Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 8 (18): 3139-3147, 2008
ISSN 1812-5654

Mediator Connector for Composition of Loosely Coupled Software Components

H. Sanatnama, A.A.A. Ghani, N.K. Yap and M.H. Selamat
Faculty of Computer Science and Information Technology,
University Putra Malaysia, 43400 UPM, Serdang, Malaysia

Abstract: Component-based software development is an approach that has many benefits, such as improving application developer productivity, reducing costs and complexity by reusing of existing code. Programming in this approach is like assembling (i.e., composing software out of prefabricated components) rather than development, which reduces skill requirements and lets expertise focuses on domain problems. Component model is the cornerstone of any CBSD methodology, which defines what components are, how they can be constructed and specifies the standards and conventions that are needed to enable composition of independently developed components. The current component models focus on the specification and packaging of components but provide almost no support for the easy composition. Component composition techniques used in these models apply either direct or indirect message passing as connection schemes, which lead to tight coupling (i.e., components mix computation with control). Therefore, we propose the mediator connector which is similar to a communication hub. It initiates method calls and manages the returns and also provides loose coupling. Mediator connector is a framework and can be reused without any modification. The major contribution of this research is we have successfully defined no explicit connector and direct message passing between components and components are loaded into the framework dynamically during runtime based on the attachment. The attachment is the compositional configuration that defines the interaction between components, which is similar to Component Definition Language (CDL) in Koala component model or Architecture Definition Language (ADL). We illustrate the feasibility of mediator connector by building a simple bank system and evaluate the loose coupling by applying Coupling Between Objects (CBO) metrics.

Key words: Component composition, component models, software coupling, software connectors, deployment phase

INTRODUCTION

The design process in mechanical or electrical engineering disciplines is based on reuse of existing system or components. Engineers in these disciplines use the components that have already been tried and tested in other system, instead of reinventing the wheel.

Although the benefits of reuse have been recognized for many years, it is only the past 15 years that it has been an evolution from original software development to reuse-based development (Sommerville, 2004).

Many techniques have been developed to support software reuse during the past 20 years. Reuse is possible at different levels (from simple function to complete application). One of these techniques which have been emerged in the late 1990s as a reuse-based approach to software system development is the Component-based Software Development (CBSD). CBSE is the process of defining, implementing and integrating or composing loosely coupled and independent components (Szyperski *et al*., 2002) into systems.

Composition of software components is a fundamental issue in the component-based software development involving the process of creating component instances, configuring and assembled them together to form composite components or application. Unfortunately, almost all existing component models focus on the specification and packaging of components but provide almost no support for the easy composition of components (Selamat *et al*., 2007). There are many different techniques and approaches such as scripting languages (Ousterhout, 1998; Beazley, 1996; Wall *et al*., 2000), composition languages (Lumpe *et al*., 1997; Oscar and Meijler, 1995; Achermann, 2001; Weerawarana *et al*., 2001; Microsystems, 2001), wrapping (Heineman and Ohlenbusch, 1999), using software buses (Purtilo, 1994) and connectors (Shaw and Garlan, 1996; Mehta *et al*., 2000) for software component composition, which focus on composition of components to providing a specific functionality. Taxonomy of existing component models done by Lau and Wang (2005) reveals that none of these models support and composition in both design and

Corresponding Author: Hamid Sanatnama, Faculty of Mathematic and Computer Science, Shahid Bahonar University of Kerman, Kerman, Iran Tel/Fax: +981-341-3220059

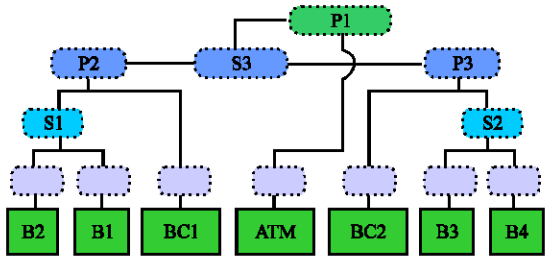

Fig. 1: Architecture of a bank example using exogenous connector (Lau *et al*., 2005)

deployment phase. The connection schemes used in these models fall under two categories: direct and indirect message passing, where as both of them lead to tightly coupling (Lau *et al*., 2005).

