Adapter Pattern in Component and Service Levels vs. Class and Object Levels

Kai Qian
Southern Polytechnic State Univ.
1100 Marietta Parkway
Marietta, GA 30060-2896
kqian@spsu.edu

Larry L. Wang
Southern Polytechnic State Univ.
1100 Marietta Parkway
Marietta, GA 30060-2896
lwang@spsu.edu

Subramanian Ananthram
Southern Polytechnic State Univ.
1100 Marietta Parkway
Marietta, GA 30060-2896
subramaniana@ ami.com

Abstract

This paper presents the adapter design pattern implemented in object-oriented class level, object level, component level, and service level (including local and remote components). The objective of this paper is to provide a survey on the decoupling quality attribute analysis of pattern-based software design, which is one of the most important criteria for software design.

The paper also presents the relationships between class, object, OO component, and service component in software development.

Practical application examples of adapter which adapts a standard Java Collection type are implemented and their UML diagrams and source code are also presented.

Keywords
Component, Adapter, Object, Service, Coupling.

1. Introduction

The quality attributes like interoperability, loose coupling, scalability, flexibility, extensibility, maintainability have become common issues in enterprise software design. Developers and designers alike are urged to think outside the box and come up with the best solutions that facilitate open architectures. Design patterns pave the way for such solutions.

"A Design pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it." [1]

Design patterns are broadly classified as creational, structural and behavioral. The focus of this paper is a flavor of the structural design pattern-Adapter pattern.

"A structural pattern helps you compose groups of objects into larger structures" [2].

An adapter pattern is a type of structural pattern. It facilitates conversion of the interface of a class into another interface that clients expect by providing a wrapper. Hence classes that would not have previously worked together now can work together seamlessly. [3][4]

We demonstrate the implementations of the adapter pattern at four levels: class, object, component, and service component. The example that we have chosen is an adaptor which adapts an adaptee of Collection type ArrayList to a target of queue type using the adapter pattern. We have chosen queue as our example since the Java standard edition 1.4 does not support queues. The adapter pattern enables us to simulate queue like behavior to the end user.

2. Coupling Metric

The following metric will be used to describe the coupling for class and object adapter.

$M = \sum w_i$ where w_i is the coupling estimate for service component interface(w_0), component interface(w_1), class-interface(w_2), object aggregation(w_3), and class inheritance(w_4) implementation and $\sum$ is the summation of all above couplings involved. They satisfy the following conditions in general

$$w_0 < w_1 < w_2 < w_3 < w_4 \text{ and } \sum_{i=0}^{4} w_i = 1.$$

The relation between w_i is based on the dependency between the adapter and its client. The interface weight W is lower because the interface plays a role of separation of its implementation and its client, In other word, the change

of implementation will not affect its clients. The smaller M is, the looser the coupling will be.

Well-partitioned software should have low coupling between its sub-elements.

3. Class adapter

3.1 Purpose

The class adapter converts the interface of a class (java.util.Arraylist) into another interface that the client expects. The class adapter implements the target interface and inherits the adaptee class by inheritance. It may need multiple inheritances supported.[5]

3.2 Description

The adapter (JClassAdapter) inherits from the existing adaptee (java.util.Arraylist) and implements from the desired target interface the client requires (i.e.) Queue interface. The UML class diagram is shown in Figure 1. The adapter may change some of the behavior of the adaptee by overriding some methods of the adaptee. This is generally used when the client wants full access to the adaptee's methods.The target interface Queue is shared by clients and the adapter.

