# Jarcler: Aspect-Oriented Middleware for Distributed Software in Java

Muga Nishizawa Shigeru Chiba
Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
Email: {muga,chiba}@csg.is.titech.ac.jp

## ABSTRACT

This paper proposes Jarcler, which is aspect-oriented middleware for using replicated objects in Java. It enables the users to customize the behavior of replicated objects per class so that the behavior fits requirements of a particular application. Although reflection has been a typical technique for customizing such behavior, this paper shows that reflection forces programmers to write a program far from their intuition; aspect-oriented programming provided by Jarcler makes it easier to describe the customization. This paper illustrates this issue through an example of simple network game.

## 1. INTRODUCTION

Making it easy to develop distributed software is one of serious demands in today's software industry. To do that, several middleware and tools have been developed. In Java, the Java RMI (Remote Method Invocation) framework is provided as part of the standard. It simplifies stub-code generation. From the research community, several tools such as Addistant developed by us [15] and J-Orchestra [16] have been proposed for making remote object reference transparent. They automatically transform a program so that local and remote references can be indistinguishable in a program.

Although the tools like Addistant significantly reduce development costs of distributed software, our experience with Addistant revealed that those tools cover only part of distributed computing. For example, a number of applications require a mechanism of distributed shared data whereas those tools do not directly support that mechanism. Although other systems like JavaSpace [8] provides distributed shared data for Java, they are not integrated with Addistant.

To introduce a mechanism of distributed shared data into a tool like Addistant, a major research issue is how the tool enables the users to customize the behavior of the distributed shared data. Since different applications often require different behavior, the customizability is indispensable in practice. Tools like Addistant do not need customizability since most of applications require only normal behavior with respect to remote object reference and remote method invocation.

This paper presents our challenge to this issue. We have been trying to extend Addistant to support distributed shared data. This extended version of Addistant, which we call *Jarcler*, addresses this issue with aspect-oriented programming (AOP). This paper shows our ideas and the design of Jarcler. Although a well known technique for addressing this issue is meta programming based on reflection [13, 12], it is often far from the users' intuition and difficult to use. This paper presents that our AOP-based solution is more appropriate than reflection for customizing the behavior of distributed shared data. Jarcler provides a special language for writing an aspect; this paper also mentions why we need a special aspect language instead of a general-purpose AOP language such as AspectJ [10].

In the rest of this paper, Section 2 briefly describes distributed shared data that we want to provide and discusses a drawback of using reflection for customizing the behavior. Section 3 presents our tool named Jarcler and mentions how it allows the users to customize the behavior in an aspect-oriented way. Section 4 discusses related work. Finally, Section 5 concludes this paper.

## 2. DISTRIBUTED SHARED DATA

Distributed shared data is a communication and synchronization mechanism among distributed processes [1]. A famous instance of distributed share data is the tuple space [9]. A C++ object on distributed shared memory [11] is another instance of distributed shared data.

### 2.1 Replicated Object

In Java, typical distributed shared data is a replicated object, which is implemented as a set of replications of an object on different hosts. To keep consistency among the replications, the underlying system monitors updates of the field values. If a field value of a replication is updated, then the update is propagated to the other replications to be reflected on every replication.

Since replicated objects are used for not only communication but also synchronization, they must meet various requirements of applications. For example, some applications may need to atomically update the values of multiple fields together. Others may require executing a notification method on every replication when a particular field value of the replicated object is updated. Furthermore, they may need to suppress or delay propagating updates of some fields to the other replications.

To meet these requirements, we decided to give customizability to our tool called *Jarcler*, which enables the use of replicated objects in Java. Jarcler transforms a program so that replications are automatically created on every hosts when a program attempts to create a replicated object. If a method is called on one of the replications, then that method call is propagated to the other replications to be consistently updated (Figure 1). Note that an update of field value is not propagated whereas a method call is propagated.