In order to minimize coupling and to maximize separation of control from computation, (Lau *et al*., 2005) proposed the exogenous connector (Fig. 1), which encapsulates control and data flow between components. Although comparison of exogenous connector with ADLs: Acme and C2 show that the former offers some advantages in the separation of control and communication, the obvious possible disadvantage of the exogenous connector is its significant connector levels that may lead to object overheat and communication inefficiency. In Fig. 1, we can see that B2, B1, BC1, ATM, BC2, B3 and B4 are computation units or components and the rest are compositional units in exogenous connector. The connector to component ratio is high and growing exponentially as number of components increases imply the inefficiency.

MATERIALS AND METHODS

Software components are sometimes compared to hardware components and the implication is that the composition of software components should be as easy as that to hardware components. When components are tightly coupled is hard to reuse them in other systems, since they depend on each other. Tight coupling also leads to monolithic systems, where one can't change or remove a class without understanding its design and dependency of other classes. The maintenance of such systems becomes very hard. In contrast, loose coupling increases the probability that a class can be reused by itself and that a system can be well-read, modified and extended much easier.

Design patterns proposed by Gamma *et al*. (1994) plays a pivotal role, when objects come to interact with each other. Among the design patterns, mediator pattern provides decoupling between objects.

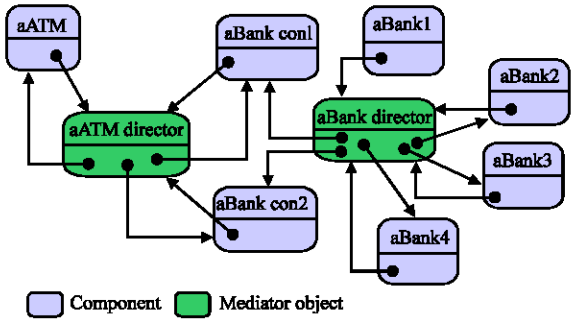

Fig. 2: A bank system using mediator design patterns

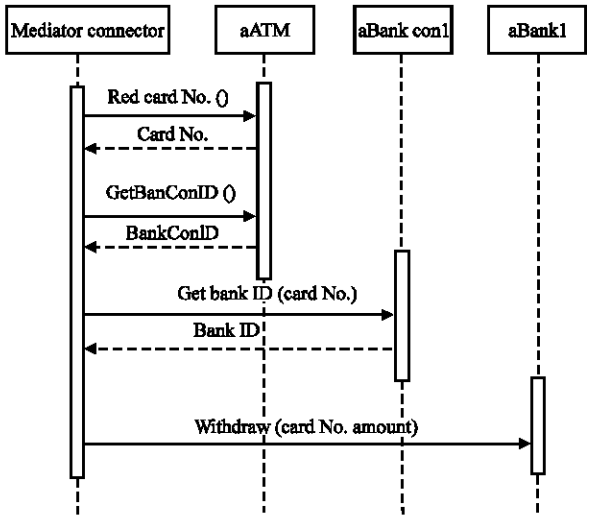

Fig. 3: Interaction diagram

With the mediator pattern communication between components is encapsulated with a mediator object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, hence reduce the coupling. Nevertheless, this indirect message passing technique still leads to tight coupling because its controls are originated from the components and not from the mediator object.

The concern of interaction or collaboration between components can be found when evolution of software engineering come along way from machine-level language to procedural programming and then to object-oriented programming and now to component-based software development. An interaction is a set of activities that happen for a specific use case in a system, based on the ability of components (requires and provides services) to send messages to each other.

