Concern Highlight: A Tool for Concern Exploration and Visualization

Eugen C. Nistor * André van der Hoek

Department of Informatics
School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425 USA
E-mail: {enistor,andre}@ics.uci.edu

Abstract

Separation of concerns is a powerful principle that can be used to manage the inherent complexity of software. One of the benefits of separation of concerns is an increased understanding of how an application works, which helps during evolution. This benefit comes from the fact that the code belonging to a concern can be seen and reasoned about in isolation from the other concerns with which it is tangled together. In this paper, we present our tool that gives the developer the freedom to annotate existing code based on different concerns. These concerns can be later analyzed or visualized as highlighted text. Although our experiences were based on the special case of high-performance computing programs, we believe that the observations are pertinent to general programming as well.

1 Introduction

A concern denotes anything of importance while developing software. Naturally, every piece of software will have different concerns tangled together, and this complexity makes not only software construction difficult, but makes long term evolution and maintenance difficult as well.

Separation of concerns is an important principle in software engineering. The idea behind separation of concerns is that one should be able to reason about a single concern separately from the other concerns tangled in the final program. Techniques such as Aspect Oriented Programming (AOP) [1] offer the mechanisms through which concerns can be written in separate modules as aspects, and then weaved into an existing application at specified join points. However, the benefits of being able to identify concerns in code, and to use these concerns to explore the code should not be limited to programs following AOP. In some cases it might not always be possible to identify and write concerns separately from the start, like in the case of exploring projects for which source code cannot be modified. The same is true for situation where identification of concerns will lead to a future refactoring where some of the concerns will be written as aspects. Furthermore, even in programs written using AOP, different concerns might crosscut the code inside one particular aspect, and a tool like the one presented here might be useful in visualizing overlapping concerns.

In this paper we present the Concern Highlight, our tool for concern exploration and visualization. The tool is built as an extension to the Concern Manipulation Environment (CME), an aspect-oriented software development environment [2], and adds support for automated and free selection of source code snippets that belong to different concerns. These concerns can then be visualized as source code highlights in an editor in Eclipse [3].

The development of the tool came from our experiences of trying to apply concern-based development and evolution to the specific domain of high-performance programs written using the Message Passing Interface (MPI) library [4]. This is an interesting field because it exhibits the particularities described above that make traditional aspect-oriented approaches difficult or impossible to apply. In the rest of this paper, we discuss the specifics of MPI programming related to aspect-orientation, and present our Concern Highlight tool. Although our work is exploratory, we believe that our observations can be suc-

*The work described in this paper was done while at the IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA

cessfully used to help concern-based development and evolution in regular programming as well.

2 High-performance Computing with MPI

The initial motivation for the project was to bring the benefits of identifying concerns and using the CME to programs written in high-performance computing. Scientific programming in general – and high-performance programs written for parallel processor environments in particular – have a number of particularities: they are in their majority written in a procedural language such as C and Fortran, they are difficult to change because of their complexity, and the information content tends to be very dense because of the algorithms embedded into code.

Message Passing Interface (MPI) is a standard for high-performance computing libraries that provides a common interface for programming in a multiprocessor environment. The power of MPI comes from the fact that a complex middleware functionality is hidden from the programmer behind a set of well defined method calls, and the parts of the program that are executed in parallel coexist with the code that will be executed only on one processor. Specifying MPI as a standard assures separation from specific hardware implementations and interoperability across different MPI implementation libraries.

Although the complete MPI specification contains a significant number of methods, a few of them are very important and used extensively. However, while using these functions, the developer needs to be aware of special requirements and assumptions – which we can consider under the realm of MPI domain knowledge – that if not respected, can lead to errors. For example, the code that will be executed in parallel is separated by two special method calls, *MPI_Init* and *MPI_Finalize*, that delineate the start and the end of the parallel code, and all the other MPI calls have to occur in between these two. Methods *MPI_Send* and *MPI_Recv* are responsible for sending and receiving data needed for communication between processes. Their parameters determine where the data is sent to or coming from, and the process where these methods are executed is blocked until the methods calls are complete. Other methods include support for asynchronous communication, synchronization, I/O, timing and logging [4].

3 The Concern Highlight Tool

CME contains a number of tools that support concern-based software exploration, implemented as Eclipse plug-ins [5, 3]. One of such tools is the Concern Explorer, which presents a hierarchical model of different concerns identified in the software. These concerns are either defined by the developer, or loaded from specialized concern loaders from source code and other development artifacts, such as build files or design documents. A powerful query evaluator can be used in the Query View to evaluate queries over the concern model. These queries not only help browsing the concern model, but can also be saved in the concern model as new concerns.

