unix.c"). We decided, by reading the comments at the beginning of these files and seeing that they do what it seems they would do, that these probably correspond to modules; Operating Systems, Commands, GUI, and Programming Languages Interface, respectively. By looking at the names of other files, reading their comments, and reading the README.txt included in the src folder that described some of the important files, we broke up the rest of the files into the rest of the modules. To find the dependencies within the module view (Figure 2) we looked at what files were "#include"ed by files in other modules. We found that almost every file simply included vim.h (and perhaps vimio.h, or another base class) in the Vim Base module, which in turn included at least one file from every module. We defined the "#include" relationship as dependency because those files would not be able to compile or run without the files that they "#include"ed.

To find which modules a certain module actually used we did text searches of its files to find if they called functions or used variables from other modules. We used Figure 1 for inspiration of what to check, but we found that the uses view was slightly different than what Figure 1 shows. Our results are shown in Figure 3.

To determine the components at runtime, we first read the official documentation of VIM that specified the main files needed to execute VIM. VIM has a README file in almost all directories containing its source code which gives an overview of the functionality of the artefact. The README files apart from containing information about the general functionality, also provide some clues as to the calling sequence by specifying the methods or parts from where this function is accessed. We attempted to build and infer the call graph in this manner. We then tried to map these files with the module view of VIM to determine which modules actually had a runtime presence. Using the calling sequence information found previously, we attempted to infer dependencies and used it as a guide to build our component diagrams.

4. DISCUSSION

We believe the overall architecture of Vim is repository style. Its functionality, as executed by the main loop in main.c, involves taking data from the buffer, processing it, executing the commands found in that data, and then doing something to the buffer or display; that is to say, it centres on the file that is being edited. All the modules are also set up to do something to or get something from the buffer, either directly or indirectly through the main loop or through the screen. Also, it can be clearly seen from

Figure 2 that all of Vim's modules depend on one central set of files (some of which, it's true, are totally miscellaneous and do not demonstrate sound architectural decisions, but some of the files in Vim Base are simply used to give access to files describing the buffer: for example, the Command module access the Buffer by including (with #include) vim.h, which in turn includes Buffer files).

Conceptually, it seems that Vim's architecture could easily be model-view-controller. In this situation the buffer (the data structure representing the file being edited) would be the model, the screen the view, and the commands the controller. One of the requirements of the model-view-controller style is that the model has no dependencies on either the view or the controller; this is met, because the Buffer module does not use the Screen or Commands modules. The view should use the model but not the commands; this also is met by the architecture of Vim that we discovered. However, in a model-view-controller style the controller should use both the view and the model; we found the Command module to only use the Screen module, and to control the Buffer through that. Nevertheless, it would probably be understandable to still describe Vim as a model-view-controller system. The reason we do not do so is because of the sloppy dependencies in Vim. With every single module depending on a set of base files, and those base files all depending in turn on all the other modules, it is difficult to see Vim as a clean model-view-controller style system (wherein each of the three pieces should be reasonably independent, and not depend on each other through a base module). As mentioned before, it would probably be possible to split up the files in the Vim Base module and redistribute the pieces of code among the other modules. If one also moved main.c to the Command module Vim could perhaps be redesigned into a true model-view-controller style. It is tempting to call the entire Vim architecture pipe-and-filter. However, in pipe-and-filter style the main idea is that all components work concurrently, receiving data from their input ports, transforming the data, and passing the data to the next component through their output ports via pipes in a defined sequence. However, in Vim the order of execution is highly customizable; there is no set sequence of operations, no straight pipeline. The order of the operations and which operations are done depends on the operating system, the user's settings, and other factors. This does not fit well into the idea of the pipe-and-filter style, so Vim is not a perfect pipe-and-filter style.

It is quite possible that Vim's architecture was intended to be model-view-controller or pipes and filters style, or even that it was meant to be a repository style system but with a much cleaner architecture and fewer extraneous dependencies. However, Vim is twenty years old and for all that time many different developers have worked on it. Whatever its intended architecture, Vim has eroded during its time spent in the hands of developers who didn't know the intended architecture or didn't care. This would be expected to happen to almost any open-source system of Vim's age, especially when one considers that the idea of software architecture is relatively new and it's possible that Vim's original designers were simply trying to make something that works, and not worrying about following a certain pattern.