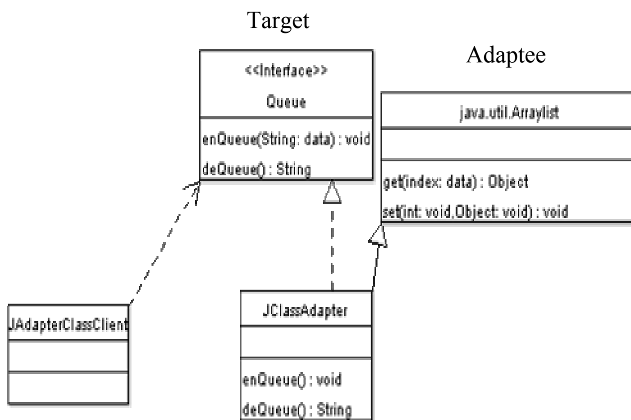


Figure 1. Class adapter.

3.3 Applicability

It is generally used in applications in which there is a need to use an existing class but the interface of the target class does not match. It has common methods from the target interface class (Queue) and the adaptee class (java.util.Arraylist).

3.4 Coupling

- This implementation introduces inheritance coupling between adapter and adaptee since the data members are inherited from the parent class and they are not redefined in the subclass. Data connections in parent and child classes are tight. This coupling is an inheritance coupling which is tighter compared with class interface coupling. The involved super and sub classes cannot be reused separately.
- There also exist a coupling between the client and the adapter since any changes in the adapter will also affect the client. Any revision of the designed adaptor software will result a recompilation of the client.
- The client needs to be aware of the adapter at build time and hence there is a static binding from client to the adapter.
- The coupling metric is given by $M_1 = w_2 + w_4$,

3.5 Advantages

- Changes of some adapted class methods will still allow the other methods to be unchanged.
- Lets you override the common methods from the interface class in the wrapper class to call the methods in the adaptee class as appropriate.
- This design results in an efficient run time performance.

3.6 Disadvantages

- Only the adaptee class can be adapted but its other subclasses cannot be used.
- This form of adapter is applicable only for languages that support multiple inheritances in some cases. It requires multiple inheritances.

3.7 Implementations

```
public interface Queue
{   // add item to the queue
    public void enQueue(String s);
      public String deQueue();
      public String peekQueue();
}
//Pattern: Adapter Class Pattern
import java.util.*;
```

```
public class JClassAdapter extends ArrayList
   implements  Queue
{   private int number=0;
     public JClassAdapter(){ number=0; }
   public void enQueue(String s)  {  add( number++ , s); }
   public String deQueue(){     . . .   }
   public String peekQueue() {   . . .  }}
//Client Program of Adapter pattern  with class model
public class JClassAdapterClient
{   private JClassAdapter list = new JClassAdapter();
     public JClassAdapterClient() {}
     private void putToQueue(String item){ . . . }
     private String getFromQueue() { . . . }
public  static void main(String argv[])
     {  JClassAdapterClient obj = new
JClassAdapterClient();   . . .  } }
```

4. Object adapter

4.1 Purpose

Object adapter converts the interface of a class (java.util.Arraylist) into another interface clients expect using delegation of responsibility. Instead of inheritance, the adapter itself instantiates an instance of adaptee class to complete the adaptation. [6]

4.2 Description

The adapter (JClassAdapter) implements a desired interface which clients requires (Interface Queue) and it holds the instance of the adaptee (java.util.Arraylist). The UML class diagram is shown in Figure 2. When the client (JAdapterObjectClient) calls any of the methods exposed by the target (Queue), the request is translated into a corresponding specific request on an instance object of adaptee (java.util.Arraylist).

4.3 Applicability

This is generally used when the client and the adaptee need to be completely decoupled from each other. In this form of adapter only the wrapper (JClassObjectAdapter) has understanding of both the client and the target. The multiple inheritance is required in any case for the object adapter design.

