Identifying Cross-Cutting Concerns in Embedded C Code

Magiel Bruntink Arie van Deursen Tom Tourwé

June 2, 2004

The identification and refactoring of cross-cutting concerns is the topic of Ideals, a four year research project conducted in cooperation with ASML, the world market leader in lithography systems based in Veldhoven, The Netherlands. The research partners in this project are CWI, TU/e, UT, and the Embedded Systems Institute (ESI) based in Eindhoven. The project started September 2004.

Large-scale industrial software applications are inherently complex, and thus a good separation of concerns within the application is indispensable. Unfortunately, recent insight reveals that current means for separation of concerns, i.e. functional decomposition or object-oriented programming, are not sufficient. No matter how well large applications are decomposed using current means, some functionality, typically called *cross-cutting concerns*, will not fit the chosen decomposition. As a result, implementations of such cross-cutting concerns will be scattered across the entire system, and become tangled with other code. Obviously, the consequences for maintenance of the system, and its future evolution, are dire.

Aspect-oriented software development (AOSD) has been proposed as an improved means for separation of concerns. Aspect-oriented programming languages add an abstraction mechanism (called an *aspect*) to existing (object-oriented) programming languages. This mechanism allows a developer to capture cross-cutting concerns in a modular way. In order to use this new feature, and make the code more maintainable, existing applications written in ordinary programming languages should be evolved into aspect-oriented applications. To that end, (scattered and tangled) code implementing cross-cutting concerns should be identified, and subsequently be refactored into aspects.

Identifying Cross-Cutting Concerns

Source code implementing cross-cutting concerns (CCCs) tends to involve a great deal of duplication. First of all, since such code cannot be captured cleanly inside a single abstraction, it cannot be reused. Therefore, developers are forced to write the same code over and over again, and are tempted to just copy and paste the code and adapt it slightly to the context. Alternatively, they may use particular coding conventions and idioms, which also exhibit similar code. We hypothesize from this observation that clone detection techniques might be ideal candidates for identifying CCC code, since they automatically detect duplicated code in the source code of an application.

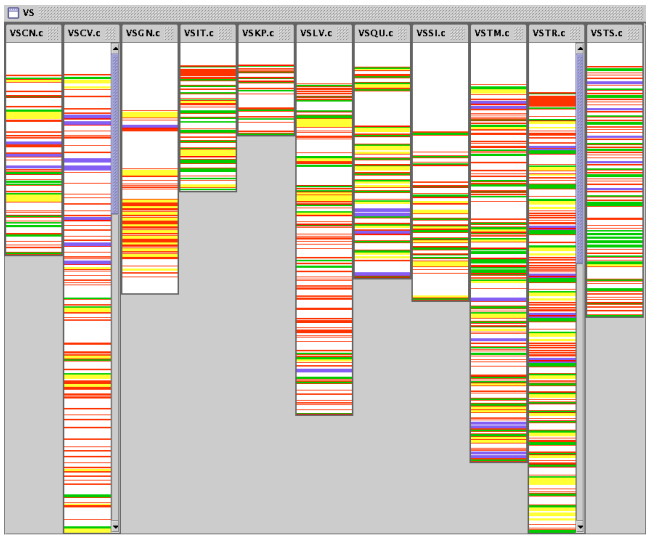
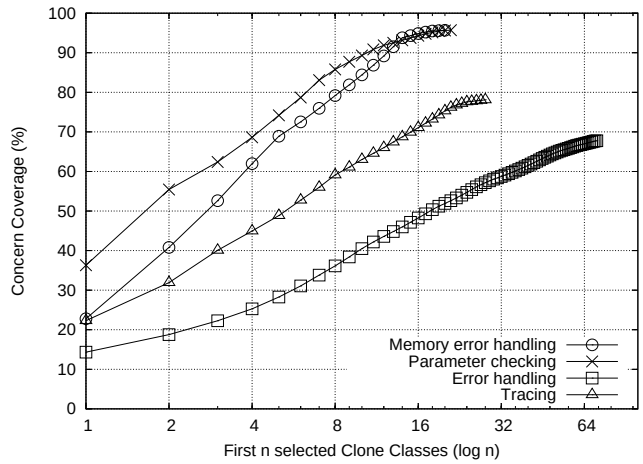


Figure 1. Cross-cutting concerns in a 19 KLOC component.