One observation we made while looking through the code is that handing the operating system is a scattered concern, tangled in almost every other module. The Operating Systems module merely contains code to deal with different operating systems; all the other modules are forced to frequently do checks to see which operating system is being used and then execute different code accordingly. These checks take the form of:

```
#if defined(UNIX) && !defined(__BEOS__) &&
!defined(MACOS_X)
# define MAY_FORK
        Int dofork = TRUE;
#endif
```

This is an example from gui.c where that file is forced to check if a certain operating system is defined in order to determine what it should do. This kind of dependency would make it difficult to add support for another operating system to Vim, because not only would you have to create the interface to go in the Operating Systems module, but you would also have to look through all the rest of the code to find the places where checks on the operating system are done. It is difficult, however, to see how these dependencies could be avoided. Other files must know what operating system they are running under in order to determine what behaviour to use. The only solution would be if files outside the Operating Systems module never needed to change their behaviour depending on the operating system; this would, however, probably result in greatly increasing the size of the Operating System module and might require serious changes in Vim's organization and perhaps even functionality. Vim has been set up so that certain features are only available on certain operating systems [5].

One previously undocumented architectural design decision we discovered was the fact that the Screen module uses a generalization style. Vim is written in C and the generalization style usually corresponds to object-oriented designs, but the Screen module uses it nonetheless. The Screen module includes one file, screen.c, that serves as a generalization of the GUI code (gui_*.c) and the terminal code (term.c). screen.c has many methods that do something both a GUI and a terminal would need to do, and within them screen.c checks to see whether a GUI or a terminal is being used and then calls the appropriate methods. This allows methods from the Command module, for example, to call a method of screen.c without having to worry about whether a GUI or a terminal is in use. This generalization allows the Screen module to hide information and to present an interface to the rest of the system. That in turn decreases the coupling between other modules, such as Command, and Screen; Command only needs to know about one file, screen.c, instead of about all the gui and terminal files. The generalization also prevents the screen from becoming a scattered concern. This way, a developer could add a new gui, for example, and only have to make a file for that gui and add some code to screen.c. Without screen.c the developer would have to search for every file in Command that does a check on what gui is being used and update the code there, which would be repetitive and obviously not desirable.

Another undocumented design decision that we have discovered was the implementation of the folding function used pipe and filter style. Folding is representation of the several lines shrinked into to a single line to ease readability of the text. To make this process reversible the content of the buffer should not change. The C-struct inside fold.c file stores information about folded lines as well as well as an array of the other folds that are being nested into the current fold. Once the folding or unfolding takes place the information about the fold is processed inside the main loop and takes effect after the redraw operation gets performed. This kind of filter between buffer and the screen allows multiple folds as well as nested folds on a single file. Furthermore, the existence of folding function as a filter simplifies the implementation of screen as the developers would have to take

care of folding in both GUI and terminal modes in the screen component.

Vim supports *filter commands*, where a filter is a program that accepts and changes text using standard IO. This is done by providing access to shell commands and utilities without leaving Vim by using the :shell or :sh command. This is a very powerful feature that allows VIM to access the full power and functionality of the UNIX filter at no "additional cost" in size or performance of the editor. This makes it possible for VIM to access powerful utilities like grep and sed for powerful pattern matching from within the editor. Vim also provides options to change the shell that is used. In addition to all these features, one can execute a command directly from the editor, without needing to drop to a shell, by using bang (!) followed by the command to be run.

VIM is in many ways similar to bash architecture in the way it takes in data and processes it in various stages. VIM adopts a pipe and filter style to accomplish text editing transforming data at each stage. Like bash, VIM has many ways to accept user input which it then passes on to the compiler for processing. The compiler itself consists of many sub components each of which perform a particular functionality. The result is then passed to a central component or manager that is responsible for maintaining the current state.