The interaction between components can be different from one system to another. However, we consider the

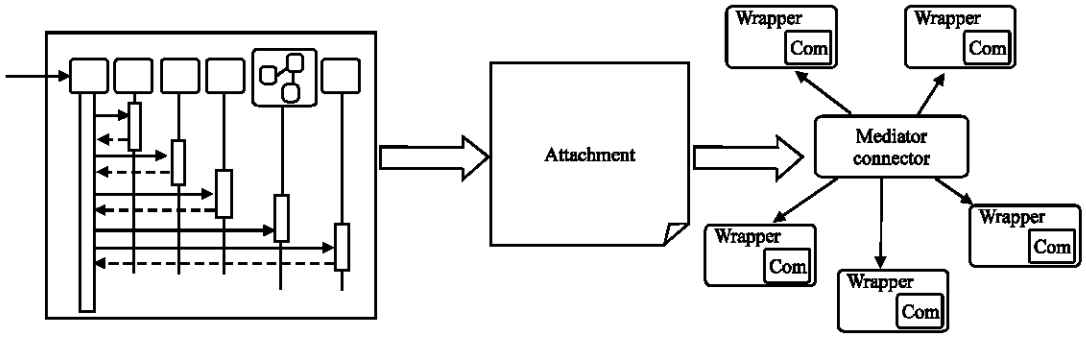

Fig. 4: The Process of building up a system using attachment and mediator connector

interaction between components as a separate issue in our model. An interaction is a sequence of method calls to components connected to a mediator connector which can be defined in design as well as deployment phase. Fig. 3 shows the interaction of ATM, aBankCon1 and aBank1 components are interacting via mediator connector. There is no direct message passing in between these components.

Mediator connector: In this section, we introduce mediator connector as a composition operator for loosely coupled software components. The mediator connector is inspired by the mediator design pattern, in which the relation between components can be many-to-many or intersecting.

The composition of components using mediator connector is based on interactions between components as a subset of behavior in a system. The interactions are described in an attachment which is similar to Component Definition Language (CDL) in Koala component model (Van Ommering *et al*., 2000) or Architecture Description Language (ADL). The description of each interaction is a usage scenario of a potential way the system is used. An interaction description is also used to explore the logic of an operation, function and method.

Mediator connector initiates and coordinates method calls to the components and handles their results. Thus it encapsulates communication. From another point of view, mediator connector is similar to a communication hub. The control originates from the mediator connector which leads to total loosely coupling between components and mediator connectors. Figure 4 shows the process of building up a system by using mediator connector. Interaction diagrams uses only for definition of interaction in attachment (i.e., no coding in application).

For each specific use case in the system, the component interaction are described (conforming the attachment syntax) in the attachment. Mediator connector will parse the attachment and build up the system by

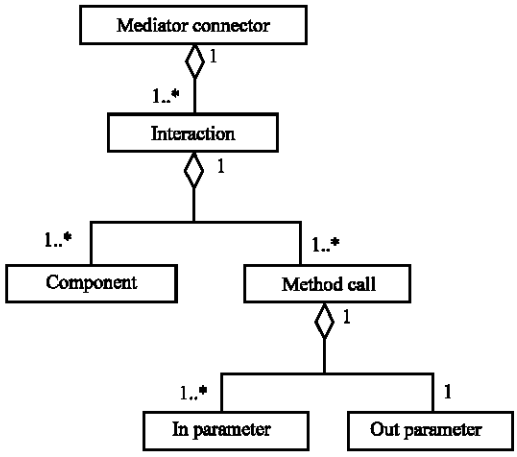

Fig. 5: Mediator connector design

initiating all the components and the connections (method calls) described in each interaction. Each interaction (sequence of method calls) can be run by invoking the run method in mediator connector giving the name of interaction as in-parameter.

Implementation: In all interaction descriptions, the following objects are typical: interaction, components, messages (method calls) and possible parameters (in/out). Thus in order to implement our mediator connector as a framework, we designed the mediator connector as shows in Fig. 5.

The mediator connector is implemented in Java programming language which is as good as any object oriented language. In our composition method, a component is semantically a Java class with (i) provides services (public methods), (ii) code that implements the provide services, however components do not request services in other components. Actually, mediator connector will invoke their provide services externally. Figure 6 shows the class diagram of mediator connector and the relations between the classes.

