

Scenario-Based Software Architecture for Designing Connectors Framework in Distributed System

Hamid Mcheick[1], Yan Qi[2] and Hafedh Mili[3]

[1] Computer Science Department, University Of Quebec At Chicoutimi, Chicoutimi, Quebec G7H2B1, Canada

[2] Computer Science Department, University Of Quebec At Chicoutimi, Chicoutimi, Quebec G7H2B1, Canada

[3] Computer Science Department, University Of Quebec At Montreal, Montreal, Quebec H3C3P8, Canada

Abstract
Software connectors is one of key word in enterprise information system. In recent years, software developers have facing more challenges of connectors which are used to connect distributed components. Design of connectors in an existing system encounters many issues such as choosing the connectors based on scenario quality, matching these connectors with design pattern, and implementing them. Especially, we concentrate on identifying the attributes that interest an observer, identifying the functions where these connectors could be applied, and keeping all applications clean after adding new connectors. Each problem is described by a scenario to design architecture, especially to design a connector based on architecture attributes. In this paper, we develop a software framework to design connectors between components and solution of these issues. A case study is done to maintain high level of independency between components and to illustrate this independency. This case study uses Aspect-Oriented Programming (AOP) and AspectJ, Design Pattern to and Program Slicing to solve main problems of design of connectors. A conclusion is given at the end of this paper.
Keywords: *Design Connector, Attribute Driven Design, Software architecture, Scenario based system.*

1. Introduction

1.1 General

In enterprise information system, each connector has a protocol specification that defines its properties [1]. Connectors can be also understood as some software elements which provide a conduit from one or many components to one or many components. The connector may also adapt the protocol and format of the message from one component to another [2].
Our design connectors process could be described as follows:

- choosing connectors based on scenario quality and stimulus described in software architecture [16],
- matching these connectors with design pattern and finally,
- implementing them using a programming language using aspect-oriented programming.

In software developing history, there are some classic connectors which are often used in desktop software system. These connectors are function call (method call), association class, class inheritance and share memory etc. These basic connectors work very well among components which are situated in a same application. Software developers always neglect them, because they do not need more code to implement them (even no code).
However, in the recent years distributed system is a growing trend. The distributed programs are making the software architecture and their connectors become more and more complicated. A distributed system is a system composed of several computers which communicate through network, hosting processes that use a common set of distributed protocols to assist the coherent execution of distributed activities [4]. The connectors situated in distributed system play a significant role in whole software architecture. Software engineers must face more challenges of connectors which are used to connect distributed components.
In this paper we mainly concentrate on the design of connectors in distributed system. Especially, we concern ourselves about the software maintenance in which new connectors are designed and added among existing components. Indeed, the idea is choosing the connectors, identifying the attributes that interest the observers, identifying the functions where these connectors could be

applied, and keeping all applications clean after adding new connectors.
We propose a set of methods to design and add the connectors. Among the methods, Aspect-Oriented Programming (AOP), Design Pattern and Program Slicing are three important technologies and they are described in detail.

1.2 Statement of the Problems

Software engineers must face more and more challenges of connectors which are used to connect distributed components, when they analyze, design, implement and maintain connectors. Naturally, this general problem raises many question: What is the reason to make connector design, implementation and maintenance become more difficult? How can we make this connector easy to reuse, replace and maintain? The answer described below to these questions leads to framing the problem area for our research:
The area of connector design and maintenance is seldom researched, especially, the maintenance of connector. Connector maintenance focuses on how to update or add connectors in an existing software system and how to keep the existing components clean instead of messing it after bringing new connector out. And this problem is quite different to design a connector for a new system.

1.3 Research Questions

The problems of our research can introduce the specific questions of research. In the following sections, we focus on two questions which software developers often meet.

- How to add the code related to new connector without modifying existing source code very much as far as possible? In other words, we should try our best to keep the existing components clean instead of messing it after bringing new connector out. And the component messed will become incompatible with the rest of your system [3].
- Where automatically to find the interesting points (or statements), which the other components want to know through connectors, in the existing source code? That means that we should determine the place of status or values in legacy component which should be provided to a new component. It is often difficult for the developer to grasp the basic architecture and relationships between code units, and this is made more difficult by the fact that the code may be poorly documented and poorly written [11].