The Concern Highlight tool is an Eclipse Plug-in that can be used to mark up source code ranges belonging to different concerns. The highlight tool functions as a complex code annotation system, but its integration with CME allows these concerns to be browsed or queried afterwards in the CME Query Analyzer. Figure 1 shows the Concern Highlight tool in a typical usage in Eclipse. On the left-side, CME's Concern Explorer displays the concern model. On the right-side, the Highlight List contains a list of concerns that the user is interested in exploring, populated with concerns selected from the Concern Explorer. The editor is shown in the middle, with the code belonging to the selected concerns highlighted in the text.

The user can associate source code snippets with concerns in the highlight list in two ways: either automatically, by using the CME's API, or manually by marking up source code in the current editor opened in Eclipse, through a context menu option. Automatic detection of source code related to a concern is desirable, but at the same time is problematic since it is highly domain dependent and depends a lot on the type of concern sought.

The format of MPI programs makes a number of concerns to be suitable for being marked up automatically in code, but in general we found three distinct types of concerns:

- Concerns that can easily be identified automatically. A simple example is marking the beginning and ending of MPI-related code, which corresponds with the *MPI_Init* and *MPI_Finalize* function calls.
- Concerns where automatic identification is possible, but manual marking would be much simpler. A good example is to try to find out usual MPI code patterns like the one shown in Figure 2. The code shown is a common MPI pattern, where one of the processes, commonly called the *master process*, has extra responsibilities than the other processes, usually related to input and output or distribution and gathering of data. Automatic detection of this pattern is fairly complex, while the developer could have marked the code as belonging to the concern while writing the code.

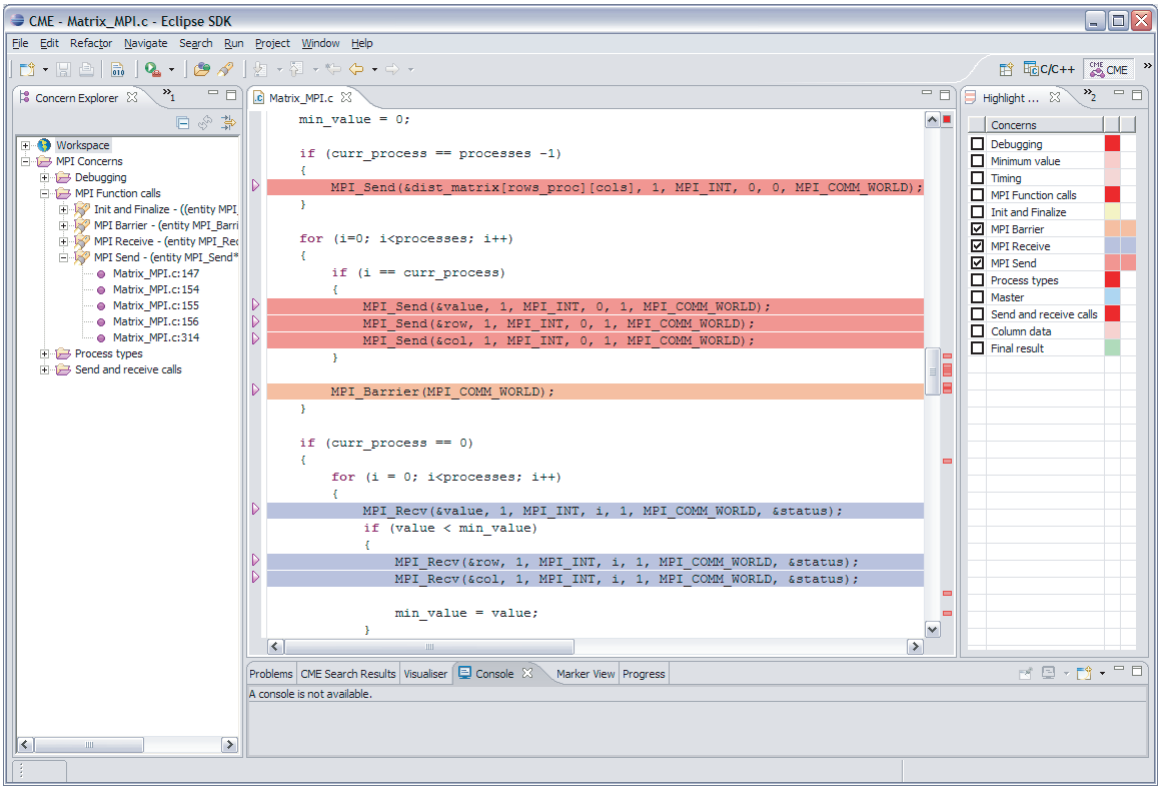

Figure 1. A screenshot of the Concern Highlight tool in Eclipse.

```
int myrank;
...
MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
...
if ( myrank == 0 ){

     /*code that is only executed on the
       first process */

}
```

Figure 2. A typical MPI pattern.