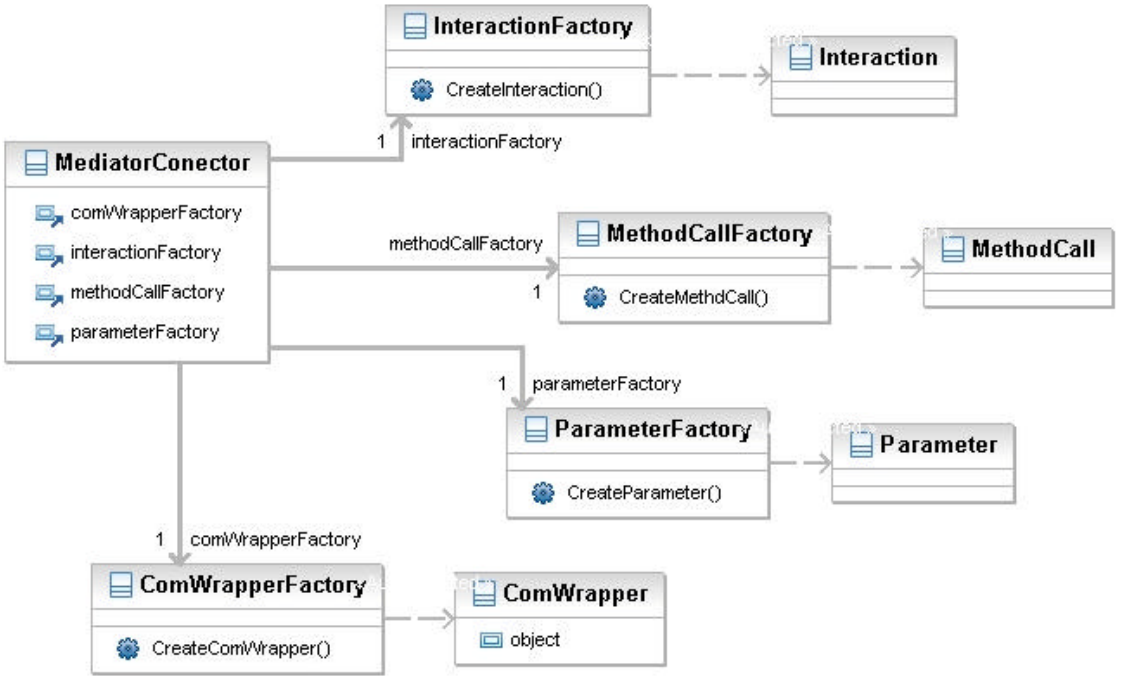

Fig. 6: Mediator connector class diagram

We may have multiple interactions in an attachment and a component or a method call repeats many times. To avoid instantiating and allocating a huge number of the same components as well as other objects, we use Flyweight design pattern which share a limited number of those objects. This can be detrimental to a system's performance. Mediator connector does not directly instantiate components; instead it gets them from a factory. The factory first checks if it has a flyweight component that fits request specific criteria (e.g., a unique id and class name); if so, the factory returns a reference to that component. If the factory can't locate a component for the specified criteria, it instantiates one, adds it to the pool and returns it to the client.

Since components are not allowed to send a message (method call) to the other components, components are wrapped in a ComWrapper (i.e., for each component in a system creates a ComWrapper object by ComWrapperFactory). This enables all the methods of a component to be invoked dynamically by ComWrapper using Java reflection.

Each interaction creates an Interaction object by InteractionFactory. Each interaction consists of a sequence of method calls to components with possible in-parameter(s) and out-parameter(s). Each method call is labeled by a class id and a class name, which indicates to which component it belongs to. Then for each method call definition, the mediator connector creates a MethodCall object by using MethodCallFactory, where the name of

```
/** creates a new instance of MediatorConnector */
   public MediatorConnector() {
      comWFactory = new ComWrapperFactory();
      methodCallFactory = new MethodCallFactory();
      interactionFactory = new InteractionFactory();
      parameterFactory = new  ParameterFactory();
      ... ... ... ...
   }
```

Fig. 7: Part of constructor code of mediator connector

the method and a reference to the component it belongs to is encapsulated. If a method call has in-parameters, Parameter object will be created by ParameterFactory and sets the references to indicate to which method call it belongs to. It also hold true for out-parameter.