In other words, the connectors should be added in distributed system without modifying more existing source code. And the point of source code which will be inserted a connector should be accurately found. For example, to maintain some large source code, if the software developers want to add a new connector between components, they must carefully analyze those codes and avoid messing the existing design and implementation. To implement these issues, we choose Aspect-Oriented Programming and Slicing tool techniques. A case study is done to proof the concept of maintaining the independency of components and then reduce the cost of the maintenance

1.4 Structure of Papers

The rest of the paper is organized as follows. We firstly present the overview of AOP, design pattern and programming slicing technique (section 2). Secondly, a mode for design of connectors is proposed in distributed system (section 3). Thirdly, an implementation based on AOP, design pattern and programming slicing are shown (section 4). Finally, a case study is given to illustrate that mode and those technologies (section 5) and a conclusion is given in section 6.

2. Background

Software maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes [5]. Designing a new connector for an existing distributed system belongs to software maintenance. And it will be a big job. Particularly, when the developers are not familiar with the system and they don't have enough documents about the project, they must spend much time analyzing and handling the existing source code. Software developers sometimes may put themselves in bad situation: they cannot find the right point (or statement) in large and complex source code files to provide the information for other components through connector. Or after adding a new connector, the source code becomes more and more difficult to read and understand. In this section we describe three important technologies: Aspect-Oriented Programming, Design Pattern and Program Slicing. They can be used to avoid the problems mentioned above

2.1 The classification of Connectors

In this section, we discuss the connector type, the interaction service between components and the usage of the existing connector.
There are eight types of software connectors [6]:

1. Procedure Call,
2. Event,

3. Data Access,
4. Linkage,
5. Stream,
6. Arbitrator,
7. Adaptor, and
8. Distributor.

The service categories provided by connector are described as below [6]:

- *Communication*: Communication connectors support transmission of data among components.
- *Coordination*: Coordination connectors support transfer of control among components.
- *Conversion*: These connectors convert the interaction required by one component to that provided by another.
- *Facilitation*: Facilitation connectors mediate and streamline component interaction.

Those four services provided by connector are requirements of component interactions. Different connector types can satisfy the requirements of the interactions between components. From Mehta et al.'s research [20], we can find that different connector types can and only can support specific service. In other words, one kind of type can only be used in some particular circumstances. For example, one connector which belongs to type Procedure Call is suitable to use for a situation in which two components need communication service. On the contrary, using a Procedure Call connector to provide a conversion service must be a bad idea.

However in Balek et al.' research work [20], Balek et al., discuss that Mehta et al.'s taxonomy does not talk about the how to design connector types, and that those connector types are at different software levels. For example procedure calls are the assembly language of software interconnection [1]. It is not suitable to be put it together with event or data access.

2.2 Aspect-Oriented Programming (AOP)

Aspect-oriented programming is a paradigm that supports two fundamental goals: [7]

- Allow for the separation of concerns as appropriate for a host language.
- Provide a mechanism for the description of concerns that crosscut other components.

AOP isn't meant to replace OOP or other object-based methodologies. Instead, it supports the separation of components, typically using classes, and provides a way to separate aspects from the components [7].

We can use AOP to crosscut classes (components) to setup relationships between them without modifying functions in the code of original components. In other words, the code can be kept clean and independent by using Aspect-Oriented Programming.

2.3 Design Patterns

Design patterns are always used to describe relationships and interactions between components (classes or objects). The design patterns in Gamma et al.'s book [10] are descriptions of communication of objects and classes that are customized to deal with general design problems. And the communication of object and class should be loose coupling without becoming entangled in each other's data models and methods [8]. Gamma et al.'s design patterns particularly deal with problems at the level of software design, especially object-oriented software design. They can be classified by criterion scope which is used to specify whether the pattern is mainly used to classes or objects. So the design patterns have two kinds: class design pattern and object design pattern [10]. Object patterns are applied to object relationships, which can be modified at run-time and are more dynamic, such as Proxy Pattern, Observer Pattern etc.

