What's so great about 'grep'? Implications for program comprehension tools[1]

Janice Singer
singer@iit.nrc.ca
Institute for Information Technology
National Research Council, Canada

Timothy C. Lethbridge
tcl@csi.uottawa.ca
Department of Computer Science
University of Ottawa

[1] We would like to thank the members of the Mitel SX2000 team for their participation. We would also like to thank Pierre Fauvel, Charles Gauthier, W. Morven Gentleman, Howard Johnson, Jelber Sayyad, and Jim Uhl for their comments. This project is sponsored by CSER (Consortium for Software Engineering Research). Support has also been provided by NSERC.

What's so great about 'grep'? Implications for program comprehension tools

Abstract

The Knowledge Based Reverse Engineering Project is currently undertaking a study of a large group of telecommunications software engineers (SEs). From this endeavour, we have come to realize that the SEs in this group engage in three main activities, navigating the directory structure, using editors to view or change source, and grepping. This has us thinking about grep and why it used so much. In this paper, we present several hypotheses about why grep is such a good tool. We also talk briefly about its limitations. Finally, we present some principles that will enable tool makers to make better program comprehension tools.

1. Introduction

The Knowledge Based Reverse Engineering Project's goal is to provide software engineers (SEs) in an industrial telecommunications group with a toolset to help them maintain their system more effectively. To achieve this goal, we have adopted an user-centered design approach to tool development [1, 2, 3]. Therefore, the first phase of our project involves an intense study of the SEs' work practices.

We are currently interviewing and, more importantly, observing the SEs as they perform their everyday activities. From these studies, we have found that not only do the SEs become increasingly expert in their system but, more significantly, they become expert in *comprehending* parts of the system with which they are less familiar. This is a natural consequence of the fact that the SEs cannot hope to learn or retain everything about even a relatively small part of the system. Instead they must become expert at relearning different parts of the system as necessary, or what we term 'just-in-time' comprehension.

Furthermore, it has become abundantly clear that as the SEs engage in just-in-time comprehension, they search, search, search. As one SE put it: "First we search to find where the problem is, then we search to find potential solutions, then we search to do impact analysis." In the organizations we have studied, 'grep' turns out to be the SEs' primary searching tool.

This has led us to think a lot about grep. On the surface, this may seem like a trivial endeavour. Everyone knows what grep does, and everyone agrees that it is a good tool. However, using grep as the model for a program comprehension tool leads us to many interesting observations. Grep is by all measures an enormously successful tool. Why? We think that if we can gain insight into this phenomenon, we can build tools that are not only excellent, but will actually be used.

The following paper is thus an examination of what's so great about grep, and what we can learn from its analysis.

2. Usage of grep during program comprehension

In this section, we address three general themes of grep usage: why it is used; why it is necessary; and when it is used.

2.1 Why grep is used

In 1/2 to 1 hour episodes we have observed SEs as they go about their daily work. The aim of this observation is not to understand the SEs' mental models or program comprehension strategies, *per se*, but rather to record and gain frequency estimates of the types of activities that comprise their work practice. What we have found is that examining source code, simple directory navigation, and greping are the three most commonly performed low-level activities. This brought us to our first question, why is grep used so much?

One hypothesis has to do with the nature of programming itself. We have found that in the course of program comprehension, the SEs are nearly always goal-directed. That is, they are always understanding the system in the context of either an enhancement or a bug-fix. Given that this is so, the SEs are more focused on solving a problem than in understanding the structure of the source code as an activity onto itself. Grep allows for goal directed activity because it encourages means-ends analysis of the source code, as opposed to simply observing its structure.

Second, grep is the most widely available tool for extracting information from a system. By extracting information, we mean getting some idea of the structure of the system. Other tools such as tree-drawers, slicers, etc., are available, but not on as many platforms. More importantly, these other tools tend to change character (both in the interface and the specification) from platform to platform, and even project to project. Grep does not. Its interface and specification are pretty much identical on all OS' (even Windows).

Finally, grep works the way users want, unlike many other tools. We review some of these factors in section 3.

2.2 Why grep is necessary

Not only did we wonder why grep was used so much, we also wondered why grep was necessary. Here we think we can gain insight from the structure of the source code itself. First and foremost, the source code is not indexed. There are no direct pointers to particular parts of the code, therefore it is necessary to have some engine to search the entire source system to find things of interest. Grep allows the SEs to do this easily because it is fast and efficient.

Another reason grep is necessary is because the source code is organized in very big files. That is, there are several routines in each file, so the directory structure does not reflect the structure of the program, i.e., a simple search of the directory structure is not sufficient for finding all routines. On the flip side, related routines are scattered across many files, so a simple editor search within one file would again not be sufficient for searching purposes. Thus, the structure of the source code necessitates a general purpose searching engine that can navigate both throughout the directory structure and within each file. Grep does this.