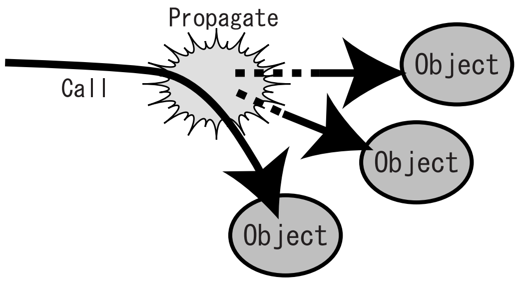


**Figure 1: A Replicated Object**

Our basic design of replicated object by Jarcler is obviously too naive to use in practice. For example, propagating all method calls is inefficient since some method calls never change field values. If a method causes external side-effects, for example, if it calls a method on another object $obj_{ex}$, then the behavior of replications should be controlled so that only one replication calls the method on the object $obj_{ex}$ to avoid redundant method calls. To address these problems, Jarcler provides the behavior mentioned above as the default one while it allows the users to customize the algorithm for propagating method calls.

## 2.2 Tic Tac Toe

Our challenge is how we should design the ability to customize the propagation algorithm. Before discussing this issue, we present a simple example of replicated object, which we will use for discussion in the rest of this paper. Suppose that we are developing a Tic-Tac-Toe game in which two players can play through a network. This distributed software consists of three components: a game server and two GUI clients, which are running on different hosts. The game server is responsible for letting each player alternately play and for determining which player wins. The GUI clients are responsible for showing a game board and interacting with the players.

The three components share the data structure representing a game board. Let that data structure be an array of nine `Square` objects, each of which represents a square on the game board.

```
class Square {
  char mark;         // 'O', 'X', or ' '
  int pos;           // position
  static char turn; // 'O' or 'X'
  TicTacToe ttt;     // the game server

  Square(TicTacToe t, int p) {
    ttt = t; pos = p; mark = ' '; turn = 'O';
  }

  void nextTurn() {
    turn = (turn == 'O') ? 'X' : 'O'; }
  void setMark(char c) { mark = c; }
  char getMark() { return mark; }

  void clicked() {
    setMark(turn);
    ttt.pressed(pos);
    nextTurn();
  }
}
```

The `Square` class has a field named `mark`, the value of which is either `'O'`, `'X'`, or `' '` (blank), and a field named `pos` to represent the position of the square on the game board. If the player clicks a square on the screen, the GUI object calls `clicked()` on the corresponding `Square` object. The `clicked()` method changes the value of the `mark` field and notifies a `TicTacToe` object of this event. The `TicTacToe` object is on the game server.

Since the `Square` object is a replicated object, a replication of this object is allocated on every host at the implementation level. To maintain consistency among the replications, if `setMark()` and `nextTurn()` are called on one replication, then they must be executed on all the replications. However, since `getMark()` does not update field values, `getMark()` should be executed only on the host where the caller exists. The behavior of `clicked()` is more complex. If `clicked()` is called on the GUI-client host, then it must be executed not on that host but on the game-server host where the `TicTacToe` object exists. The reference value in field `ttt` would not be valid on the GUI-client hosts. Furthermore, the execution of `clicked()` on the game-server host must be performed only if the current turn is of the player who clicked on the screen.

## 2.3 Reflection

To satisfy these complex requirements above mentioned, the algorithm for keeping consistency among replications must be customizable. Reflection is a well known technique to address this problem [5]. In a system using reflection, a program is translated at compile time (or load time) so that every replication is associated with an object called a *metaobject*. The metaobject can trap a method call on the replication and alter the behavior of that call.

To implement the replicated `Square` object, if the metaobject traps a method call, then it executes the method on the replication while it sends messages through a network to the metaobjects associated with the other replications. These metaobjects execute the method on those replications. If the trapped method is `getMark()`, the metaobject does not send messages but only executes the method on the replication. If the trapped method is `clicked()`, then the metaobject sends a message to the metaobject on the game-server host so that `clicked()` is executed on that host.