One of the quality attributes that Vim achieves well is portability. In Vim's source code we found many files to interface with many different operating systems and programming languages. However, Vim's extensibility is more difficult to qualify. It has good extensibility in the sense that it would be easy to write a file for a new operating system or programming language, but a closer inspection of the code shows that the operating system concern is scattered throughout the other modules, so it might be more difficult to extend Vim than it originally appears to be. Thus, while Vim is portable and currently interfaces with many operating systems and programming languages, it would be difficult to extend it to work with an additional operating system because that concern is scattered. (Vim is still easily extensible to other programming languages, however, even if its operating system concern is scattered.) Vim's understandability is poor. Because it is such an old system and has thus been worked on by so many people, its code is not very consistent. The comments at the top of files, for example, were very different from file to file; some told the other and the date the file was written, some described the functionality in detail, some had hardly anything at all. One file had a comment at the top saying who the original author was, but then adding that the code had changed "beyond recognition" since it was written so it would not be worthwhile to contact that original author.

5. CONFORMANCE CHECKING

To check the accuracy of our module views Understand for C analyser was used [11]. Understand has a very useful utility of exporting all of the file dependencies into a single CSV file. To verify our uses view we wrote a small program in Java that took this CSV file as an input and mapped the source files into corresponding modules as well as produced the module dependencies. The reflection model for uses view is presented in Figure 5 below. As it can be seen from the model there are many divergences from our uses view. The reason for this lies in the fact that we examined each source files for external function calls to other source files manually. Some function calls that we missed while examining with the naked eye showed up while doing the deep automatic analysis of the whole architecture.

Our uses reflection model (Figure 5) supports our conclusion that Vim has a repository style architecture even more than our previous models did. It can clearly be seen in this diagram that every other module uses the Buffer module. This supports the idea that everything in Vim centres around one central data structure – the buffer, as represented by the Buffer module – and Vim does in fact have a repository style architecture.

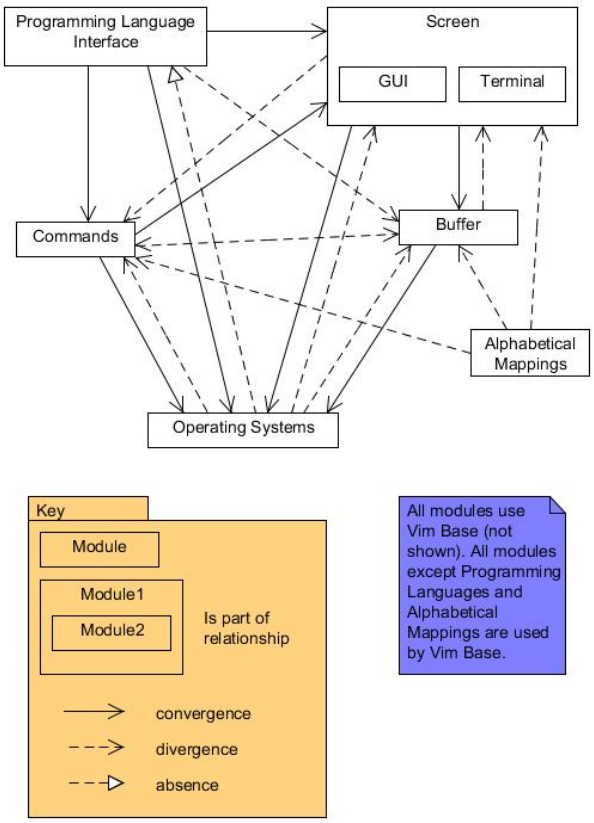

Figure 5: Uses Reflection Model

As is already mentioned in section 4, the dependencies in the module view were produced based on the include statements in the source files. By using the same Java program each source file was examined for the included source files and module dependencies were determined accordingly. The reflection model we produced for our module view is shown in Figure 6 below.