2.3 When grep is used

The final general question we had about grep was when it is used. It turns out that grep is used primarily for navigation purposes in the service of three general themes. As the SE in the introduction said, grep is used for finding the problem, finding solutions, and evaluating the solutions.

Finding the problem refers to finding the place in the source code where the problem occurs. This is generally done by using grep to follow the control flow of the program, or looking for places where certain data is modified. The candidate locations are usually confirmed via replication of the situation that resulted in failure.

Once the source code location is found, grep is used to locate places where a fix to the program can occur. Here the SEs are generally searching for a particular variable or routine to change.

Finally, after a SE has decided on a candidate solution (and, in fact, sometimes after a solution has actually been implemented), SEs use grep to look through the source code to see what impact the solution would have (impact analysis). Here, grep is used to find routines or variables that access the changed routines or variables. This type of search is usually recursive until the SE is satisfied that the change will have no major impact.

Above we have concentrated on the general scenario of grep usage: why it is used; why it is necessary; and when it is used. The rest of the paper will focus more specifically on the details that make grep such a useful tool, and also some point out a few things that would make it an even better one.

3. Good things about grep

3.1 Success, little cost of failure, and understanding limitations = trust

SEs trust grep and the results it produces. This is no little thing. If a tool is trusted, it will be used over and over again, while a tool that is suspect will be discarded as soon as it produces an incomprehensible or failed result. This is especially true when one considers the time pressure faced by SEs. They do not have the time to mess around with tools that are not going to perform. Grep does perform, and is therefore trusted.

We hypothesize that grep is trusted so completely for three specific reasons: grep succeeds most of the time; when grep fails, there is little cost; and finally, SEs understand the limitations of grep and therefore in which contexts to use it.

Grep succeeds most of the time because it does a small and easily interpretable task. Grep only searches for text strings in a specified set of files or directory structure. That's all. Because of this, it is easy for SEs to understand what grep is doing. More importantly, though, it is easy for SEs to know what input to type to get out a desired output. Grep pulls no surprises - what you see is what you get. Because grep is thus a small tool that does a particularly well-understood task, it usually responds in the way that SEs expect. That is, it gives correct results most of the time.

When grep does fail, it generally does so for three reasons: it finds no match; it finds the wrong match; it finds too many matches. When grep finds no match, it could be that the target does not exist or that it was incorrectly spelled. When grep finds an incorrect match, the SE may have inadvertently looked for the wrong target, or again, incorrectly spelled the target. When grep finds too many matches, the SE may have specified the wrong subset in

which to search, or been unaware that the target was used so frequently. In all cases where grep fails, though, it is generally pretty easy to understand why, and more importantly it is easy to re-specify a search that will be acceptable. Therefore grep usage costs SEs little time and mental effort, even when it produces unexpected results.

Finally, grep has plenty of limitations, but again, because it is a small tool that does one thing well, it's limitations are easy to understand and further act to define the role of grep in programming. If you are doing a search for a text string in a directory structure or group of files, use grep. If you are doing something else, don't use grep. Thus, because grep does a very well defined thing, it is obvious when you should be doing that, or indeed when you should be doing something else. This fact again leads to grep's sucess. Grep is used when it is the right tool to be used.

It cannot be stated strongly enough. Tools that are trusted are tools that are used. Tools that are trusted are tools that perform the way they should and are well understood (even when they fail). Grep is such a tool because it generally succeeds, costs little when it fails, and has a specific context of use. SEs trust grep, therefore grep is used.

3.2 Command line interface

Simply put, most SEs very much like command-line interfaces. There are a number of reasons for this.

SEs are more adept at typing commands then they are at using menu-driven systems. Furthermore, it is not clear that you can gain any power or usefulness in searching by using menus and dialogue boxes. With a command line interface, SEs can easily specify search targets and parameters, without having to navigate a complex menu system or dialog box. Perhaps menus would be more efficient if the SEs were selecting pre-specified subsets of files on which to search. However, at this point, we observe SEs specifying all sorts of different subsets. With a command-line interface, they can easily change the subset at will.

Additionally, command line interfaces give SEs the chance to repeat and edit previous commands, and easily review the results of recent commands, etc. Related to this, command-line interfaces are more amenable to the creation of scripts and macros that automate repeated tasks. We have observed software engineers writing special scripts that provide specific macros to grep or search in specific subsets of files. It is only natural that SEs would want to do this since, after all, they are expert programmers.