In our research we primarily focus on the object design patterns and extend the area of the design patterns which are applied to distributed architecture (especially the messaging-oriented system). For example, the Observer pattern can be reinterpreted and redesigned in distributed architecture, which is sometimes called publish-subscribe style. In order to describe the relationship of components, we use Aspect-oriented programming (AOP).

2.4 Spring Framework

The Spring framework is a wide-ranging framework to develop enterprise Java applications. It provides a lightweight solution and a potential one-stop-shop for building enterprise-ready applications [18]. The Spring Framework is organized as modular. These modules are grouped into 6 types [18]: i) Core Container, ii) Data Access/Integration, iii) Web, iv) AOP (Aspect Oriented Programming) Instrumentation, and v) Test.

Core container is one of most important modules. Especially, in core container, Inversion of Control (IoC, it is also known as dependency injection) is key feature of Spring framework. Dependency Injection is based on Java language constructs, rather than the use of framework-specific interfaces. Instead of application code using framework APIs to resolve dependencies such as configuration parameters and collaborating objects, application classes expose their dependencies through methods or constructors that the framework can call with the appropriate values at runtime, based on configuration [19].

A connector situated in a distributed system must be based on certain transport mechanism (such as TCP/UDP socket, messaging system…) to connect components in network. According to some situation, the distributed connector needs to change the transport mechanism during the

runtime. For example, when the quality of network becomes more stable, the system has the intention of dynamically changing the TCP socket to UDP socket. For reuse of design (and code) and flexible loading mode of java class (or bean), we apply IoC of Spring framework to achieving this objective.

2.5 Program Slicing

Program slicing is a well-known program analysis and transformation technique that uses program statement dependence information to identify parts of a program that influence or are influenced by an initial set of program points of interest (called the slice criteria) [11]. A program slice constructed by identifying program points that affect a given program point is called a backward slice. A program slice composed of program points affected by a program point is called a forward slice [13].
Engineers apply program slicing technique in lots of programming area:

- Program Debugging;
- Program Comprehension;
- Program Testing.

When starting to design a connector, we firstly want to find a point (or statement) in one source code of component. The information included in that point will be notified to another component through connector. Fortunately, the finding process can be automatically done by using backwards program slicing starting with a variable as the slicing criteria.

2.6 Attribute Driven Design

The Attribute-Driven Design (ADD) method is an approach to defining a software architecture in which the design process is based on the quality attribute requirements the software must fulfill [16]. ADD is situated after requirements analysis and before high level design. So the input to ADD is a set of requirements which are regarded as a set of quality attribute scenarios; and the output is the input of high level design. There are 8 steps performed when designing an architecture using the ADD method [21]. These steps are described in Figure1.
The quality attribute scenarios mainly include availability, modifiability, performance, security, testability and usability. Among the steps of ADD, analyzing those quality attribute scenarios is the most important step of the design method.

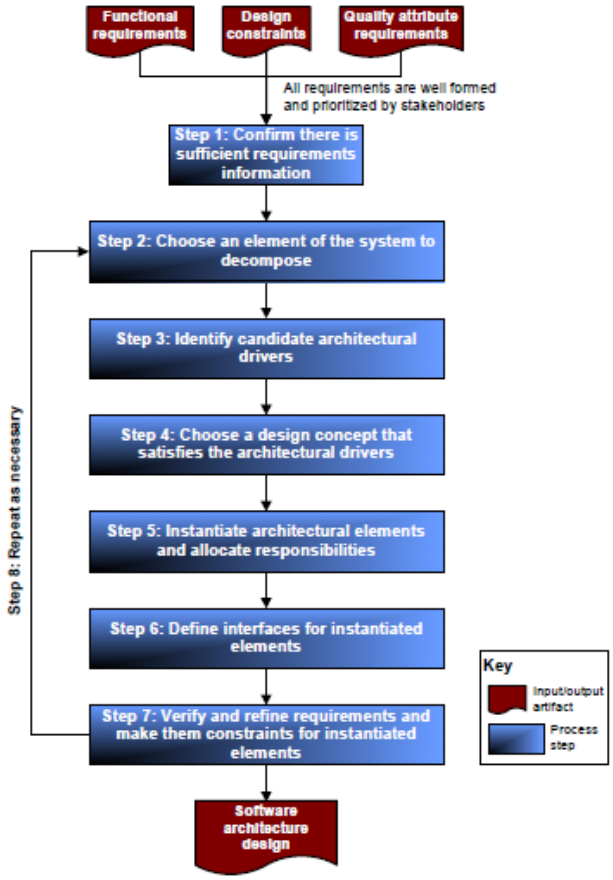

Fig. 1 Software architecture design process.

