IS ACADEMIC OPEN SOURCE SOFTWARE DEPENDABLE?

Shigeru Chiba

Tokyo Institute of Technology, W8-50, Ohkayama 2-12-1, Meguro-ku, Tokyo 152-8552, Japan
chiba@acm.org

Abstract I have been developing several open source products as part of my academic research projects and a few of them has obtained a large number of users. In this paper, I present my experience of such open source software, OpenC++ and Javassist, and mention my thought about whether or not academic open source software can be dependable.

Keywords: Java; C++; Open Source Software.

1. My experience of OpenC++

OpenC++ [3, 1] is an extensible C++ compiler based on computational reflection [10]. This software has two versions: version 1 and version 2, which are quite different software products although they share the same name.

OpenC++ version 1 [3] released in around 1993 can be regarded as a language providing a mechanism called runtime metaobjects. The runtime metaobjects are objects that can intercept method calls and field accesses to other objects. This ability is useful for implementing a number of language extensions such as dependability, distribution, and persistence. OpenC++ was one of the first languages that provided runtime metaobjects with semi-production quality and have been widely used as a research platform for implementing various language extensions by using metaobjects [6, 11].

Fabre et al [6] used OpenC++ version 1 for implementing automatic object replication, which significantly improves dependability of software against host failure. Their idea was that metaobjects intercept all method calls to mission-critical objects and send copies of the method calls to corresponding backup objects running on different hosts. If that mission-critical objects become down, the backup objects are substituted for the original objects. Because of OpenC++, this replication mechanism could be easily implemented and thus the developers of that mechanism could focus their concerns on higher-level issues such as replication policies and execution performance.

OpenC++ version 2 [1] was released in 1996. It is a toolkit of source-to-source translator in C++ although the programming interface was constructed with the concept of metaobjects. From the academic viewpoint, OpenC++ version 2 was a system using a technique called compile-time reflection, which dramatically reduces runtime penalties due to metaobjects, compared to others in particular OpenC++ version 1.

OpenC++ version 2 has been widely used. Since 2000, more than 6,000 copies of this software have been downloaded by both academia and industry. For example, Fabre's group again used OpenC++ version 2 and developed a checkpointing system for improving dependability [7].

Although OpenC++ was an academic product, its quality was acceptable enough to build real applications. However, maintaining OpenC++ to be compatible to the latest specifications of the C++ language needed tremendous efforts. In particular, supporting C++ templates was a hard task and thus OpenC++ has not fully supported C++ templates yet; OpenC++ can process a program using templates but metaobjects cannot intercept template classes. The limitation of development resources is a major problem of academic open source software.

OpenC++ version 2 is now on <http://opencxx.sourceforge.net> and it is maintained by a group of volunteers since 2003. This is mainly because it has been getting difficult more and more for me to have time to maintain OpenC++ these years. Since the goal of academic software development is to prove that some idea is realistic and right, maintaining software after the research project finished is usually difficult.

2. My experience of Javassist

Javassist [2, 5] is a bytecode engineering toolkit for Java. The development of this software started in 1998 as a research project of reflective computing in Java and this software was first released in 1999. We are currently using this software as a research platform of aspect-oriented programming systems [9,4, 8].

A unique feature of Javassist is that it allows the developers to manipulate a class definition (i.e. a class file) at the bytecode level without detailed knowledge of the specifications of Java bytecode. They need only the knowledge of the Java language. Javassist provides source-level abstraction, through which the developers can manipulate bytecode. Javassist automatically translates class-file manipulation described at source-level vocabulary into the equivalent manipulation at bytecode level.

A typical application of Javassist is to implement a component providing a non-functional concern such as persistence and synchronization. Using such a component is troublesome since developers must carefully insert a number

of method calls in the client components so that the client components can appropriately call methods in that component. Javassist can be used for automatically inserting such method calls. This automation can avoid wrongly inserting method calls and thus decreasing stability and dependability of the software.

Since 2000, more than 6,800 copies of Javassist has been downloaded. Recently, Javassist recorded more than 500 downloads per month in the fourth quarter in 2003. The development of Javassist since 2003 is being hosted under the umbrella of JBoss.org, which is an open source project developing the JBoss J2EE server and several complement products. JBoss is a commercial-quality open source software and it has the third biggest share in the J2EE server market. Since JBoss internally includes Javassist and it has more than 7,000 downloads per month, the number of the Javassist users is huge if such indirect users are included.

The relatively larger user base of Javassist than OpenC++ worked well for improving the quality of the software. In general, a larger user base can find a larger number of bugs so that it can improve the stability of the software. A few people among the Javassist users have contributed source code to Javassist for extending the functionality.