The design of mediator connector makes it as a template which can be stored in a repository in the same way components are stored. The instance of mediator connector is created and deployed along with component instances.

Figure 7 shows the constructor code of mediator connector, where all factory objects are created.

Figure 8 shows the summarized of the code for building up structure of behavior in a system by parsing an attachment file. We have implemented a parser which is embedded in mediator connector. The parser receives an attachment file and creates proper objects with their references. An interaction can be run from anywhere

(such as an event in the user interface), with minimum of code writing as shown in Fig. 8.

A bank system example: Figure 9 is an attachment for a bank system with possible interactions and configurations.

Using the above attachment, we can illustrate the bank system's behavior by considering the operation of the Withdraw. The interaction starts when a customer intends to withdraw money from his account. The operation starts by calling the mediatorCon.run (Withdraw) as shown in the Fig. 8.

Then run method will get appropriate Interaction object which has a list of references to the MethodCall objects. Mediator connector goes through the list and for each MethodCall object it gets access to the reference of their ComWrapper object. Then each MethodCall objects will be sent as in-parameter to the execute method of their ComWrapper object.

In our Withdraw example the MethodCall objects are: ReadCard, getBankConId, getBankId and Withdraw.

Sometimes at the time of parsing (creation of method call instances) the reference of components to which they

```
//Creates a new instances of mediator connector
mediatorCon = new MediatorConnector();

//Parsing attachment file and building the system objects structure
      if(file !=null){
         mediatorCon.StartParser(file.getPath());
      }else{
         System.out.println("Couldn't parse the file :" + file.getName());
         return;
      }
……
//Running an interaction, giving the name of interaction
mediatorCon.run("Withdraw");
……
```

Fig. 8: Part of the code for the bank system using mediator connector

```
Attachment{

   Contains{
        Component = {"atm","ATM"}
        Component = {"bc1","BankCon"}
        Component = {"bc2","BankCon"}
        Component = {"b1","Bank"}
        Component = {"b2","Bank"}
        Component = {"b3","Bank"}
        Component = {"b4","Bank"}
   }
   Interaction = "Withdraw" {
        customerInfo= atm.ReadCard()
        BankConId  = atm.getBanConId(customerInfo)
        BankCon = mediator.getBankCon(BankConId, customerInfo)
        BankId = BankCon.getBankId( BankCon, customerInfo )
        Bank = mediator.getBank(BankId, customerInfo)
        Bank.Withdraw(customerInfo)
   }
   Interaction = "Deposit" {
        customerInfo= atm.ReadCard()
        BankConId  = atm.getBanConId(customerInfo)
        BankCon = mediator.getBankCon(BankConId, customerInfo)
        BankId = BankCon.getBankId( BankCon, customerInfo )
        Bank = mediator.getBank(BankId, customerInfo)
        Bank.Deposit(customerInfo)
   }
   ……
}
```