- Some concerns, such as the name of the user that performed a certain change in code, or trying to determine the code that was changed when transforming a program from a non-parallel to a parallel version, might be either impossible to detect automatically, would have to rely on comments written consistently throughout the code, or be imported from some other artifacts such as configuration management logs. In this case, human identification of concerns seems like the best solution.

Manual marking of the source code is more suitable for scenarios where the user wants to understand an existing application's implementation. Many times the source code for the libraries one uses in a project forms the most complete documentation at hand. The task of finding out how the existing application works is driven by a type of concern, which is the problem that needs to be solved. Marking up this concern will prove beneficial in documenting the code in such a way that if somebody else will be interested in the same concern to be able to focus on only those parts of the code that are relevant. For example, somebody might be interested in only looking at the code pertinent to the *master process*. In order to do so, they would select the code that looks like the pattern in Figure 2, either manually or with the use of a tool, and mark them as being a part of the *master process* concern. The integration with CME adds the benefit of persistence, such that if somebody new to a project wants to understand the code based on concerns, can use the Highlight View later to select this concern and to quickly see where that source code is located.

One of the important features of our tool is the visualization of concerns by highlighting the text of the source code ranges associated with it in the current ed-

itor in Eclipse. The concerns in the highlight list have an associated color, and the user can choose, by selecting a check-box near each concern, which ones should be highlighted. Since the concerns have a hierarchical nature, selecting a parent concern will automatically select all of the other concerns that are its sub-concerns. In this case, all the source code ranges associated with the sub-concerns will be highlighted using the color of the parent concern, unless they are explicitly selected themselves in which case they will be displayed with their own color. Figure 1 shows three different concerns selected and the corresponding source code snippets highlighted in the editor. The highlight feature is implemented on top of Eclipse's support for annotations, and currently most text-based editors from Eclipse are supported.

4 Observations

The main purpose of our tool was to enhance software understanding through recording code exploration traces and associating them with concerns. The main benefits would be seen during software evolution stages, since the code can be browsed on a concern-basis rather than the file or class-based organization. Moreover, if some part of the code needs to be changed, other parts of the code associated with it through a concern will probably hint the developer where to be careful about the effects of the change.

Besides helping understanding and existing program, we believe that concern-based development can potentially avoid common errors. MPI programming includes a number of domain rules that are either documented in books [6, 7], or are learned with experience. Some of the typical errors can be found with the help of either a static or dynamic analyzer. However, we believe that using concern modeling can help avoid some of these errors and offer significant advantages during software evolution.

Probably one of the most compelling arguments in this regard is based on a simple observation related to a typical development scenario that disregards concerns. While writing the software, the developer's intentions, which are naturally related to a concern, are translated into programming. This information is encoded in the specifics of the programming language used, in the libraries used, and possibly in comments. In this way, the concern information is lost as an explicit description. However, after the program is written, special analyzers will try to determine the presence of errors. While errors related to the syntax of the program can be easily found by a compiler, the more complex errors are based on domain knowledge. Typically, these errors can be found by trying to infer the concerns back from the code, and to figure out if a domain-based pattern was broken. Having concerns marked out and preserved together with the program being developed has the potential of avoiding some of these errors.

In the current state, our tool can help the developer in detecting errors by perusing the code related to a concern. As an example related to the scenario above, a simple rule in MPI is that *MPI_Send* calls have to be matched by corresponding *MPI_Receive* calls. When developers write the code, they know exactly which *MPI_Send* is matched by which *MPI_Receive*. However, this information is then lost as explicit information, and only encoded in the parameters given to these method calls. Later, static and dynamic analyzers will try to detect errors due to wrong values for parameters, and in order to do so they will try to guess back these correspondences. Depending on how the program was written, it might not be possible to achieve this statically since the values of the parameters have to be evaluated. Figure 1 shows such an example, with the editor showing a piece of code with four *MPI_Send* calls and three *MPI_Receive* calls. Only the three *MPI_Send* calls grouped together correspond to the *MPI_Receive* calls. This is a type of information that can save important resources consumed by a dynamic analyzer that can induce an important overhead [8]. Moreover, just by looking at the code, the fact that the *MPI_Send* calls are grouped together, and the corresponding *MPI_Receive* calls are not, can give the developer a hint: possibly not all messages that are sent are also received. A quick look at the code reveals that indeed, some of the messages are not received unless a condition is fulfilled, and this can lead to memory leaks and failure.

