On the Role of Program Comprehension in Embedded Systems

Janet Feigenspan, Norbert Siegmund, Jana Fruth
University of Magdeburg, Germany

Today, we are surrounded by computers. However, only a minor part is working stations we might think of. The major part, about 98 %, are embedded systems [15], for example PDAs, mobile phones, sensors, or credit cards. For embedded systems, resource constraints regarding memory capacity and performance are characteristic. Furthermore, the hardware is heterogeneous, which leads to challenges regarding how to tailor software to actual hardware of application scenarios [10].

In practice, embedded systems are typically implemented in C using conditional compilation with the C preprocessor. Conditional compilation allows users to customize software to new hardware without having to implement or adapt source code. To illustrate this, we show a source-code excerpt of BerkeleyDB[1], an embedded data base, in Figure 1. We can configure BerkeleyDB to run on several systems without having to change or introduce new source code. For example, in Line 5, we see an *ifdef directive* `#ifndef`, followed by a variable `HAVE_QUEUE`. This means that if `HAVE_QUEUE` is not defined, the following 3 lines are included by the preprocessor. If `HAVE_QUEUE` is defined, then the lines are deleted and, as defined by the *ifdef directive* `#else` in Line 9, the following lines (10–16) are included instead. Hence, by simply specifying variables, we can adapt BerkeleyDB to run on different hardware, different operating systems (e.g., Windows or Linux) or for different application scenarios (e.g., with or without transaction support), without changing the source code.

Although preprocessors are widely used in practice, in the literature, they are considered 'harmful' [13] or even as '#ifdef hell' [6]. Arguments against the preprocessor are based on the fact that we can annotate everything, even single variables or opening brackets without the corresponding closing one. Hence, it can be very difficult for a programmer to get an overview of source code that is annotated with *ifdef directives*. Specifically, they can be (i) scattered, (ii) 'hidden', (iii) nested, or (iv) used to annotate long code fragments. For example, if we want to implement a logging mechanism, this means that we have logging source code in many different locations in different files. Hence, according *ifdef directives* are scattered, as well. As another example, in Figure 1, Line 16 states that over 100 additional lines of code are defined, before the `#ifndef` in Line 5 is closed with the corresponding `#endif`. Hence, beginning and end of an *ifdef directive* may not even appear on the same screen, which makes it difficult for a developer to keep track of it.

```
1  static int __rep_queue_filedone(dbenv, rep, rfp)
2    DB_ENV *dbenv;
3    REP *rep;
4    __rep_fileinfo_args *rfp; {
5   #ifndef HAVE_QUEUE
6     COMPQUIET(rep, NULL);
7     COMPQUIET(rfp, NULL);
8     return (__db_no_queue_am(dbenv));
9   #else
10    db_pgno_t first, last;
11    u_int32_t flags;
12    int empty, ret, t_ret;
13    #ifdef DIAGNOSTIC
14      DB_MSGBUF mb;
15    #endif
16  // over 100 lines of additional code
17   #endif
18  }
```

Figure 1: Code excerpt of Berkeley DB.

Understanding the source code of a program is crucial for many important tasks. We may introduce security leaks, because we do not understand portions of the source code. We may write bugs more often, because we do not understand the program flow completely. We may waste energy of the embedded system, because we cannot efficiently utilize the hardware due to the effort of writing variable code specific for some hardware components. All these problems increase maintenance costs and cause risks for reliability and security.

To address comprehensibility issues of preprocessor usage, we have to measure program comprehension. To define suitable measures, we first have to understand what program comprehension is. In the literature, we can find three different categories of program-comprehension models: Top-down models, bottom-up models, and integrated models. Top-down models describe how developers derive a general hypothesis of the purpose of the program and then refine it stepwise by examining source code in more and more detail [3, 8, 11]. Bottom-up models describe how a programmer examines statements of a program and groups them into semantic chunks. These chunks can be combined further until the developer has an understanding of the general purpose of a program [7, 9]. Typically, a developer uses both approaches: Top-

[1] `http://www.oracle.com/technetwork/database/berkeleydb`

down, if she is familiar with a program's domain (e.g., she does not have to examine a method called `quickSort` statement by statement, because she knows from the name what it does), and bottom-up, if she has no hint what a program does [16].