A problem of this technique is that writing a metaobject is not simple or easy to read. For example, the definition of the metaobject for the class `Square` would be something like this:

```
class SquareMetaobject extends Metaobject {
  char myMark;  // 'O' or 'X'

  Object trapMethodCall(Object target, Method m,
                        Object[] args) {
    String name = m.getName();
    if (name.equals("netxTrun")
        || name.equals("setMark"))
      invokeOnOthers(m, args);
    else if (name.equals("clicked")) {
      if (myMark == Square.turn)
          return invokeOnGameServer(m, args);
      else
          return null;
    }

    return m.invoke(target, args);
  }

  void invokeOnOthers(Method m,
                      Object[] args) { ... }
  void invokeOnGameServer(Method m,
                          Object[] args) { ... }
      :
}
```

The metaobjects are instances of the class `SquareMetaobject`. The super class of `SquareMetaobject` is `Metaobject`, which is provided by the reflection system as the root class of all the metaobjects. If a method $m_b$ is called on a `Square` object, then `trapMethodCall()` is called on the corresponding instance of `SquareMetaobject`. The parameters `m` and `args` represent the called method $m_b$ and the actual arguments to that method $m_b$. The resulting value of `trapMethodCall()` is the result of the method call.

If the called method $m_b$ is `setMark()` or `nextTurn()`, then the metaobject sends messages containing `args` to the other metaobjects by `invokeOnOthers()` so that the method $m_b$ is also executed on the other replications. Then it executes the called method $m_b$ on the local `Square` object. The method `invoke()` in `Method` performs normal execution of the method. If the called method $m_b$ is `getMark()`, then the metaobject does not send a message but it just executes the called method $m_b$ by calling `invoke()`.

If the called method $m_b$ is `clicked()`, then the metaobject first compares `Square.turn` and `myMark` to determine whether the current turn is the player's. `Square.turn` represents the current turn. We assume that the value of `myMark` (`'O'` or `'X'`) is the mark of the player using the `Square` object corresponding to the metaobject. If `Square.turn` is equal to `myMark`, then the metaobject sends a message containing `args` to the game-server host so that `clicked()` is executed on that host.

Interesting features of the definition of `SquareMetaobject` shown above are the followings:

- The definition of `trapMethodCall()` is too long since all the behavior of method calls is described in that method, and
- Basic operations such as method calls and field accesses are described with different syntax. For example, a method call:
  ```
  ((Square)target).setMark('X')
  ```
  is described in a metaobject with different syntax:
  ```
  m.invoke(target, new Object[] {
                      new Character('X') })
  ```
  where `m` is a `Method` object representing the `setMark()` method.

  If the former syntax were used in `trapMethodCall()`, the method call would be recursively trapped by the metaobject and then it would result in infinite regression. Suppressing method-call interception by metaobjects within `trapMethodCall()` is an ad-hoc solution; a well-designed reflective system never adopts such a solution [4].

These features are defects in our example. However, this fact does not mean that reflection is poor technology since these features are pragmatically useful in other examples. The basic architecture of reflection is for defining altered behavior that can be *generically* applied to various methods and fields. For example, methods declared in a class differ from each other with respect to the name, the number and types of parameters, and the body. With reflection, however, all the methods are represented by a `Method` object so that a program can deal with all the methods in the same form; all the differences are abstracted away. Parameters to a method are represented by an array of `Object` for the same reason. If some parameters are primitive types such as `int` and `double`, they are converted into wrapper classes such as `Integer` and `Double` so that those parameters can be stored as well as other `Object` type parameters in the array of `Object`.

Although reflection is suitable for describing customized behavior common to a number of methods, it is not suitable for describing the behavior differently customized per method. Since the customization for replicated objects is the latter "fine-grained" one, reflection is not the best solution.

## 3. JARCLER

Since reflection is not suitable for customizing the behavior of a replicated object, we have developed a new mechanism based on aspect-oriented programming. Our tool *Jarcler*, which provides a replicated object in Java, allows the users to customize the behavior of the replicated object with this mechanism.

### 3.1 Aspect

To customize the behavior of a replicated object, the users write a description separated from the rest of the program including the `Square` class. We below call this description *an aspect of Jarcler*, or for short, *an aspect*.

In an aspect, the users can specify the followings:

- the class of replicated objects,
- the hosts that replications are allocated on, and
- the algorithm of consistency control among replications.