Although the initial use of the tool was on procedural programs, we believe that its benefits will be maintained in both regular object-oriented programs as well as programs written using aspects. Aspect-oriented programming offers the mechanisms to write the code that belongs to a concern in a separate module. However, concerns can be cross-cutting each other in such a way that the same line of code belongs to multiple concerns. An example from our experience with MPI is having the the same *MPI_Send* call belonging to a master program, being linked to some other *MPI_Receive*, and dealing with some special variables important in the algorithm. Isolating such lines of code in separate aspect modules just based on one of those concerns, using AOP for instance, still leaves the problem of having the other concerns overlapping. Therefore, even in programs where some aspects are written in separate modules, our tool would still be useful in visualizing how the different concerns overlap in the source code.

5 Related Work

Both CME and the AspectJ Development Tools projects [2, 9] include a source code visualizer, originating from SeeSoft [10]. Our tool is similar to them in the fact that it shows the position of the occurrence of each concern in text, and identifies each concern with a different color. However, our tool displays this information in the editor for each individual file, with the user selecting which concerns to be displayed, while the other visualizers show a global view of the code more suitable for a statistical overview of an entire project.

FEAT [11] is a similar tool in that it is offering a mapping between source code text and different features (or concerns). However, FEAT purposely moves away from free source-code mark-up text and stores relationships between different language artifacts, such as method calls traces, as concern graphs [12]. We preferred a free annotation solution, where the user is selecting any part of source code that they think is necessary. FEAT is also implemented as an Eclipse plug-in, but we could not use it in our projects since it works on Java implementations, while our MPI programs were implemented in C or C++.

The Aspect Mining Tool (AMT) [13] presents a discussion of text-based and type-based analysis as tools for discovering hidden concerns in code, with a visualization tool also similar to SeeSoft. We found that since MPI programming is much more rigorous than general programming, text-based and type-based mining is easier in finding MPI-related concerns. However, the AMT tool could provide useful insights for evaluating the possible use of the Concern Highlighter in general object-oriented programming.

6 Conclusion

In this paper we presented our experiences with developing a tool for concern-based exploration. Although initially developed for MPI programming, for which we have discussed its potential benefits and possible uses, we believe that the Concern Highlight tool would prove helpful for identification and visualization of concerns in any type of programs, even the ones written using AOP. The identification of concerns through our tool, together with the integration with CME, can increase software understanding and ultimately help with software evolution.

Our work was exploratory, and a better evaluation of its usefulness can only come from its use in real-life projects. In its current form, our tool can be used to determine what kinds of concerns are more likely to be identified, and what are the possibilities of automation in concern detection, for both MPI programs and in general programming.

The Concern Highlight tool is integrated within CME and is available for download from http://www.eclipse.org/cme.

7 Acknowledgements

The authors would like to thank Harold Ossher, Stan Sutton and William Chung from the IBM T.J. Watson Research Lab for their valuable input and guidance throughout the project.

This research was supported in part by the Defense Advanced Research Projects Agency under grant NBCHC020056.

References

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming. (1997)

[2] Concern Manipulation Environment Eclipse Technology Project. (http://www.eclipse.org/cme)

[3] Eclipse. (http://www.eclipse.org)

[4] The Message Passing Interface (MPI) Standard. (http://www-unix.mcs.anl.gov/mpi/index.htm)

[5] Harrison, W., Ossher, H., S.M. Sutton, J., Tarr, P.: Concern modeling in the concern manipulation environment, IBM Research Report RC23344 (September 2004)

[6] Snir, M., Otto, S.W., Walker, D.W., Dongarra, J., Huss-Lederman, S.: MPI: The Complete Reference. MIT Press (1995)

[7] Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press (1999)

[8] Vetter, J.S., de Supinski, B.R.: Dynamic software testing of mpi applications with umpire. In: Supercomputing '00, Washington, DC, USA, IEEE Computer Society (2000) 51

[9] AspectJ Development Tools Eclipse Technology Project. (http://www.eclipse.org/ajdt)

[10] Eick, S., Steffen, J., E.E. Sumner, J.: Seesoft - a tool for visualizing line oriented software statistics. In: IEEE Transactions on Software Engineering. (1992) 957–968

[11] Robillard, M.P., Murphy, G.C.: Feat: a tool for locating, describing, and analyzing concerns in source code. In: Proceedings of the Proc. Int. Conf. on Software Engineering (ICSE). (2003)

[12] Robillard, M.P., Murphy, G.C.: Concern graphs: Finding and describing concerns. In: Proceedings of the Proc. Int. Conf. on Software Engineering (ICSE). (2002)

[13] Hannemann, J., Kiczales, G.: Overcoming the prevalent decomposition in legacy code. In: Workshop on Advanced Separation of Concerns (ICSE). (1991)