3. Model for Design of Connectors in Distributed System

In this section, we present a mode for design of connectors in distributed system. In distributed system, components are situated in separate node, which may be computer, PDA or server etc. Figure 1 shows distributed components (A, B, C and D) which are situated in different node (1, 2 and 3) in network.
In Figure 1, the lines are actually software connectors which are used for communication among components. In other words, connectors can be applied to describing the relationship between components (or distributed components).

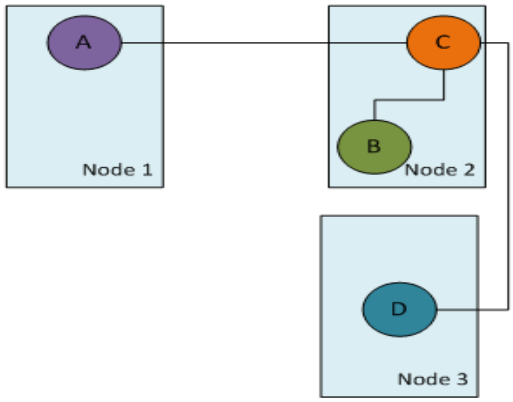

Fig. 2 Distributed components in different nodes.

We introduce a new model for design connectors based on the distributed architecture (Figure 2). According to some design pattern, component D wants to get some value or status of component A. So we must add a new connector to describe the relationship between components A and D.

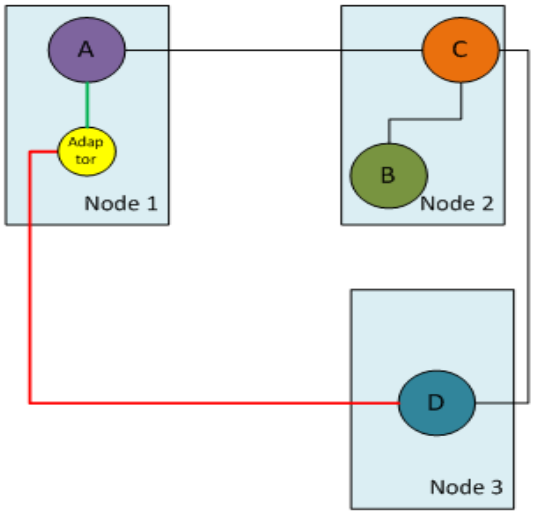

Fig. 3 New added connector between components.

In the figure 3, a new connector is described by two lines (a green and a red one) between component A and D. In other words, the new connector consists of two sub-connectors. The following diagram (figure 4) gives a description of the new connector. We add a new component which runs in Node 1 as an Adaptor. Aspect-oriented programming technique is applied to the sub-connector between component A and Adaptor according to design patterns. We can get the benefit from the AOP technique to deal with communication between A and Adaptor without any modification.

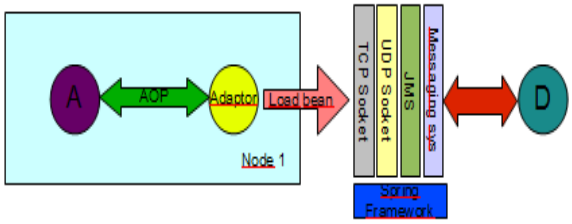

Fig. 4 The new connector.

The only thing we need to do about component A is to analyze the source code. That's mean we should find the interesting statements which are used to crosscut the component. Programming slicing technology is applied to analyze the source code. As for this kind of slicing, we can use backward slice to get data dependence among the source code. Another sub-connector between Adaptor and component D is usually implemented by using common network protocol or a messaging system, such as TCP (/UDP)/IP, JMS, etc. In order to support the switch dynamically from one transport mechanism to another, we use Spring framework to configure these java beans at run time.