An aspect starts with keyword `replicate`, which is followed by a class name. An instance of this class is treated as a replicated object. We below show a simple example of aspect, which makes a `Point` object a replicated object. Suppose that the `Point` class has two methods `move()` and `getX()`.

```
1: replicate Point {
2:   hosts "A","B";
3:   int getX() {
4:     return this@local.getX();
5:   }
6: }
```

The line 2 specifies that the replications of a `Point` object are allocated on hosts named `A` and `B`. If an instance of `Point` is created on one of those hosts, the runtime system of Jarcler automatically creates another instance of `Point` on the other host. The two instances, which we call *replications* of a replicated object, are independent of each other; different values can be stored in the same field of the two instances.

By default, to maintain consistency among the replications, a method call on a replication is automatically propagated to all the other replications.[1] Hence, if the method `move()` is called on a replication, that method is executed on all the replications. However, the users can customize this behavior by a method declaration in an aspect, which specifies how the consistency is maintained when a method with the same signature is called on the replication. In the example shown above, the behavior of calls to `getX()` is altered (Line 3 to 5).

A method declaration in an aspect is similar to an around advice of AspectJ. The join point is method invocation on a replication. The method body of `getX()` in the aspect is executed instead of the body of the method `getX()` in the class `Point`. With respect to the name scope, a method declaration in an aspect is treated as if it is in the corresponding class. It can access other methods and fields in the same class. `this` is a reference to the corresponding replication. Unlike reflection, special syntax is not needed to access other fields and methods.

Within the method declaration, special syntax `this@`*hostname* is available for representing a replication on the specified host. This corresponds to `proceed()` of AspectJ. For example, the following declaration implements the default behavior for `move()`:

[1] In the current implementation, field accesses are not propagated. If a new value is directly stored in a field, the consistency among replications is broken. Hence, all the updates of field values must be performed through a method.

```
void move(int x, int y) {
  this@"A".move(x, y);
  this@"B".move(x, y);
}
```

If `move()` is called on a replication, then it is executed on all the hosts `A` and `B`. This declaration was not included in the aspect for `Square` shown above since the default behavior is implicitly implemented if any method declaration is not given in the aspect.

`this@` can be followed by a special host name `local`. `this@local` represents a reference to the callee replication. Note that a method invoker first calls a method on one of the replications, which usually exists on the local host and then the method call is propagated to other replications. `local` represents the host where there is the replication on which the invoker first called the method. Therefore,

```
int getX() {
  return this@local.getX();
}
```

implements that, if `getX()` is called, the method call is never propagated but the method is executed only on the local host. This customization improves execution performance if `getX()` causes no side-effects.

## 3.2 Tic Tac Toe with Jarcler

Now, we show an aspect for the `Square` object of the Tic-Tac-Toe example in Section 2.2. The aspect for `Square` written in Jarcler is as follows:

```
 1:replicate Square {
 2:  hosts "server","client1","client2";
 3:
 4:  char myMark;  // 'O', 'X', or ' '
 5:
 6:  Square(TicTacToe t, int p) on "client1" {
 7:    myMark = 'O'; this@local(null, p);
 8:  }
 9:
10:  Square(TicTacToe t, int p) on "client2" {
11:    myMark = 'X'; this@local(null, p);
12:  }
13:
14:  Square(TicTacToe t, int p) on "server" {
15:    myMark = ' '; this@local(t, p);
16:  }
17:
18:  char getMark() {
19:    return this@local.getMark();
20:  }
21:
22:  void clicked() {
23:    if (myMark == turn)
24:      this@"server".clicked();
25:  }
26:}
```

Line 2 specifies that replications of a `Square` object are allocated on three hosts, which are one game-server host named `server` and two GUI-client hosts named `client1` and client2. The rest of the lines alters the behavior of the constructor and the methods getMark() and clicked(). The behavior of the other methods nextTurn() and setMark() is the default one; method calls are automatically propagated to all the replications to maintain consistency among the replications.

Line 4 introduces a new field myMark into the class Square. A field declaration in an aspect is regarded as adding a new field to the class associated with the aspect. The value of this field represents the mark ('O' or 'X') of the player accessing the replication through a GUI object. The constructors in Line 6 to 16 assign a value to this field. If a replication of Square is created, the constructor defined in the aspect is called first. In an aspect, multiple constructors with the same signature can be declared for different hosts. The host name following on specifies which host the constructor is prepared for. The field myMark is set to 'O' if the host is client1, or 'X' if the host is client2.

