Implementing Software Connectors through First-Class Methods

Cheoljoo Jeong and Sangduck Lee
Computer & Software Technology Laboratory
Electronics and Telecommunications Research Institute
Taejon, 305-350, Republic of Korea

Abstract

Recently, vast amounts of research efforts are made to derive a higher-level abstraction mechanism than the class abstraction in object-oriented languages. Notable examples include the research activities on software architectures and component-based software development. Of these, software architectures allow us to focus on the issue of software connectors, which are used to describe interactions between software components in the context of architectural styles. But, few contemporary programming languages support the explicit notion of connectors, which makes it difficult to practice software development based on software architectural concepts. In this paper, we present our experience in implementing software connectors through first-class methods in the Java programming language. We view software connectors as methods over methods and show that a suitable set of reflective facilities can be a nice device for implementing software connectors.

1 Introduction

Despite the strengths of object-oriented programming languages, a few shortcomings of those languages have also been discovered. For example, a problem with object-oriented programming is that it is hard to describe the collaboration between several classes [5, 10]. Given that a single collaboration defines a well-defined functionality, the fact that the implementation of this single functionality is dispersed through multiple classes makes it difficult to maintain the integrity of the behaviors of classes. When a need to change either the behavior of a collaboration or the behavior of any class involved in some collaborations, we should be careful not to violate the correctness of the system. As remedies to the shortcomings of object-oriented languages, a number of approaches have been proposed. aspect-oriented programming [8], subject-oriented programming [4], adaptive programming [10], software architectures [11], and so on. Given that there are two aspects, data and processes, in any computations, above approaches have one thing in common: they reflect the relatively recent realization that we have neglected to develop a high-level abstraction mechanisms for processes. Of those approaches, software architectural approach seems to be the most general in the sense that the key concept of software connectors in software architectures represents all kinds of interactions between software components.

Though the software architectural concepts [12] provide a sound background for a higher-level abstraction mechanism for describing component interactions, it’s hard to put those concepts into practice since there are few programming languages supporting them. Architectural description languages such as UniCon are not enough since they are declarative and not executable. In this paper, we present our experience in implementing software connectors through first-class methods.

Most popular object-oriented programming attaches every method of a program explicitly to a specific class. The methods are not first-class values since we have no way to manipulate them but to invoke them. When we expose the methods to be first-class values through a metaobject protocol, we are endowed with the power to:

• designate a set of methods which would be used in connecting software components and

• to build a procedural abstraction over the designated set of methods, i.e. methods over methods.

Our first-class treatment of methods can be a first step toward implementing first-class connectors as advocated in:

We argue that the abstractions embodied in the connecting lines of architectural diagrams require first-class treatment and indicate what form this might take in a language.
The Java programming language provides a metaobject protocol which allows us to manipulate methods. Since objects are first-class values in Java, the existence of ‘method’ metaobject provides us a natural way to deal with methods as first-class values. We present how we can implement software connectors using Java reflective facilities. Throughout the presentation, we show that ‘methods as first-class values’ can help us implement software connectors with as little support from programming languages as possible. Our idea can be used only for implementing the concepts of software connectors but also the concepts of collaborations and aspect-oriented programming.

The organization of this paper is as follows. In section 2, we briefly review the reflective facilities provided by the Java Development Kit 1.x. In section 3, we present how we can implement software connectors in an experimental programming language, SchemeJ, which supports the notion of first-class methods and, in section 4, we show a simple software connector example written in SchemeJ. In section 5, concluding remarks and further research directions are given.

2 Reflective facilities in Java

In this section we briefly review the reflective facilities provided by the Java programming language. Java supports some reflective facilities through java.lang and java.lang.reflect packages [3].

2.1 Reflective facilities in Java

The Java programming language defines a metaobject protocol which allows us to access the internals of the Java virtual machine. Of several meta-level classes, we are particularly interested in the java.lang.reflect.Method class. A Method object can be used to invoke the underlying method through the following method.

Object invoke(Object obj, Object[] args);

Since objects are first-class values in object-oriented programming languages, Method metaobject provides us a natural way to make methods first-class citizens in object-oriented languages.

2.2 Methods as first-class citizens

One important consequence resulting from the first-class treatment of methods is that we can select a set of methods from a set of classes and group them and treat them as a single unit. This ability enables us to abstract collaborations between software components into a software connector by defining a method over the methods of the components. This method accepts the participating methods as its arguments and invokes them in a predefined rule, e.g. event-based, message-passing-based, and so on.