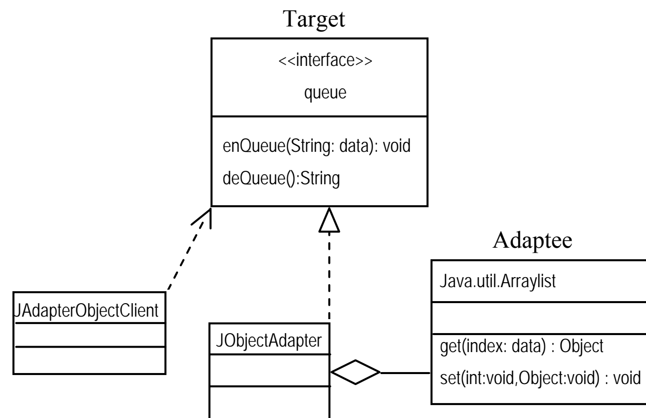


Figure 2. Object adapter.

4.4 Coupling

- Coupling exists between adapter and the adaptee is an aggregation coupling. The adaptee instance becomes a part of adapter.
- Since the adapter holds the instance of the adaptee the client is not exposed all the methods of the adaptee. In this way the adapter decouples the client from the adaptee.
- The client needs to be aware of the adapter at build time and hence there is a static binding to the adapter.
- The coupling metric is given $M_2 = w_2 + w_3$,
  So $M_2 < M_1$ since $w_3 < w_4$.

4.5 Advantages

- Since the object adapter does not use inheritance it holds an instance of the adaptee hence it can wrap the adaptee and any derivatives of the adaptee.
- Unlike the class adapter where all methods of the target are exposed to the client the object adapter may choose the desired methods that it wants to expose to the client.
- There is no restriction of multiple inheritance requirements.

4.6 Disadvantages

- It makes it harder to override the target behavior because it will require inheriting from the adaptee

and making the wrapper refer to the subclass rather than the adaptee itself.

- There still is an aggregation coupling although it may be looser than the inheritance coupling.

4.7 Implementations

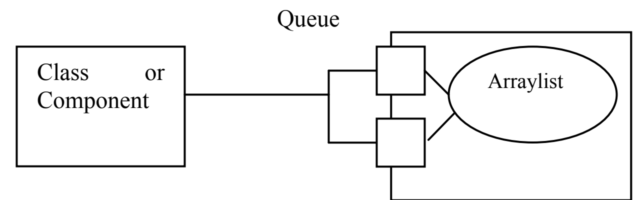
```
public interface Queue{ //Same as code in 3.7
}
import java.util.*;
public class JObjectAdapter implements Queue
{  //Holds instance of arraylist
   private ArrayList arrayList = new ArrayList();;
   private int number=0;
   public JObjectAdapter(){  number =0; }
   public void enQueue(String s) {  . . .  }
   public String deQueue(){ . . . }}
   //Client Program of adapter pattern using Object model
public class JAdapterObjectClient
{ private JObjectAdapter list = new JObjectAdapter();
   public JAdapterObjectClient() {  }
   private void putToQueue(String item)
   {     list.enQueue(item); }
   private String getFromQueue()
            {  return list.deQueue(); }. . .}
 public static void main(String argv[])
  {JAdapterObjectClient obj = new JAdapterObjectClient();
     //rest code same as  in 3.7 } }
```

5. Component level adapter

5.1 Local adapter

5.1.1 Purpose. Component adapter converts the interface of a class (java.util.Arraylist) into another interface clients expect by component-based design and implementation. Local adapter can only be used by a local client.

5.1.2 Description. The adapter bean is a java bean hence it implements the Serializable interface. The java bean holds an instance of the adaptee (java.util.Arraylist) and invokes methods on the java.util.Arraylist. See its component diagram in Figure 3.



Same Site (Local)

Figure 3. Local component adapter.

5.1.3 Applicability. This is generally used when a pluggable component is required. The pluggable component here is a java bean. The target java bean exposes a set of properties that can affect its state. Properties exposed may be accessed using a set of assessors and mutators, i.e., getters and setters.