3.3 Straightforward specification

The specification for grep is straightforward and easily understood. Little typing is involved and the syntax resembles a natural language search. Although various versions of grep have specialized options, users can ignore them if they like. SEs are only required to specify two arguments: a target pattern to search for and a source domain in which to search for it. In contrast, many sophisticated searching tools make all the options visible at every invocation, confusing the user, and forcing him to make choices that may or may not make sense in the context of the search.

Additionally, grep's specification makes it easy to see in which files or directories the search will take place. There is no mystery involved. If an SE wants to look at only one subset of files, he can easily accomplish this. If on the other hand, he wants to search an entire directory structure, this is available as well.

Grep also uses wildcards and pattern matches. This allows SEs to specify inexact searches. This is extremely important. Sometimes SEs cannot remember all the variable names; sometimes they know that an interesting set of variables all begin with a certain code and they would just like to find those, etc. The ability to use wildcards vastly increases the number of successful or semi-successful searches that may be conducted.

Finally, the straightforward specification of grep searches also allows SEs to understand failures. When a SE gets an unforeseen result, it is usually pretty easy to look at how they specified the search, and see why they got the results they did. That is the specification is clearly related to the results. The SE does not have to go back and agonize about how various options interacted to generate specific results.

3.4 Works in 'parallel'

Grep produces many results at once. All the places that match the search pattern are displayed. Many other search tools (including searching within many editors) walk you through each occurrence of the target in the source. It takes precious time away from the SE to find the occurrence of interest. With grep, the SE can go directly to the part of the source code that they would like to see. This allows the SE to direct the search, rather than having to follow a tool's model of how a search should proceed.

3.5 Widely available with identical implementations (from the user's perspective)

From a user's perspective, grep is the same on many different platforms and across many different operating systems. Because of this, it is not necessary to relearn grep or its peculiarities each time one changes machines. The knowledge about how to use grep already exists, so SEs are likely to try it as their first line of attack in exploring a new system. That is grep knowledge very clearly transfers from one machine to another.

3.6 Scaffolded

Grep works well for both novice and expert programmers and also for novice and expert grep users. That is, a programmer does not have to be a SE and a SE does not have to be an expert grep user to get something from grep. It supports searches for many different skill levels, and because of this, is used by many different skill levels. As people become expert SEs and expert grep users, they can then change their usage patterns to reflect their current level of understanding. In this way, grep behaves like an adaptive user interface, revealing itself as the skill level of the user increases, and only if the user feels the need.

3.7 Mental model of task matches tool parameters - what comes first, chicken or egg?

Finally, grep matches well the mental models of SEs. That is, grep defines searches the way SEs expect searches to be defined. It's not clear here, whether grep itself is a superior tool, or whether grep usage over time modifies the mental models of those using it. It does not seem to matter that much. What does matter is that it now matches the expectations of the expert SEs in the group.

Because of this, the cognitive burden for using the tool is very small. That is, it does not take alot of brain power or intensive thinking to figure out how to initiate a search. This brings us back to the goal-directed activity of SEs - grep allows the SEs to focus on the goal of solving the problem, and not worry about the correct usage of the tool. Grep is in many ways an invisible resource. Using it is so ingrained into the thought and usage patterns of the SEs that even were they to try, they would not be able to think of searches in another way.

Because grep is so much used, then, the cognitive load of SEs is reduced, thereby enabling them to concentrate on the real problems they face, and not the problems inherent in figuring out and using new tools.

Thus grep is a tool that is used often and used well. There are several reasons for this. Grep search specification is well understood and can be done via a command-line format. Grep has something to offer to SEs of many different skill levels, and is available in identical form across many operating systems and platforms. Grep allows SEs to direct the search process. Most important of all, though, grep is trusted, and it is trusted because it succeeds, and more importantly, because SEs understand its limitations and therefore the appropriate contexts of its use.

4. Current limitations of grep

In the previous section we have identified some good things about this ubiquitous tool. However, grep is not without its downside. In this section we identify some limitations to grep's usefulness. The key is to improve grep without detracting from the advantages we have listed above. This issue will be further considered in the final section of the paper.

4.1 Output interpretation requires effort.

Users of grep have to spend considerable time reading and interpreting its output. The actual parts of the text that match the search pattern are not immediately visible (only the entire line that contains the match).

This could be partially rectified by making the same improvement that was made to produce the Gnu tool 'less' from the tool 'more': All the matched strings could be highlighted.

Another problem with output interpretation involves context. Most versions of grep allow the user to see only those lines that match the pattern. It would be useful if all versions of grep would automatically display a couple of lines above and below the 'hit.

4.2 No near searches

Grep provides no facility to correct spelling or otherwise help the user whose search has retrieved nothing. A more intelligent grep that, upon failure, tried variants of the search pattern (and of course informed the user that it had done so) would be useful, and a nearly transparent improvement.