3 Software connectors through first-class methods

In this section we present how we can implement software connectors using the Method class. For an experimental purpose, we have designed and implemented a programming language, SchemeJ [6], which allows composition of software components. A process of composition is essentially a process of defining a software connector. SchemeJ is an R5RS-compliant [7] extension of Scheme equipped with operators for manipulating Java bytecodes.

3.1 The SchemeJ programming language

SchemeJ is Java-bytecode-aware version of Scheme. One salient feature that distinguishes SchemeJ from other similar languages like Kawa [2] lies in its ability to deal with methods as first-class values not as subordinates to classes. In this paper, for simplicity, we define components just as objects of any Java class. We are in the process of defining a Java interface, say JComponentI, so that components would be defined to be objects of any Java class that implements the JComponentI interface (for the status of this research effort, see [6]).

While components are Java classes, connectors are any SchemeJ procedures that are defined using components. SchemeJ provides standard procedures for handling Java classes and objects. Of these, we present a subset of them which has the direct relevance to our presentation.

- (load-class string): loads the class file represented by string and returns the corresponding class.
4 Implementing software connectors in SchemeJ

In this section we present a simple example showing how we can implement architectural styles. For this we implement the interaction style of implicit invocation [12].

4.1 Implicit invocation style of interaction

In the implicit invocation style of component interaction, computation is invoked by the occurrence of an event and there is no explicit interactions among processes. To implement this style, we need to implement a event-dispatch system which invokes appropriate methods subscribing to according to the generated events.

To implement this style we need to design a event-subscription mechanism and event-dispatch mechanism. In Scheme, we can use the data-directed programming technique to implement the event-subscription mechanism [1]. We also add hooking mechanism so that each method not only listens to events but also generates events. For these services, we need to implement the following procedures.

- (create-object c symbol list): creates an object, symbol of class c with list of constructor arguments
- (send o m list): invokes the method m of object o with the list of arguments
- (invoke m list): invokes the method m with the list of arguments; note that this procedure does not need the name of the object to which the message is send
- (get-methods obj): returns the set of method objects of the corresponding class where obj is either a Java class or a Java object; note that a method is an ‘object’

Given the above procedures, we can select the methods from a set of classes through the get-methods procedure. And group methods using the Scheme list manipulation procedures such as append. The standard procedure invoke is important in the sense that we can treat methods independent of the objects or classes of which they are attributes.

Note that the invoke-with-hooks procedure now replaces the role of invoke procedure which was used execute the methods. In implicit invocation style of interaction, generated events determine which methods are to be invoked and the order of invocations.

4.2 Implementation

Since implementing publish-subscribe mechanisms can be easily implemented using data-directed programming technique, we just present definitions of procedures invoke-with-hooks, and event-dispatch. The procedures are defined in Figure 1. The event-dispatch procedure was simplified to handle a single event. It can be modified to a full-fledged event loop mechanism.

Now, we show how we can use the above procedures to describe a implicit invocation style of interaction. Suppose we have three methods, m1, m2, and m3. And there are two types of events, e1 and e2. Given that m1 subscribe to e1, we can describe this situation as follows.

(add-listener 'e1 m1)
(event-dispatch 'e1)

Evaluation of above expressions will result in the invocation of m1.
(define invoke-with-hooks
  (lambda (method args)
    (begin
      (get-pre-hooks method)
      (invoke method args)
      (get-post-hooks method))))

(define fire
  (lambda (method-list)
    (if (null? method-list)
      (void)
      (begin
        (invoke-with-hooks
          (car method-list))
        (fire (cdr method-list))))))

(define event-dispatch
  (lambda (event)
    (let (method-list (get-listener event))
      (fire method-list))))

Figure 1: Procedures for implicit invocation style of interaction

If want to add an ordering between methods invocations we can use the hooking mechanism. Suppose that we want that methods m2 and m3 are invoked after every invocation of the method m1. Then we can add a post hook to the methods m1 and let m2 and m3 subscribe to the event e2.

(add-listener 'e2 m2)
(add-listener 'e2 m3)
(add-post-hook m1
  (lambda () (event-dispatch 'e2)))
(event-dispatch 'e1)

5 Conclusion

In this paper we presented a method for implementing software connectors through first-class methods. A first-class method is one which can be assigned to variables, passed as arguments to another method, and used as a return value of some method. We have shown that, of various meta-level concepts, the exposure of the notion of methods allows us to implement software architectural concepts in a straightforward way.

We have used a reflective technique to achieve our goal and reflection certainly seems to be a sufficient mechanism for ‘simulating’ various approaches including software architectures, aspect-oriented programming, and so on. But reflective techniques are often argued as so powerful that they can be dangerous to average programmers [9]. So we need to search for a way to represent higher-level abstraction mechanisms which is both safe and powerful.

References