At last we draw a conclusion about the mode – how to design a message-based connector in distributed system.

i) Analyze the relationship between two components;
ii) Choose a design pattern to describe the relationship;
iii) Slice source code of component to find the place of the statement which another component is interested in;
iv) Crosscut the component by using AOP;
v) Add an adaptor to connect two different protocols;
vi) Configure the different transport bean by using Spring.

This model can be not only applied to distributed system, but desktop application as well.

4. Implementation of Connector Model

The implementation of design of connector consists mainly of developing tools used by the mode. Details of implementation are given in case study (section 5.2). Note that the appendix shows the implementation of the protocol (connector) between two components.

4.1 AspectJ and Gang-of-Four Design Patterns

AspectJ is an implementation of AOP for the Java language built as an extension to the language. A compiler and a set of JAR files take common Java code and AspectJ

aspects and compile them into standard Java byte-code, which can be executed on any Java-compliant machine [7]. The Gang-of-Four (GoF) design patterns [10] offer flexible solutions to common software development problems. Each pattern is comprised of a number of parts, including purpose/intent, applicability, solution structure, and sample implementations [12]. GOF provides 23 well-known design patterns. Software developers can use one of them or combine some of them to describe the relationship between components. Design of connectors depends on it. Some results show that using AspectJ improves the implementation of many GoF patterns [12].

4.2 Indus Java Program Slicer and Kaveri

Indus Java program slicer is a part project of Indus [14]. The Indus slicer is the first and only publicly available Java slicing framework, and can handle almost all features of Java [11].
Kaveri is an eclipse plug-in front-end for the Indus Java slicer. It utilizes the Indus program slicer to calculate slices of Java programs and then displays the results visually in the editor. Kaveri is an effective tool for simplifying program understanding, program analysis, program debugging and testing [15] [17].
Using Kaveri plug-in can automatically slices the Java source code and provides the data and control dependence. According to the results of data and control dependency, developers can easily find the crosscutting points for design connectors.

5. Case Study

In this section, we present one case study to design connectors in distributed system. Firstly we do a design process by using ADD approach; then the actual design and implementation of the case are discussed by using the model described above and related tools.

5.1 ADD Design Process

In this section we adopt Attribute-Driven Design (ADD) approach to define a distributed software architecture in which the design process of the new connector is based on the quality attribute requirements the software must fulfill. The approach can follow a recursive process that decomposes a system or system elements by applying architectural tactics and patterns that satisfy its driving quality attribute requirements.

In the case study, we only focus on an important system quality attribute: modifiability scenarios. By following the ADD method, we discuss the steps of design in more detail.

Step1, choose the module to decompose. The existing system which runs on desktop is targeted, since it is the system's primary element.
Step2, choose the architectural drivers. As the input of ADD, one quality scenarios are chosen as the quality requirement: How to add the code related to new connector without modifying existing source code very much as far as possible? In other words, we should try our best to keep the existing components clean instead of messing it after bringing new connector out. And the component messed will become incompatible with the rest of your system. We present the possible value for each portion of modifiability scenario.

Table 1: Quality Attribute Scenario

Modifiability Quality Attribute Scenario	
Portion of Scenario	**Possible Value**
Source	Developer and system administrator.
Stimulus	Hope to add a connector to setup the relationship between the two existing modules.
Artifact	Software structure and system environment.
Environment	Design time
Response	Makes modification without messing the existing source code
Response Measure	How much source code is modified

Step 3, choose an architectural pattern. We choose Publish-Subscribe architectural pattern.

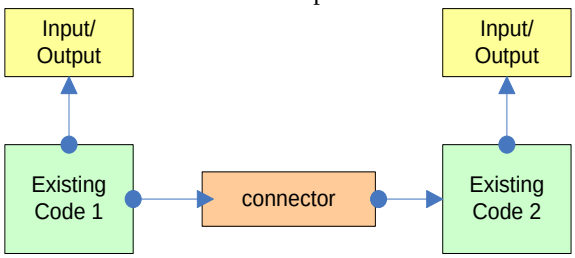

Fig. 5 Connector between components.