4.3 No semantic searches

Grep searches only for matching character strings in text. An improved grep might allow the user to, for example, limit the search to matching variable names (routine names, words in comments etc.) and/or significant editing parameters (such as underlining and colour) in source code.

4.4 No memory

Each grep invocation starts from scratch. It might be useful if the results of previous greps could be preserved automatically so various intelligent enhancements could be made:

- The user could display a history of past searches, and repeat selected ones or re-display the results
- The user could repeatedly work within the same context (i.e. the same set of files) - without the need to re-specify this each time.
- The user could search for the same pattern within a different context.

Shell-based history mechanisms can only provide limited help with this sort of thing.

4.5 No browsing

One cannot easily browse the results of grep. By 'browse' we mean, click on a search result and display it in the context of the file. Although a grep-like capability embedded in emacs can do this, it would be useful if grep itself had an option to present its results in this manner.

4.6 Slow, if you grep over a very large set of files.

Many software systems have thousands of files and millions of lines of code. Searching through a significant number can be time consuming. It would be nice to have a facility that would pre-index source files (as in an information retrieval system) so that grep could run much faster – of course, except for the speedup, grep would work exactly the same way.

5. Conclusion: What can grep tell us about building good program comprehension tools?

Grep is probably the most widely used program comprehension tool, because of (and despite) the fact that it is a small tool with a simple specification. Researchers in this field would benefit from studying why such a tool is so popular and, as a consequence:

- Develop new tools that have some of the same attributes as grep
- Improve on grep, without sacrificing its advantages.

5.1 Desirable attributes of tools

As of a result of our studies of grep, the following are some of the attributes that we believe a new or enhanced tool should have:

a) It should be small with a simple specification: The SE should perceive that the cost of learning it (or re-learning it as necessary) should be less than the benefit obtained from its use. Note that it is the SE's *perception* that is important, not the real cost/benefit analysis.
b) It should be flexible: The SE should be able to do a lot with it. A tool that is only occasionally useful will be forgotten.
c) It should be usable from the command line and easily integrated with the other tools SEs use on a regular basis. In other words, the tool's designers should recognize that SEs are already successful at what they are doing, and have established work practices that should be enhanced, not replaced.
d) It should be in the public domain and available on multiple platforms, so it can gain widespread use.

5.2 Desirable attributes of research practices

In addition to telling us about how to design a good program comprehension product, our studies have also suggested some improvements researchers in this field can make to their tool-development process:

a) Study the needs of software engineers and develop tools bottom-up using user-centered design. Today, too many tools originate with 'brilliant ideas' on the part of the researchers. Although these tools may have powerful capabilities, the costs of learning and using the tool may exceed the benefits. The bottom-up approach to tool development is becoming recognized as an important future direction in program comprehension research [4]. Among software engineers, there is a general skepticism about tools: SEs are often reluctant to even try a new tool, partly because of the bad reputation of existing tools. The less we focus on the SEs' needs, the worse this reputation will get.

b) Look at existing tools that software engineers use, and understand both their good and bad points. Then try to improve existing tools by eliminating the bad points while preserving the good ones. Unfortunately, in their zeal to improve tools, many tool developers are not aware of some of the good attributes (e.g. simple specification). They therefore tend to eliminate these too.

c) When you do develop big, complex tools (perhaps because you have no choice), at least think of integrating facilities that software engineers are used to (such as grep).

5.3 Conclusion

Grep is by all measures an enormously successful tool. In this paper, we have explored some reasons why we believe this to be the case. First and foresmost, SEs trust grep. They trust it because it is reliable, and because it behaves in the way they expect it to behave. It matches their mental model of the search process. This kind of trust is not something to be taken lightly, and in the end has to be earned by tools, because it is not given freely by SEs. We believe therefore that it is extremely important to understand why SEs use the tools they do, and to aim towards enhancing their work environment, and not replacing their work practice. In this way, we believe we can build small tools to solve specific problems that will gain acceptance and eventually be well trusted by the SEs using them, just like grep.

6. References

[1] Singer, J. and Lethbridge, T. (1996), "Methods for Studying Maintenance Activities", *Proc. Workshop on Empirical Studies of Software Maintenance*, Monterey.

[2] Lethbridge, T and Singer, J. (1996), "Strategies for Studying Maintenance", *Proc. Workshop on Empirical Studies of Software Maintenance*, Monterey.

[3] Avison D. and Wood-Harper T. (1990), *Multiview methodology*. Oxford: Blackwell Scientific

[4] Tilley, S. and Smith, D. (1996), *Coming Attractions in Program Understanding*, CMU/SEI-96-TR-019. Software Engineering Institute, Carnegie Mellon University.