Fig. 9: A simple bank system example described in an attachment

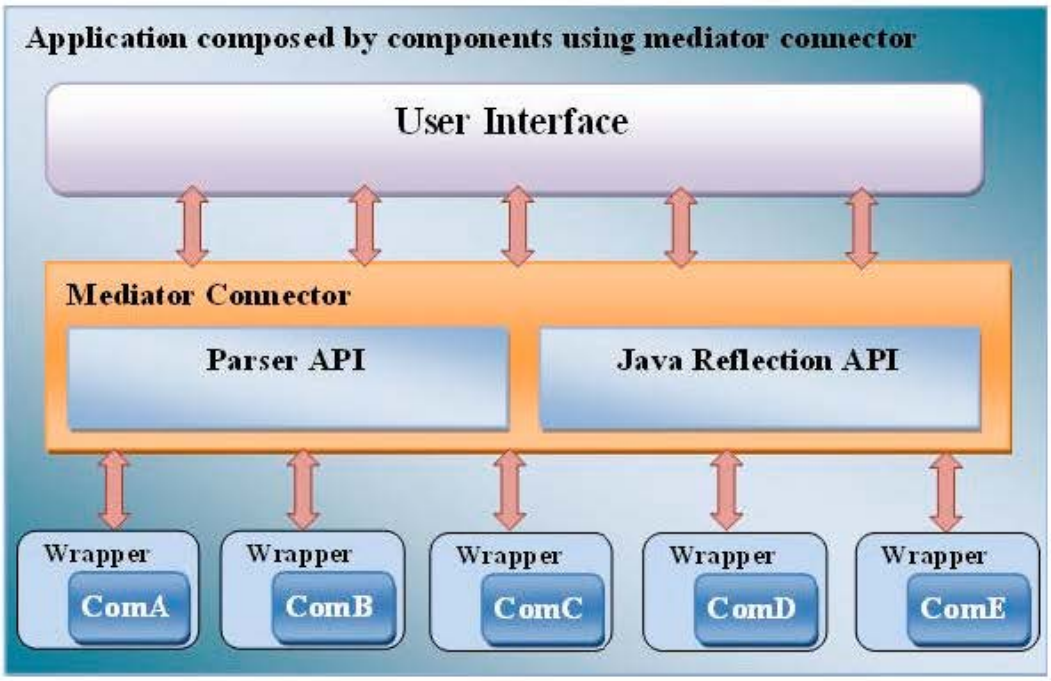

Fig. 10: System architecture using mediator connector as composition operator

should be sent to, is not identified. This will be done by mediator connector at the run time by using the result from previous calls.

Figure 10 shows an application architecture composed by loosely coupled components. Components are wrapped in the wrapper objects, which can interact with mediator connector and it's through user interface that we can interact with mediator connector.

Evaluation: The bank system example in the previous section demonstrates the feasibility of mediator connector in composing components based on their interaction definition. Composition of loosely coupled components makes it simpler for system maintenance and evolution (i.e. managing changes in both components and also mediator connector).

The most obvious advantage of the mediator connector is the use of attachment as the composition constructor. This provides a paradigm in software development, in which components can be easily unplugged and replaced due to the flexibility in defining their interaction independently. Our approach is also not far from the way the developers are used to develop a system (i.e. they don't need to learn a new programming language in order to use mediator connector).

To measure the coupling between components, which is the main objective of this research, we have selected Coupling Between Objects class (CBO = Number of classes to which a class is coupled) metric developed by Chidamber and Kemerer (1994) to measure the static coupling between components in order to show that mediator connector provides loose coupling.

Coupling is defined as: two objects are coupled if and only if at least one of them acts upon the other (Fenton, 1994). In other words, since coupling is the degree of interaction between classes, the basic idea underlying all coupling metrics is very simple: count how many interclass interactions there are in the system. Multiple accesses to the same class are counted as one access.

Suppose there is a system including a set of components $C = \{C_1, C_2, \ldots C_n\}$, where n is the number of components in the system. If Coup(i) is the CBO of C_i, then total CBO of components used in a system will be:

$$TCBO = \sum_{i=1}^{n} Coup(i)$$

A component may have a group of classes working together toward a common end; a subsystem, which shouldn't be counted as coupling compare to object classes. That's why we designed a measurement framework, which is more accurate in order to measure the coupling between components using mediator connector.

According to above definition if we assume that T_{before} is the total CBO of components before composition in a system, T_{after} the total CBO of components after

Table 1: Measurement values of CBO components in an aspect j control flow graph builder tool

AjcF graph builder tool	Measurement of T_{CBO}		
Components	T_{before}	T_{after}	T_{mc}
Build data	0	1	0
File loader	0	0	0
Graph builder	10	11	10
Splash screen	0	0	0
Main frame	19	22	20
Total	29	34	30

Table 2: Measurement values of coupling between components in a simple bank system

Simple bank system	Measurement of T_{CBO}		
Components	T_{before}	T_{after}	T_{mc}
ATM	0	1	0
Bank	0	0	0
Bank consortium	0	1	0
Teller machine	22	25	23
Total	22	27	23

composition in a system not using mediator connector and T_{mc} the total CBO of components after composition in a system using mediator connector. Then according to the CBO definition (multiple accesses to the same object are counted as one access) the total minimum coupling in a system using n components will be:

$$(T_{min}) = n$$

Finally the T_{CBO} in a system composed by n components can be defined as follow:

$$1 \le (T_{min}) \le | (T_{after} - T_{before}) |$$

The result of applying above measurement framework on four case studies is shown in Table 1-4.