Step 4, instantiate modules and allocate functionality with module decomposition view.

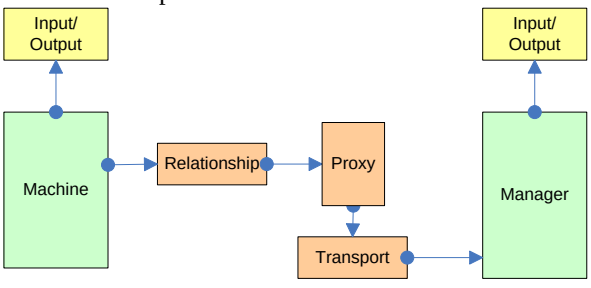

Fig. 6 Proxy between components.

Step 5, define interfaces of the child modules.

- Module Relationship provides the service to setup Observer design pattern;
- Module Proxy is the agent of Machine and it can produce the information;
- Module Transport carries the information produced by Proxy.
- Module Manager receives and consumes the information.

Step 6, Verify quality scenarios as constraint for the child modules. The modifiability quality attribute scenario can be satisfied by the multiple child modules. The Relationship module is designed to setup the interaction without modifying module Machine; Transport module is responsible for linking Manager and Proxy together.

5.2 Actual Design and Implementation

In this sub-section, following the design result of ADD, we present an example to design connectors by using the model and developing tools described above in order to reduce the cost of maintenance to compose software based on legacy components. In the example, we use Java and AspectJ as main developing language; use Observer design pattern [12] and UDP/IP network protocol to describe the relationship of components to implement a new connector; use Kaveri in Eclipse to slice a source code to get the "pointcut" for connector. Some key parts source code of the existing system is described as below:

Class AppSystem

```
public class AppSystem {
    public static void main(String[] args) {
        Machine OneMachine = new Machine();
        byte i = 0;
        double speed = 0.0;
        OneMachine.method4();
        OneMachine.methodF();
        OneMachine.method2();
        OneMachine.methodG();
        speed = OneMachine.getSpeed();

    }
}
```

In the simple application, there are two classes: Machine and AppSystem. Now we decide to upgrade simple application to a distributed system based on some requirements. Because another component named "Manager" which runs in another computer in network wants to be notified when the speed of the Machine has been changed. Apparently, a new connector should be added in the system.

After analyzing the relationship between Machine and Manager, we decide to choose Observer design pattern to describe the relationship. In order to use Observer design pattern and AspectJ, we should firstly find the point around which the speed is changed. The point is also called pointcut in AOP.

Class Machine:

```
public class Machine {
    private double speed = 0.0;
    private double temperature = 0.0;
    public double getSpeed(){
        double ret = 0;
        ret = speed;
        return ret;
    }
        … …
    public void methodF(){
        method1();
        methodH();
        method2();
        method1();
    }
    public void methodG(){
        method2();
        methodH();
        method2();
    }
    public void methodH(){
        speed = speed + 1;
    }
}
```

We assume that the code of Machine is very large and complex. So if we manually scan the whole source code, we may be get the wrong statement or miss some points. So we do a program slicing for the system by using Kaveri. We pick statement "speed = OneMachine.getSpeed();" as a criteria. And I choose "value of the expression". Then I start the process by using backward program slicing. The slice result is described as below:

Class Machine:

```
public class Machine {

    private double speed = 0.0;
    private double temperature = 0.0;
    public double getSpeed(){
        double ret = 0;
        ret = speed;
        return ret;
    }
    public void methodF(){
```

```
            method1();
            methodH();
            method2();
            method1();
      }
      public void methodG(){
            method2();
            methodH();
            method2();
      }
      public void methodH(){
            speed = speed + 1;
      }
}
```

Class AppSystem

```
public class AppSystem {
      public static void main(String[]
args) {
            Machine OneMachine = new
Machine();
            byte i = 0;
            double speed = 0.0;
            OneMachine.method4();
            OneMachine.methodF();
            OneMachine.method2();
            OneMachine.methodG();
            speed                    =
OneMachine.getSpeed();
      }
}
```

At last, we add a new class Adaptor in that simple application. The Adaptor works as a role of Observer and relays information about speed to component Manager. The Machine works as a role of Subject. So a new connector between Machine and Manager is designed and added in that distributed system. After adding all code about new connector, the existing component Machine is not changed. It is kept very clean in this situation. We just added some new files to the system which can be easily removed or changed without affecting the logic of original components.