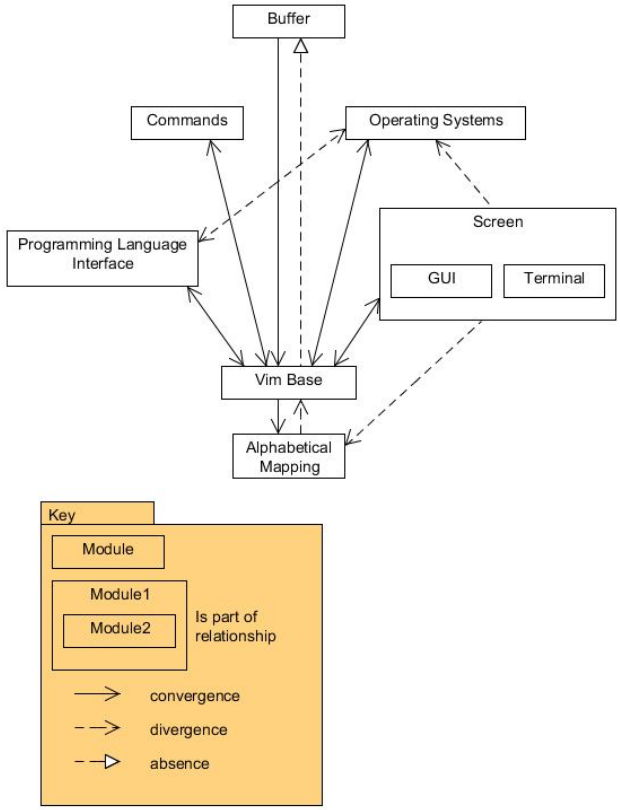

Figure 6: Module Reflection Model

By examining the order of invocation of entities on VIM (call graph) and by using the official documentation as a reference [10], we were able to determine that the following events take place in sequence. When VIM is called, it does a fork() and exits the current process. Vim then checks the environment variables (by calling vimrc.vim file) and sets their values accordingly. The plugin scripts are then loaded (a list of which is maintained in plugin.c). Then the main.c file is called which starts VIM in the Normal mode. Further files loaded depend on the actions performed by the users. While we were able to determine the order of actions performed, we were unable to find a direct concrete mapping between these actions and our components. It so happens that the components we thought VIM has does map to certain functionalities contained in specific files. But whether our components correspond to specific files or not was unclear. This is an open ended question we were unable to decide.

REFERENCES

[1] H. Dayani-Fard, Y. Yu, J. Mylopoulos, and P. Andritsos. Improving the build architecture of legacy C/C++ software systems. 8*th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 04-08, April 2005 http://oro.open.ac.uk/24311/1/dayani-fard05fase.pdf

[2] M. W. Godfrey and E. H. S. Lee. Secrets from the Monster: Extracting Mozilla's Software Architecture. *The Second International Symposium on Constructing Software Engineering Tools*, 15-23, June 2000 http://cpath.csi.muohio.edu/omeka/archive/files/a562e12fab321e2177dc44c5220e9336.pdf#page=15

[3] M Godfrey and Q Tu. Growth, Evolution, and Structural Change in Open Source Software. *Proceedings of the 4th International Workshop on Principles of Software Evolution*, 103-106, 2001. http://flosshub.org/system/files/tu2001.pdf

[4] Hayco de Jong and Taeke Kooiker. *My Favorite Editor Anywhere*. CWI, Department of Software Engineering. oai.cwi.nl/oai/asset/11023/11023D.pdf

[5] B. Moolenaar. *Vim Reference Manual*. http://vimdoc.sourceforge.net/htmldoc/develop.html

[6] R. Paul. *Two decades of productivity: Vim's 20th anniversary*. http://arstechnica.com/information-technology/2011/11/two-decades-of-productivity-vims-20th-anniversary/

[7] J. B. Tran, M. W. Godfrey, E. H. S. Lee, and R. C. Holt. Architecture Analysis and Repair of Open Source Software. 2000.

[8] University of Toronto. Tutorial II Vim Editor. *CSC408H1F/CSC2105H1F Software Engineering*, 2004-2005. http://www.cs.toronto.edu/~yijun/csc408h/handouts/tutorial2.pdf

[9] *Vim's web site.* www.vim.org

[10] *Vim wiki*. http://vim.wikia.com/wiki/Understanding_VIMRUNTIME

[11] *Website hosting Understand for C.* www.scitools.com

6. APPENDIX: CONTRIBUTORS

Cleo Kesidis: revisions from milestone 2, presentation, conformance checking analysis of uses and module views
Mangala gowri Krishnamoorthy: revisions from milestone 2, presentation, conformance checking of C&C views
Kamran Naghiyev: revisions from milestone 2, presentation, conformance checking coding and analysis of uses and module views