Line 18 to 20 specify the behavior of getMark(). Since getMark() does not cause side-effects, the calls to getMark() are not propagated to the other replications. They are executed only on the local host.

Finally, Line 22 to 25 specify the behavior of clicked():

```
22:  void clicked() {
23:    if (myMark == turn)
24:      this@"server".clicked();
25:  }
```

If clicked() is called on one of the GUI-client host, it must be executed not on that host but on the game-server host. Furthermore, if this turn is not of the player who clicked on the screen, the method call to clicked() must be ignored. To implement this behavior, the values of the fields myMark and turn of the corresponding replication are compared in Line 23. Note that a method declaration in an aspect is treated as if it is in the corresponding class with respect to the name scope. If the two values are equal, clicked() is executed on the game-server host instead of the host where the callee replication exists.

## 3.3 Implementation

Some readers might think that the customization shown in this section could be described within the confines of the standard inheritance mechanism; they might not think an aspect of Jarcler is necessary. In fact, the description of an aspect looks similar to that of a subclass.

However, this observation is not true. From the implementation viewpoint, method declarations in the aspect for Square are used not for overriding methods declared in Square but for producing stub or wrapper code that implements a replicated Square object. Jarcler transforms the original definition of the Square class into the following definition[2]:

```
class Square {
     :
  void clicked() {
    if (myMark == turn) {
      // send a message to the game-server host.
      sendMessage("server", "clicked", "()V",
                  null);
    }
  }

  void sendMessage(String host, String method,
                   String signature,
                   Object[] args) { ... }

  void org_clicked() {
    // the method body of the original clicked().
  }
}
```

A replication on each host is an instance of this modified Square class. The original clicked() method is renamed to org_clicked() and a new version of clicked() is declared. The method body of the new clicked() is a copy of clicked() in the aspect although this@"server" is replaced with an expression implementing that semantics. The new clicked() sends a message to the game-server host so that the method org_clicked() is executed on that host.

A crosscutting concern that Jarcler deals with is the behavior of replicated object, which cuts across different classes of replicated objects. The default implementation of this crosscutting concern is automatically produced by Jarcler to be embedded in each class such as Square. However, if the user writes an aspect of Jarcler, the implementation is customized according to that aspect. Hence, if comparing Jarcler and AspectJ, an aspect in Jarcler corresponds to a sub-aspect in AspectJ, which extends another aspect, while the default implementation by Jarcler corresponds to that super-aspect. Since an aspect in Jarcler extends another aspect embedded in the Jarcler system, the standard inheritance mechanism of Java, which is for extending a class, cannot substitute for Jarcler.

# 4. RELATED WORK

## Reflection

A number of researchers have been proposed reflective middleware or languages for distributed computing. Their systems allow the users to customize the behavior of remote method calls [7, 6] or communication channels [2]. Although these systems provide different kinds of metaobjects, their design goal is to avoid repeatedly writing a slightly different code for every method to implement customized behavior. In other words, their focus is on enabling to implement customization, such as persistence and fault tolerance, with a single (or a small number of) generic code, which is short but reusable to cover a number of different methods. In the context of aspect-oriented programming, reflection can be regarded as a technique for making an aspect — a modular unit implementing a crosscutting concern — as reusable as possible.

However, in this paper, we have been discussing an aspect specialized for a particular class, which is not reusable in

[2]The definition shown here is a simplified version. The real one includes several support methods that are not shown here.

principle. To use for implementing these aspects, reflection is too powerful and it rather makes the programming more complicated. For example, the users cannot use standard Java syntax; the `invoke()` method must be called on a `Method` object for calling a method. This is inconvenient since the specialized aspect tends to access particular fields and methods. Reflection is necessary when implementing new behavior from scratch, but not when customizing existing behavior to fit a particular case.

We, therefore, exploited an AspectJ-like advice mechanism for Jarcler so that the users can easily describe aspects. Writing an aspect in Jarcler fits the user's intuition more than writing a reflective program; for example, the users of Jarcler can use regular Java syntax (except `this@`) in an aspect for calling a method and accessing a field and a parameter. While reflection is appropriate for writing general behavior such as the default one of Jarcler, our design of Jarcler is appropriate for writing behavior specialized for a particular class.