6. Conclusion

This paper proposes a model and discusses the design of connectors in distributed system. At first, we present two problems when adding a new connector: 1) How to keep all source code clean after adding new connectors? and 2) Where automatically to find the place of status or values in legacy component which should be provided to a new component through connector? Then a connector model is proposed to deal with the problems. In this model, AOP, design pattern and programming slicing are combined to resolve the problems of design of connector. Although, a case study is done to show the independency between components. It illustrates how to use the combination of these technologies to support high level of independency. For example, the example shows that it is easy to find a point in a complex source code and that the original code is not almost changed after being added a new connector.

In our future work, we are planning to analyze all kinds of connectors according to different design patterns. And we will provide different and generalized solutions of design connectors based on different results of analysis.

Appendix

This Appendix shows the implementation of the protocol (connector) between two components given in section 5. This protocol is implemented in two artifacts: ObserverProtocol and ObserverProtocolImpl.

```
package connector;

/*
 * This file implements the Observer
Protocol.
*/
import java.util.WeakHashMap;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;

public abstract aspect ObserverProtocol
{
      /*
To set which class can be subject. */
      protected interface Subject { }

      /*To set which class can be
observers. */
      protected interface Observer { }

      /* Stores the mapping between
Subjects
      * and Observers. For each
subject, a
      * LinkedList is of its observers
is stored.*/
      private                WeakHashMap
perSubjectObservers;

      /*
```

```
      * Returns a Collection of the
observers of
      * a particular subject.
      * param s: the subject for which
to return the observers
      *   return   a:   Collection   of
observers of subject
      */
      protected                    List
getObservers(Subject s)
      {
       if (perSubjectObservers == null)
        {
            perSubjectObservers  =  new
WeakHashMap();
        }
        List        observers        =
(List)perSubjectObservers.get(s);

        if ( observers == null )
        {
            observers=new LinkedList();
            perSubjectObservers.put(s,
observers);
         }
         return observers;
      }

      /*
      * Adds an observer to a subject.
      * param s: the particular subject
to attach a new observer
      * param o: the new observer to
attach
      */
      public  void  addObserver(Subject
s, Observer o)
      {getObservers(s).add(o);}
      /*
      *  Removes  an  observer  from  a
subject list.
      * param s: the particular subject
      * param o: the observer to remove
      */
      public                        void
removeObserver(Subject s, Observer o)
      { getObservers(s).remove(o);}
      /*
      * The join points after which to
do the update.
      */
      protected    abstract    pointcut
subjectChange(Subject s);

      /*
      * Call updateObserver after a
change of interest to
      * update each observer.
      * param s: the subject on which
the change occured
      */
      after(Subject                 s):
subjectChange(s)
      {
         Iterator        iter        =
getObservers(s).iterator();
         while ( iter.hasNext() )
         {
            updateObserver(s,
((Observer)iter.next()));
         }
      }

      /*
      *Defines how each Observer is to
be updated
      *  when  a  change  to  a  Subject
occurs.
      * param s: the subject on which a
change of interest occured
      *  param  o:  the  observer  to  be
notifed of the change
      */
      protected      abstract      void
updateObserver(Subject s, Observer o);
}
```

```
package connector;

import main.*;

public   aspect   ObserverProtocolImpl
extends ObserverProtocol
{
      /*
      *  Declare  the  Subjects  and  the
Observers
      */

      declare     parents:     Machine
implements Subject;

      declare     parents:     Adaptor
implements Observer;

      /*
      *  Set  the  pointcut.  Advise  the
method methofH().
      */
```

```
      protected                  pointcut
subjectChange(Subject subject):
            call(void
Machine.methodH()) && target(subject);

      /*
      *  Updating  the  observer,  when
monitoring the change of the subject.
      */
      protected                      void
updateObserver(Subject          subject,
Observer observer)
      {
        double speed = 0.0;
        Machine         machine         =
(Machine)subject;
        Adaptor         adaptor         =
(Adaptor)observer;
        speed = machine.getSpeed();
        adaptor.upDate( speed );
      }
}
```

Acknowledgments

This work was sponsored by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and by the University of Quebec at Chicoutimi (Canada).