5.1.4 Coupling.

- Changes to the component implementation will not require making changes to the client as long as the design information, also known as design environment, does not change. This makes a low coupling between the client and the target. It is called component interface coupling.[6]
- Local adapter provides low coupling between client and the component as any changes to the component will not affect the client as long as the manifest file and the API exposed by the component do not change.
- The coupling metric is given by $M_3 = w_1 + w_2$, So $M_3 < M_2$ since $w_1 < w_3$

5.1.5 Advantages

- The client and the target java bean are loosely coupled.
- The separation of interface and its implementation makes the software itself much more portable and usable.

5.1.6 Disadvantages

- This form of adapter does not support remote clients.
- Developing beans in currently available tools is often tedious. Java beans are slowly being replaced by enterprise java beans where ever appropriate.
- Component Deployment is required.

5.1.7 Implementations

```
package edu.spsu;
public interface Queue{     //Same as the interface in 3.7}
//Local Adapter Bean
```

```
package edu.spsu;
import java.util.*,  java.io.*;
public class AdapterBean implements Queue,Serializable
{   private ArrayList arrayList = new ArrayList();
   private int number;
   public AdapterBean(){   number =0;  }
   // enQueue,  deQueue(), peekQueue() same as in 4.7}}
//Client Program to access local component
import edu.spsu.*;
public class JCompClient
{   private AdapterBean list = new AdapterBean();
     public JCompClient() { } . . .}
   public static void main(String argv[])
   {  JCompClient obj = new JCompClient();
      . . .  } }//Manifest File is omitted here.
```

5.2 Remote adapter

5.2.1 Purpose Convert the interface of a class (java.util.Arraylist) into another interface clients expect by component-based design and implementation. Remote Adapter can be used by a remote client and supports better decoupling attributes for its clients.

5.2.2 Description The adapter class (AdapterSessionBean) is a stateful session bean that holds the instance of the adaptee (Arraylist) .We expose a remote interface (AdapterSessionRemote) to a client. The client invokes methods exposed by the remote interface to perform operations on the target (See figure 4).

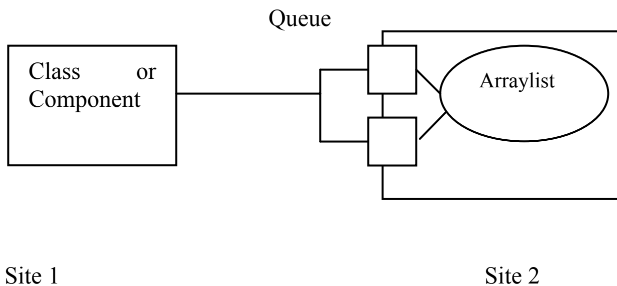


Figure 4. Component – remote adapter.

5.2.3 Applicability. This is primarily used in a distributed environment. The client performs a remote look up of the component at run time and gets back the remote interface AdapterSessionRemote. The client may then call the business methods exposed in the remote interface.

5.2.4 Coupling.

- There exists a low coupling between client and the component. The client access the component using the interface published by the component.
- Changes to the component will not require making changes to the client as long as the published interface remains unchanged .
- Client can bind to the component dynamically. Client only needs to be aware of the component name and the URL used for looking up the component.

5.2.5 Advantages.

- It is highly portable since the client and the component can be on different machines.
- A single component can be used to service the request of multiple clients.
- It supports dynamic binding at run time.