Hence, program comprehension is an internal cognitive process that cannot be observed directly. Instead, we have to find indicators to measure it. Typically, (i) software measures, (ii) self estimation, (iii) tasks, and (iv) think-aloud protocols are used. First, software measures, such as lines of code or cyclomatic complexity are believed to have a link to program comprehension. They are calculated based on a properties of source code and do not require any human subjects. Hence, they are easy to apply, however, not very reliable, because their link to program comprehension is not empirical validated [2]. Software measures should only be used in combination with the other three techniques, because they recruit human subjects. Second, using self estimation, subjects are asked how much they think they understood of source code. This is closer to the real comprehension process, however it can easily be biased [4]. Third, tasks are often used to measure program comprehension. For example, in maintenance tasks, subjects are asked to locate and/or fix a bug [4]. Since to fix a bug, subjects have to understand the underlying source code, a successful bug fix can only be based on a comprehension process. Last, in think-aloud protocols, subjects verbalize their thoughts [12]. This way, we can observe the comprehension process itself.

To profit from the benefits of preprocessor usage and handle the introduced problems at the same time, we have to take care of the comprehensibility issues. This way, we support the maintenance of preprocessor-based software, in that we save time and cost. Typically, a maintenance programmer spends up to 60 % of her time with understanding source code [14, 17] and the cost for maintenance are the major contributing factor for the software development [1]. Hence, if we can improve comprehensibility of source code, we can reduce the time and cost developers need to maintain source code and thus allow developers to spend more time improving reliability and security of source code.

Now that we can measure program comprehension, we can start improving it. An example can be found in [5]. In this approach, background colors are used to highlight source code that is annotated with *ifdef directives*. For example, in Figure 1, Lines 5 to 17 would be annotated with a background color and Lines 13 to 15 with another background color. The effectiveness of background-color usage was shown with an experiment, in which subjects should solve comprehension tasks.

Acknowledgements

Feigenspan's, Siegmund's, and Fruth's work is funded by the German Ministry of Education and Science (BMBF), project 01IM08003C.

References

[1] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[2] J. Boysen. *Factors Affecting Computer Program Comprehension*. PhD thesis, Iowa State University, 1977.

[3] R. Brooks. Using a Behavioral Theory of Program Comprehension in Software Engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 196–201. IEEE CS, 1978.

[4] A. Dunsmore and M. Roper. A Comparative Evaluation of Program Comprehension Measures. Technical Report EFoCS 35-2000, Department of Computer Science, University of Strathclyde, 2000.

[5] J. Feigenspan et al. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Proc. Int'l Conf. Evaluation and Assessment in Software Engineering (EASE)*, pages 66–75, 2011.

[6] D. Lohmann et al. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proc. Europ. Conf. Computer Systems (EuroSys)*, pages 191–204. ACM Press, 2006.

[7] N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychologys*, 19(3):295–341, 1987.

[8] T. Shaft and I. Vessey. The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research*, 6(3):286–299, 1995.

[9] B. Shneiderman and R. Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Parallel Programming*, 8(3):219–238, 1979.

[10] N. Siegmund et al. Challenges of Secure and Reliable Data Management in Heterogeneous Environments. In *Proc. Int'l Workshop on Digital Engineering*, pages 17–24. ACM Press, 2010.

[11] E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, 1984.

[12] M. Someren, Y. Barnard, and J. Sandberg. *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*. Academic Press, 1994.

[13] H. Spencer and G. Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, pages 185–198. USENIX Association, 1992.

[14] T. Standish. An Essay on Software Reuse. *IEEE Trans. Softw. Eng.*, SE–10(5):494–497, 1984.

[15] D. Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, 2000.

[16] A. von Mayrhauser and A. Vans. From Program Comprehension to Tool Requirements for an Industrial Environment. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 78–86. IEEE CS, 1993.

[17] A. von Mayrhauser, A. Vans, and A. Howe. Program Understanding Behaviour during Enhancement of Large-scale Software. *Journal of Software Maintenance: Research and Practice*, 9(5):299–327, 1997.