3. Academic open source

An academic project is a good framework for developing innovative software. It is usually easy to release the resulting products to the public as open source software. Releasing open source software is rather motivated in an academic project since the goal of academic project is often to propose a new idea and convince people that the idea is right and useful. Open source software is a great promotion tool for showing the idea to people.

On the other hand, an academic project is not a good framework for maintaining open source software and improving the quality of that software. Developing a commercial-quality software is not a goal of most academic projects. If a primary developer of the software is a student or a post-doctoral researcher, she would leave the project in a few years. Moreover, the lifetime of the project itself is usually less than five years. Thus, keeping the developers in the team and continuing maintenance is not easy.

Some might think open source software could be maintained by volunteers. In fact, OpenC++ is currently maintained by a group of volunteers. However, volunteer-based maintenance is not sufficient to catch up the rapid evolution of the specifications of the C++ language, for example. The resources of the volunteer group is limited for maintenance. This is not surprising. While developing new software is exciting and it can attract talented developers, maintaining

existing software is usually boring and hence developers do not want to spend a large amount of their time.

I have an interesting experience. The OpenC++ mailing list sometime receives a message from people who are new to the project. They take a very quick look at the source code of the software and usually say that the style of the source code is very old-fashioned (they do not know that the source code was written more than eight years ago!) and hence the source code must be completely rewritten. Otherwise, they say that the software should be redesigned by using today's XML technology or something similar. However, if we say "OK, your suggestion is right. Can you kindly do that work and check new source code in the CVS?", then they say "Ah.. I don't have time to write code for your project (since I am a very important guy and hence busy). The only contribution I can do now is to give my (variable) consultation to the project." In most cases, this is their last message post to the mailing list.

To make academic open source software dependable, we need another kind of framework for the maintenance phase. An academic project can be a cradle of open source software but it cannot take care of the software after it grows up. Professional open source proposed by Marc Fleury of JBoss Inc. can be such a follow-up framework. In his framework, developers are paid by a company that is selling technical supports of their open source software. Thus they must spend 50% of their work time for maintenance and technical supports of the software. However, they can use another 50% for extending the software or developing a new version of the software. In other words, 50% for money and 50% for fun.

There are high-quality academic open source software. However, keeping that quality after the initial development is difficult. Since keeping the quality is mandatory to use open source software as part of highly dependable software, we need a framework, such as professional open source, for motivating developers to maintain the software.

Today, some industrial people regard open source software just as inexpensive software. However, the cost of using open source software would go up if the software can be used as a dependable component. Dependability is expensive. The users must pay directly or indirectly developers of the software so that the developers can be motivated for the maintenance.

References

- [1] Chiba, S., "A Metaobject Protocol for C++," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices vol. 30, no. 10, pp. 285–299, ACM, 1995.
- [2] Chiba, S., "Load-time structural reflection in Java," in *ECOOP 2000*, LNCS 1850, pp. 313–336, Springer-Verlag, 2000.

- [3] Chiba, S. and T. Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture," in *Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, pp. 482–501, Springer-Verlag, 1993.
- [4] Chiba, S. and K. Nakagawa, "Josh: An Open AspectJ-like Language," in *Int'l Conf. on Aspect Oriented Software Development (AOSD'04)*, pp. 102–111, 2004.
- [5] Chiba, S. and M. Nishizawa, "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators," in *Proc. of Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, pp. 364–376, Springer-Verlag, 2003.
- [6] Fabre, J., V. Nicomette, **T. Parnenou**, R. J. Stroud, and Z. Wu, "Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming," in *Proc. of the 25th IEEE Symp. on Fault-Tolerant Computing Systems*, pp. 489–498, 1995.
- [7] Killijian, M. O., J. C. Ruiz-Garcia, and J. C. Fabre, "Portable Serialization of CORBA Objects: A Reflective Approach," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices vol. 37, no. 11, pp. 68–82, 2002.
- [8] Nishizawa, M., S. Chiba, and M. Tatsubori, "Remote Pointcut – A language construct for distributed AOP," in *Int'l Conf. on Aspect Oriented Software Development (AOSD'04)*, pp. 7–15, 2004.
- [9] Sato, Y., S. Chiba, and M. Tatsubori, "A Selective, Just-In-Time Aspect Weaver," in *Proc. of Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, pp. 189–208, Springer-Verlag, 2003.
- [10] Smith, B. C., "Reflection and Semantics in Lisp," in *Proc. of ACM Symp. on Principles of Programming Languages*, pp. 23–35, 1984.
- [11] Stroud, R. J. and Z. Wu, "Using Metaobject Protocols to Implement Atomic Data Types," in *Proc. of the 9th European Conference on Object-Oriented Programming*, LNCS 952, pp. 168–189, Springer-Verlag, 1995.