### AspectJ

Soares *et al* reported that they could use AspectJ for improving the modularity of their program written with the Java RMI framework [14]. Without AspectJ, the program must include the code following the programming convention required by the Java RMI. AspectJ allows to separate that code from the rest into a distribution aspect. A difference between their work and our work is that they did not use aspect-oriented programming for customizing the functionality provided by the middleware, that is, the Java RMI in their work.

An interesting question is whether we can use AspectJ instead of Jarcler's aspect language for customizing the behavior of a replicated object. Our answer is "No" because we need a language for specializing (or sub-aspecting) a sort of "aspect" providing stub or wrapper code for implementing a replicated object. This aspect performs relatively complex program generation, which cannot be described in AspectJ as long as AspectJ does not provide better capability of reflection or something equivalent. Since this aspect is hard-wired in the Jarcler System as the default implementation, we need a special aspect language, which provides special syntax support like `this@` so that a sub-aspect can access the functionality of that "super" aspect hard-wired in the system.

## 5. CONCLUSION

This paper presented our tool named Jarcler, which allows to use a replicated object in distributed software written in Java. A unique feature of this tool is that the behavior of the replicated object is customizable in an aspect-oriented way. Although reflection is a well-known technique for customizing middleware for distributed computing, we showed that aspect-oriented programming enables customization that fits the users' intuition better than reflection. Although reflection is suitable to describe behavior reusable for a number of classes, this paper discussed customized behavior only for a particular class. This paper claimed that an AspectJ-like advice mechanism is suitable for that customization.

We have finished the design of Jarcler and we are currently implementing Jarcler using Javassist [3], which is a bytecode translator based on reflection. Since the default behavior of replicated objects must be applicable to any class, implementation of that behavior needs reflection. However, this implementation using reflection is hidden from the users; Jarcler internally uses reflection but provides an aspect language for the users to customize the implementation.

## 6. REFERENCES


[1] Bal, H. E., J. G. Steiner, and A. S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *Computing Surveys*, vol. 21, no. 3, pp. 261–322, 1989.

[2] Cazzola, W., "mChaRM: Reflective Middleware with a Global View of Communications," *IEEE Distributed System On-Line*, vol. 3, February 2002.

[3] Chiba, S., "Load-time structural reflection in Java," in *ECOOP 2000*, LNCS 1850, pp. 313–336, Springer-Verlag, 2000.

[4] Chiba, S., G. Kiczales, and J. Lamping, "Avoiding Confusion in Metacircularity: The Meta-Helix," in *Proc. of the 2nd Int'l Symp. on Object Technologies for Advanced Software (ISOTAS)*, LNCS 1049, pp. 157–172, Springer, Mar. 1996.

[5] 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.

[6] Fabre, J. C. and T. Pérennou, "A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 78–95, 1998.

[7] Forman, I. and S. Danforth, *Putting Metaclasses to Work*. Addison-Wesley, 1998.

[8] Freeman, E., S. Hupfer, and K. Arnold, *JavaSpaces$^{TM}$ Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[9] Gelernter, D., "Generative Communication in Linda," *ACM Trans. Prog. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.

[10] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pp. 327–353, Springer, 2001.

[11] Li, K., "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Comp. Syst.*, vol. 7, no. 4, pp. 321–359, 1989.

[12] Maes, P., "Concepts and Experiments in Computational Reflection," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 147–155, 1987.

[13] Smith, B. C., "Reflection and Semantics in Lisp," in *Proc. of ACM Symp. on Principles of Programming Languages*, pp. 23–35, 1984.

[14] Soares, S., E. Laureano, and P. Borba, "Implementing Distribution and Persistence Aspects with AspectJ," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 2002.

[15] Tatsubori, M., T. Sasaki, S. Chiba, and K. Itano, "A Bytecode Translator for Distributed Execution of "Legacy" Java Software," in *ECOOP 2001*, LNCS 2072, pp. 236–255, Springer-Verlag, 2001.

[16] Tilevich, E. and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning," in *ECOOP 2002 — Object-Oriented Programming*, LNCS 2374, pp. 178–204, Springer, 2002.