Separation of Distribution Concerns in Distributed Java Programming

Michiaki Tatsubori
Doctoral Program in Engineering, University of Tsukuba
Tennohdai 1-1-1, Tsukuba, Ibaraki 305-8573, Japan
mt@is.tsukuba.ac.jp

ABSTRACT
Distributed design decisions in a distributed Java program crosscut the module structure of non-distributed concerns in the program. We propose a tool supporting aspect-oriented distributed programming, with which programmers can specify the distribution aspect of program simply and separately from non-distributed Java program, for enhancing the modularity of program. The aspect-weaver of this tool is a bytecode translator implemented as a customized class loader of Java virtual machine (JVM). Thus the tool is applicable to code supplied by third-parties without source code and the resulting distributed program runs on regular JVMs.

Keywords
Distributed programming, bytecode translation, Java, load-time weaving, aspect-oriented programming

1. PROBLEM DESCRIPTION
Object-oriented modularization with classes often fails to encapsulate distributed design decisions that crosscut the module structure of non-distributed concerns in a distributed program. Code related to distributed concerns is often tangled with other code and scatters over a number of classes. Lack of modularization implies low maintainability of the program which makes programming difficult. For example, programmers often change the decomposing points of program in distributed environment for reducing the overhead of network communication. But such the change of design decision brings about changing remote or local object allocation code scattering over the program. Programmers have to modify a number of code pieces for implementing a simple change of distributed design decisions.

Design patterns[3] are useful for solving design problems but they are not almighty. Though programmers can modularize some part of their program using several programming techniques found in pattern catalogs, such the techniques often result in redundant code of program and are not always applicable. For example, a design based on the ABSTRACT FACTORY pattern allows programmers to write a centered code controlling instance allocation. They can change the policy of distribution of objects by overriding factory methods in a subclass of the Factory class. But, with this design, a class of the Factory role provides a number of factory methods for all the combinations of created classes and contexts creating instances. What redundant code they must write! Furthermore, they cannot edit the boot-strap classes of the Java virtual machine (JVM) or the classes supplied by third-parties without source code though implementing this design requires to do so.

2. GOAL STATEMENT
A distributed programming tool addressing this problem stated above is desired. Programmers should be able to specify distributed allocation of software components in a centered code separated from the non-distributed concerns. And this tool should be applicable to boot-strap classes and ones supplied by third-party.

The main goal of our proposing tool for supporting distributed programming in Java is as follows.

- The tool must preserve the portability of Java program so that Program developed with the tool should be able to run on existing regular Java virtual machines. The runtime support of distributed program should be written in Java.
- The tool must allow the programmers to easily specify whether each object is allocated in a distributed environment. For reducing the programing cost, the object allocation should be specified at an appropriate abstract level.
- The tool must hide implementation details of remote object references from the programmers. The programmers should not have to modify the program so that remote references in the program follow a particular protocol specified by the tool.

There is another contribution of this work. The proposing tool is to be a domain-specific aspect-oriented programming (AOP) tool. AOP is a programming paradigm addressing the problem of tangling code, with advanced separation of crosscutting concerns[5]. AspectJ is one of the most famous

AOP languages and its design is dedicated to a general purpose AOP support[4] in Java. If its generality were enough for separating the distribution concerns, we could build our tool as a translator which translates our domain-specific aspect language to the AspectJ language. It is impossible now and thus our experience contributes, besides to distributed programming, to researches for the generality of AOP languages like AspectJ. Generic purpose AOP languages should have enough writability for expressing the division of the distribution purpose aspects we propose.

3. APPROACH TO BE USED

An aspect language for distribution concerns and an aspect-weaver for this language is the key design of the tool we propose. The proposing tool, named *Addistant*, accepts distribution aspects besides code of components for non-distributed logic of program. Then it produces distributed Java program.

3.1 Aspect Language

Addistant allows the developers to specify a policy of object allocation for each class. They can use special language constructs for specifying a group of classes related closely. The language provides a specifier for classes in a package or sub-packages since they are often related closely and allocated on the same host. Also it provides a specifier for a subclass-tree for the same reason. For example,

```
<import ... from="display">
   subclass@java.awt.Component
</import>
```

means that all the subclasses of the class `Component`, including `Component` itself are allocated on a host specified by a variable `display`.

Also, Addistant provides several different techniques for implementing proxy-master model. Programmers can choose one of these implementation techniques for each class. The differences among these techniques are mainly how a proxy class is declared, how caller-side code, that is, expressions of remote method invocations, is modified, and how a master class is modified. For example, they can choose a suitable technique for boot-strap classes.

3.2 Load-time Weaving

In Addistant, weaving of aspects are performed at load time of classes. The runtime system weaves a distribution aspect into base Java code of program. This translation of the base code is implemented in a customized class loader for Java[6]. The class loader of Addistant modifies bytecode of classes according to the specification given as a distribution aspect, before loading classes on JVM.

The weaver of Addistant is built using Javassist[1], which provides a tool for bytecode editing and employs object-oriented structural reflection for Java[7].

3.3 Evaluation

For evaluating how our tool enhances the maintainability of distributed program, we need a new kind of metrics. Since our tool makes it unnecessary for programmers to write program code belonging to a distribution aspect, we must measure difficulty in writing that program code and thus how the programmers' efforts are reduced with our tool. Then we can compare our tool with other tools, which require programmers to write various amounts of program code for a distribution aspect.

Existing metrics for object-oriented design[2] are basically for measuring how the complexity of a program is reduced if the program is divided into several different aspects. However, the metrics that we need are, for example, how many lines or words are needed for a distribution aspect, how the program code for this aspect is spread over the whole program, and so forth.

4. STATUS

We have developed a prototype system which is designed with the proposing approach[8]. It is concentrated to separating the concern of distributed object allocations and the concern of proxy class implementation method. With this system, programmers can specify a simple distribution aspect for obtaining a distributed program with remote presentation from existing, non-distributed program, for example.

5. REFERENCES

[1] S. Chiba. Load-time structural reflection in Java. In *ECOOP 2000 – Object-Oriented Programming*, LNCS 1850, pages 313–336. Springer-Verlag, 2000.

[2] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *Proc. of OOPSLA'91*, ACM SIGPLAN Notices (26) 11, pages 197–212, 1991.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pages 327–353. Springer-Verlag, 2001.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming*, LNCS 1241, pages 220–242. Springer-Verlag, 1997.

[6] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proc. of OOPSLA'98*, ACM SIGPLAN Notices (33) 10, pages 36–44, 1998.

[7] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In *Reflection and Software Engineering*, LNCS 1826, pages 119–135. Springer-Verlag, 2000.

[8] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" Java software. In *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pages 236–255. Springer-Verlag, 2001.