Sixth FTW PhD Symposium
Faculty of Engineering, Ghent University
5th November 2005
Aspect Orientation in the Procedural Context of C

Ir. Bram Adams
Promoter(s): Prof. Dr. Ir. Ghislain Hoffmann, Prof. Dr. Ir. Herman Tromp

Abstract—Since the rise of Information Technology, companies have invested fortunes in software systems, to the extent that they are now totally reliant on them. Paradoxically, as time goes by and business requirements are adjusted every day, fewer people seem to know how and why this legacy software still works. Aspect Oriented Programming (AOP), an emerging programming paradigm, has been identified as an important technique to aid in re-engineering these systems, because it modularizes crosscutting concerns without actually modifying the original source code ("obliviousness"). Our research focuses on the integration of AOP in C. To get a handle on current industrial conditions, we applied our new aspect language (called Aspicere), tools and the AOP methodology to a realistic case study. This provided us with valuable feedback regarding common legacy C practices and integration issues of AOP tools to existing build processes.

Keywords—AOP, legacy systems, C

I. INTRODUCTION

Having invested massively in the digital revolution, the industry now faces the dramatic consequences. Indeed, in an attempt to become more competitive, companies have virtualized most of their (critical) business processes in costly software systems. The initial intent was to rule out as much human errors and indeterminism as possible. However, to recover from these large expenses, the acquired systems need to serve far more years than originally foresen. In the meantime, they must manage somehow to cope with new technologies like the Internet, web services, …

Unfortunately, the initial designers and programmers of these “legacy” systems have long since vanished. The current maintainers do not understand large portions of the code and they do not want to touch it either, as the company’s core operations rely on it. To fulfil new requirements, however, billions of lines of code are still added each year to these existing code silos.

Due to the critical nature of the legacy problem, new technologies like Aspect Oriented Programming (AOP) are evaluated in light of this. In the remainder of this paper, we will elaborate on AOP and its implications on procedural languages like C and the legacy problem (section II). Then, we move on to our aspect language, Aspicere1, in section III and its application to a realistic case study in section IV. We conclude with a summary of our current findings in section V.

II. ASPECT ORIENTED PROGRAMMING

Originally, AOP has been developed to overcome apparent shortcomings in the Object Oriented (OO) world [1]. When mapping a system design to source code, a certain class structure has to be chosen. Common OO practices recommend to decompose things from the perspective of the main concern, i.e., with regard to the actual objectives of the whole system. If a bank account system is developed, typical business objects are customers, accounts, saving formulas, … Additional concerns like security, persistence, logging, … however, are not modularized well in this skeleton structure, but have to be added somewhere inside the main decomposition. Typical properties of these systems, are: scattering Invocations of these auxiliary concerns are spread throughout the whole system.

Tangling Main concern code is intermingled with other concerns’ implementations, making things hard to keep track of.

A scattered concern tangled between another concern, is said to be “crosscutting” with respect to the latter (the reverse does not necessarily hold). It is important to notice that crosscutting is inherent to the modeling of concerns and as such is not tied to OO systems. The legacy systems and languages of section I face the same problems.

AOP not only identifies the crosscutting problem, it also proposes a new methodology to resolve these issues. Briefly, instead of manually invoking a crosscutting concern’s code in the constructed program (the “base” program), the concern’s core functionality is encapsulated into its own module (an “aspect”): the base program is “oblivious” to this concern. Of course, this aspect code should be linked to (“woven in”) the base program somehow, guided by conditions specified on the base code (“quantification” by “pointcuts”). If these are met during the execution of the base program (“join point”), the relevant crosscutting code (“advice”) inside an aspect is automatically invoked.

Applying these ideas to legacy languages like Cobol and C, we get a much cleaner separation of concerns, resulting in easier to understand software systems. Equally important is the obliviousness of aspects. Prior to re-engineering an application, one can apply aspects to it to gather all kinds of information for reverse engineering without modifying any code.

As a first step in exploring the interaction between AOP and traditional, procedural languages, K. De Schutter designed the AspectCobol language [2]. Building further on the resulting framework, our aspect language Aspicere tries to apply the AOP ideas on C. We will explore this in the next section.

III. ASPICERE

A. Join point model

The first thing to consider when designing an aspect language, is an appropriate join point model: the circumstances when and where aspects are allowed to interact with the base program. In procedural programming languages like C, procedure calls and variable accesses seem prime candidates for this. We can interpret both variable reads and writes as procedure invocations, which simplifies everything.
C. Weaving framework

IKE Cobble [2], Aspicere’s weaver is a preprocessor to a normal compiler. Preprocessed C code is transformed into a more manageable XML representation. Using a Prolog engine and XPath, we match join points with the relevant advice, before modifying the XML-structure using DOM-manipulation. Finally, everything is transformed again to (woven) source code and fed into a C compiler.

IV. CASE STUDY

To evaluate our language and weaving prototype, we applied them to an industrial code base of 407 C modules, ported from an ancient architecture to a more modern Linux platform [5]. As an initial experiment, we applied Fig. 1’s tracing aspect.

It turned out that the largest part of the code was still written in traditional (pre-ANSI) C. This means e.g. that procedure declarations (in header files) don’t need to specify any arguments yet. As a consequence, our weaving needs to guess the right argument types from the calling context. Due to C’s complexity, this is not that trivial. That is why we currently need to “skip” certain join points if we can’t make up the right typing information.

Aside from this complexity, our weaver’s join point matching slows the weaving process considerably down. Scalability obviously needs closer examination.

Finally, because our weaver serves as a preprocessor to a real C compiler, the existing makefile hierarchies are not oblivious to our tools. Automatically patching the build scripts should address this issue.

V. CONCLUSIONS

WE argued why legacy systems could be killer applications for AOP, based on obliviousness and quantification. Then we mapped the AOP ideas on the C language and briefly explained Aspicere, our aspect language for C. All work thusfar has been evaluated on a realistic case study, of which we showed the most relevant results. Our weaving needs to be refined more before performing a second, more complex case study.

ACKNOWLEDGEMENTS

We would like to dedicate this paper, in memoriam, to professor dr. ir. Ghislain Hoffman, who passed away on August 25th, 2005. The author also wants to thank Kris De Schutter and professor dr. ir. Herman Tromp for their support and knowledge sharing.

REFERENCES