Measurements in all four cases show that the T_{CBO} using mediator connector in a system is always:

$$| T_{mc} - T_{before} | = 1$$

A system composed by components using mediator connector has only one object reference and that is a reference to the mediator connector instance. On the other hand mediator connector does not have any reference to the components. All connections for message passing will be created at runtime according to composition definition in the attachment. This leads to totally loose coupling between components and mediator connector, which increase reusability of components. This means that there is no need of changing, modifying, or customizing the components in order to reuse them in other system.

Table 3: Measurement values of coupling between components in an MP3 player system

MP3Player	Measurement of T_{CBO}		
Components	T_{before}	T_{after}	T_{mc}
Play list manager	3	3	3
MP3 file	7	7	7
Main frame	10	12	11
Total	20	22	21

Table 4: Measurement values of coupling between components in a display map system

Display map	Measurement of T_{CBO}		
Components	T_{before}	T_{after}	T_{mc}
Address finder	1	1	1
Post code stripper	0	0	0
Scale reader	0	0	0
Mapper	1	1	1
Main frame	4	8	5
Total	6	10	7

```
package system;
import java.lang.reflect.*;
import java.io.*;
import connectors.*;
public class BankSystem {
public static void main (String[] args) {
// create instances of components
ATM atm = new ATM("1"); ... Bank bank4 = new Bank("4");
// create level-one connectors
Invocation invATM = new Invocation((Object) atm));……..
// create level-two connectors
Invocation[] invsBank12 = new Invocation[2];
Selector s1 = new Selector(invsBank12);
// create level-three connectors
Connector[] consBC1 = new Connector[2];
Pipem p2 = new Pipem(consBC1); ... Pipem p3 = new Pipem(consBC2);
// create level-four connectors
Connector[] consAB = new Connector[2];
Selectorm s3 = new Selectorm(consAB);
// create level-five connectors
Connector[] consm = new Connector[2];
Pipem p1 = new Pipem(consm);
……..
```

Fig. 11: Outline code of a bank system using exogenous connector (Lau *et al.*, 2005)

Comparison with exogenous connectors: The implementation of a bank system using exogenous connectors in Fig. 11, demonstrates the creation of all component instances. This leads to that the (T_{min}) in a system using exogenous connector is always equal to n.

All levels of exogenous connector are tightly coupled to each other and also invocation connectors to components (computation units). Adding or removing a component change the whole structure of connectors in the system.

CONCLUSION

In this study we have presented mediator connector for composition of loosely coupled software components based on the interaction between components. We also have presented its definition, implementation and showing its feasibility by building a small bank system using loosely coupled components. Mediator connector uses an attachment where the interactions between components are described. The attachment we have presented is at the preliminary stage; however we are working on its format and syntax in order to make it well-formed as a generic framework for describing the components and the interaction between them. So far we have only used sequential composition and the connector we presented belongs to the deployment phase. We are studying the ability of our approach for design phase, where the composite components can be constructed and deposit in a repository. We believe that our approach is a novel development methodology towards the component oriented development paradigm.

ACKNOWLEDGMENT

This is an ongoing research at UPM, Malaysia and is supported by eScience Fund SF0704, Ministry of Science Technology and Innovation, Malaysia.

REFERENCES

Achermann, F., M. Lumpe, J.G. Schneider and O. Nierstrasz, 2001. PICCOLA-a Small Composition Language. In: Formal Methods For Distributed Processing: A Survey of Object-Oriented Approaches Howard, B. and J. Derrick (Eds.). Cambridge University Press, New York, ISBN: 0-521-77184-6, pp: 403-426.