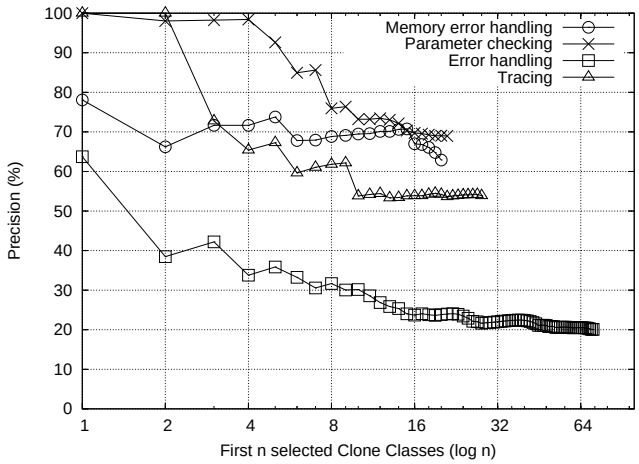
Case Study

Our experiment was performed on a software component (called *VS*) of 19,000 lines of C code, part of the larger code base (comprising over 10 million lines of code) of ASML. Developers at ASML currently use an idiomatic approach to implement CCCs. Consequently, similar pieces of code are scattered throughout the application, as can be seen in Figure 1, where each column represents a module and each color represents one of four CCCs. For now we focussed on CCCs dealing with tracing (green), pre and post condition checking (yellow), memory error handling (blue), and general error handling (red). All together, the CCCs we considered comprised roughly 31% of the code.

For the experiment an ASML developer manually annotated the source code lines of the 19 KLOC component, in-



(a) Concern coverage of the clone classes.



(b) Precision of the clone classes.

Figure 2. Clone detection results.

dicating for each source code line to which CCC it belongs. As a result, each CCC is defined by a set of source code lines. Subsequently, we use a clone detector to obtain *clone classes*, i.e. sets of code fragments that are cloned (similar).

In order to evaluate to what extent the clone detector meets our goal of identifying CCC code, we investigated the level of *concern coverage* met by the clone classes. *Concern coverage* is the fraction of source code lines of a concern that occur in the clone classes, i.e. the source code lines that occur as part of one of the cloned code fragments. Conversely, we evaluate the *precision* obtained by the clone classes. *Precision* is defined as the ratio of concern lines occurring in the clone classes, to the total number of lines occurring in the clone classes.

We are primarily interested in achieving sufficient coverage without losing too much precision. Therefore, we focus on the number of clone classes needed to cover most of a concern, where we consider 80% to be a sufficient coverage level.

Results

Figures 2(a) and 2(b) show the coverage and precision levels obtained by the clone classes. The horizontal axis represents those sets of n clone classes, that each obtain the highest possible coverage for a CCC. Vertically, we have the level of coverage in Figure 2(a) and the level of precision in Figure 2(b). In this paper we only present the results of a clone detector that implements Baxter's AST-based clone detection algorithm.

Observe that as the number of clone classes increases, the coverage displays a monotonic growth, whereas the precision tends to decrease. The highest coverage is less than 100% in all cases: the remaining percentage corresponds to concern code that is coded in such a unique way that it does not occur in any clone class.

The parameter checking and memory error handling concerns are covered sufficiently well (80%) using a limited number of clone classes. Furthermore, the level of precision obtained by the clone classes in the parameter checking case is very high: the first 4 selected clone classes obtain a precision as high as 98%. In contrast, the tracing and error handling concerns do not show the same results. Both concerns are not covered sufficiently, although the coverage of tracing is almost sufficient (78%). Additionally, the obtained precision is low. An exception is the precision obtained for the tracing concern by the first 2 clone classes: they cover 32% of the tracing code at 100% precision. However, if more clone classes are considered the precision degrades quickly.

Conclusion

Evaluating the clone classes in terms of concern coverage and precision identified many clone classes that will be of value during our research within the context of the ASML code base. The component we considered is a small, but representative, example of the components that exist within this code base. Therefore, the clone classes that we found to cover large portions of CCC code will probably be of great use for identification of CCC code in the other components.

In the more general sense, we intend to work toward a fully automatic "concern mining" tool. Based on the results presented in this paper, we can conclude that clone detection techniques are capable of characterizing large portions of the code of certain CCCs. However, in our experiment we included an important manual step; the knowledge of an expert is used to identify the CCC code. Further experiments will be needed to show that clone detection techniques can

also find CCC code using a more limited amount of expert knowledge.