References

[1] Shown and Garlan, Software Architecture, Prentice-Hall 1996.
[2] James McGovern, Scott W. Ambler, A practical guide to enterprise architecture, 2004.
[3] Jeffrey Voas, Reliable Software Technologies, Maintaining Component-Based Systems, "IEEE Computer Society Press", USA, 1998.
[4] Paulo Veríssimo, Luís Rodrigues, Distributed systems for system architects, Kluwer Academic Publishers, USA, 2001.
[5] ISO/IEC 14764: Software Engineering — Software Life Cycle Processes — Maintenance, 2006.
[6] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. "Towards a taxonomy of software connectors". In Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), pages 178-187, Limerick, Ireland, June 4-11, 2000.
[7] Joseph D. Gradecki and Nicholas Lesiecki, Mastering AspectJ: Aspect-Oriented Programming in Java, Wiley, 2003.
[8] James W. Cooper, Java design patterns: a tutorial, Addison-Wesley Professional, 2000.
[9] Jan Hannemann, "Design Pattern Implementations using Aspect-Oriented Programming", available at http://hannemann.pbworks.com/Design+Patterns, 2008.
[10] Erich Gamma, Ralph Johnson, John Vlissides, Richard Helm. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[11] Venkatesh Prasad Ranganath, John Hatcliff, Slicing Concurrent Java Programs using Indus and Kaveri, International "Journal on Software Tools for Technology" Transfer (STTT), Vol.9, Numbers 5-6, pp. 489-504, 2006.
[12] Jan Hannemann and Gregor Kiczales, Design Pattern Implementation in Java and AspectJ, "ACM", USA, 2002.
[13] Venkatesh Prasad Ranganath, Indus - Java Program Slicer, 2008.
[14] Indus project, Available at http://indus.projects.cis.ksu.edu/
[15] Ganeshan Jayaraman, Kansas State University, Indus–Kaveri, 2008.
[16] Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice, Addison-Wesley, 2003.
[17] Venkatesh Prasad Ranganath , John Hatcliff, "Slicing concurrent Java programs using Indus and Kaveri", International Journal on Software Tools for Technology Transfer (STTT), v.9 n.5, p.489-504, October 2007
[18] Reference Documentation of Spring Framework, Available at http://www.springsource.org/
[19] Rod Johnsonet al., Professional Java development with the Spring Framework, John Wiley & Sons, 2005
[20] D. Balek and F. Plasil (2000). "Software connectors: A hierarchical model". Technical Report 2000/2, Charles University.
[21] William G. Wood (2007). "A Practical Example of Applying Attribute-Driven Design (ADD)",Version 2.0. TECHNICAL REPORT CMU/SEI-2007-TR-005 ESC-TR-2007-005.

Hamid Mcheick professor of computer science at the University of Quebec in Chicoutimi, Chicoutimi, Canada. He has PhD in computer Science from Montreal University, Msc in Computer science from University of Quebec at Montreal, Canada, and Bachelor in Computer Science-Mathematic from Lebanese University. Professor Mcheick is a member of IEEE, IEEE Society and ACM. He is interested in software architecture, evolution and distributed object and service oriented computing.

Yan Qi Student Master degree at the University of Quebec at Chicoutimi. He has Bachelor in Computer science from China. He is interested in Software architecture area.

Hafedh Mili professor of computer science at the University of Quebec in Montreal, Montreal, Canada. Throughout his academic career, he worked on a variety of subjects, starting with knowledge representation, object-oriented software engineering, aspect-oriented development, service-oriented computing, and business process engineering.