Beazley, D.M., 1996. SWIG: An easy to use tool for integrating scripting languages with C and C++. Proceedings of the 4th Annual USENIX Tcl/Tk Workshop, 1996, Monterey, California, pp: 15-15.

Chidamber, S.R. and C.K. Kemerer, 1994. A metrics suite for object oriented design. IEEE Trans. Software Eng., 20: 476-493.

Fenton, N., 1994. Software measurement: A necessary scientific basis. IEEE Trans. Software Eng., 20: 199-206.

Gamma, E., R. Helm, R. Johnson and J.M. Vlissides, 1994. Design Patterns: Elements of Reusable Object-Oriented Software. 2nd Edn., Addison-Wesley Professional, ISBN: 0201633612.

Heineman, G.T. and H.M. Ohlenbusch, 1999. An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Worcester Polytechnic Institute, Computer Science Department.

Lau, K.K. and Z. Wang, 2005. A taxonomy of software component models. Proceedings of 31st EUROMICRO Conference, 30 Aug.-3 Sept., IEEE Computer Society Press, pp: 88-95.

Lau, K.K., P. Velasco Elizondo and Z. Wang, 2005. Exogenous connectors for software components. Proceedings 8th International SIGSOFT Symposium on Component-based Software Engineering, LNCS., 3489, May 03, Springer Berlin/Heidelberg, pp: 90-106.

Lumpe, M., J.G. Schneider, O. Nierstrasz and F. Achermann, 1997. Towards a formal composition language. Proceedings of the ESEC'97 Workshop on Foundations of Component-Based Systems, ESEC'97, Schneider, Lumpe, pp: 178-187.

Mehta, N.R., N. Medvidovic and S. Phadke, 2000. Towards a taxonomy of software connectors. Proceeding 22nd International Conference on Software Engineering, June 04-11, ACM Press New York, USA., pp: 178-187.

Microsystems, S., 2001. Long-term Persistence for JavaBeans Specification. http://jcp.org/jsr/detail/57.jsp.

Oscar, N. and T.D. Meijler, 1995. Requirements for a composition language. Proceeding of the ECOOP '94 Workshop on Models and Languages for Coordination of Parallelism and Distribution. ECOOP 94, Springer-Verlag, pp: 147-161.

Ousterhout, J.K., 1998. Scripting: Higher-level programming for the 21st century. IEEE. Comp., 31: 23-30.

Purtilo, J.M., 1994. The POLYLITH software bus. ACM Trans. Programming Language Syst., 16: 157-174.

Selamat, M.H. and H. Sanatnama, A.A.A. Ghani and R. Atan, 2007. Software component models from a technical perspective. Int. J. Comput. Sci. Network Security, 7: 135-147.

Shaw, M. and D. Garlan, 1996. Software Architecture: Perspectives on an Emerging Discipline. 1st Edn., Prentice-Hall, Inc. Upper Saddle River, NJ, USA., ISBN: 0131829572 .

Sommerville, I., 2004. Software Engineering. 7th Edn., Addison-Wesley, New York, ISBN: 0321210263 .

Szyperski, C., D. Gruntz and S. Murer, 2002. Component Software: Beyond Object-Oriented Programming. 2nd Edn., Addison-Wesley, New York, ISBN 0-201-74572-0.

Van Ommering, R., F. Van der Linden, J. Kramer and J. Magee, 2000. The koala component model for consumer electronics software. Computer, 33: 78-85.

Wall, L., T. Christiansen and J. Orwant, 2000. Programming Perl. O'Reilly and Associates. 3rd Edn., O'Reilly and Associates, ISBN: 0596000278 .

Weerawarana, S., F. Curbera, M.J. Duftler, D.A. Epstein and J. Kesselman, 2001. Bean markup language: A composition language for JavaBeans components. Proceedings of the 6th USENIX Conference on Object-Oriented Technology System (COOTS 2001), January 2001, USENIX, San Antonio, Texas, USA., pp: 173-187.5.