5.2.6 Disadvantages

- Call overhead for the client.
- Lower throughput in distributed scenarios

5.2.7 Implementations

```
package edu.spsu;
public interface Queue {   /*Same as the interface in 3.7*/ }
//Remote Adapter bean Home
package edu.spsu;
import java.rmi.*; import javax.ejb.*;
public interface AdapterSessionRemoteHome  extends
EJBHome
{      public abstract AdapterSessionRemote create()
       throws RemoteException, CreateException; }
//Remote Adapter bean Component
package edu.spsu;
import java.rmi.RemoteException;
import javax.ejb.EJBObject; import javax.swing.*;
public interface AdapterSessionRemote extends EJBObject
{public void enQueue(String s)   throws RemoteException;
    public String deQueue()   throws RemoteException;
    public String peekQueue()   throws RemoteException; }
//Remote Adapter bean
package edu.spsu;
import java.util.*, javax.ejb.*,javax.naming.InitialContext;
```

```
public class AdapterSessionBean
    implements SessionBean,Queue
{public AdapterSessionBean() {  }
   ArrayList arrayList; int number;
// enQueue, Queue(). peekQueue() are same as in 4.7
 // Default Session bean methods
    public void ejbCreate()
    { arrayList = new ArrayList();  number=0;   }
public setSessionContext(SessionContext  sessioncontext)
    {       SessionContext ctx = sessioncontext; }}
//Client Program to access remote component adapter
import edu.spsu.*, java.rmi.*,javax.naming.*, java.util.*;
public class JCompClient
  { private AdapterSessionRemote adapterSessionObj;
   public void JCompClient(){ }
   private void getContext(){
 // preparing properties for an InitialContext object
try {
        // Get a reference to the Bean and its home interface
       Object ref  = jndiContext.lookup("AdapterSession");
  // Create a AdapterSession object from the Home interface
    adapterSessionObj = home.create(); . . . }
    static public void main(String argv[])
   {  JCompClient obj =new JCompClient();
      obj.getContext(); // the rest is same code as in 4.7
   } } //The deployment descriptor snippet is omitted here.
```

6. Service level adapter

6.1 Purpose

Convert the interface of a class (java.util.Arraylist) into another interface clients expect by service-oriented component design and implementation. Grid Service can be used by any platform remote client and supports better decoupling attributes for its clients.

6.2 Description

The adapter class (QueueService) holds the instance of the adaptee (Arraylist). The methods that the client can invoke are exposed in a GWSDL file (Grid Web Services Description Language). The client can invoke methods exposed by this Queue service to perform operations on the target.

6.3 Applicability

This is primarily used in a distributed environment. The client performs a remote look up of the component at run time and gets back the remote interface QueuePortType. It works on the request-respond SOAP proposal. The client may then call the methods exposed by the service

6.4 GWSDL

The interface of a service level component is specified by the XML formatted GWSDL file which exposes the Service so that any SOAP enable client can access this service programmatically.

The following GWSDL interface file defines the service name as QueeService and all types of exposed operations, messages, and ports such as:

```
<types><xsd:element name="addQueue" type="xsd:int"/>
<xsd:element name="addQueueResponse">  . . .  </types>
<message name="AddInputMessage”> . . . </message>
<message name="AddOutputMessage”> . . . </message>
<portType name="QueuePortType"  . . .
<operation name="addQueue">
<input message="tns:AddInputMessage"/>
<output message="tns:AddOutputMessage"/> </operation>
</portType>
</definitions>
//Queue Interface:
import java.rmi.*,java.util.*,
org.globus.examples.stubs.QueueService_instance.*;
public interface Queue  {
public AddQueueResponse addQueue ( int a ) throws Exception;
public int deQueue ( DeQueue params) throws Exception ;}
// Queue Adapter : Service level Adapter
package org.globus.examples.services.core.first.impl;
//import all required packages
public class QueueService implements Resource,
ResourceProperties, Queue { // Resource Property set
private ResourcePropertySet propSet;
private int value; private String lastOp;
```

```
private ArrayList arr = new ArrayList();
 public QueueService() throws RemoteException {}
            // Get/Setters for the RPs
public int getValue() { return value; }
public void setValue(int value) { this.value = value; }
public String getLastOp() { return lastOp; }
public  setLastOp(String lastOp) {this.lastOp = lastOp; }
public AddQueueResponse addQueue ( int a )  throws
RemoteException { . . . }
public int deQueue ( DeQueue params) throws
RemoteException {   . . .  }
// Required by interface ResourceProperties
public ResourcePropertySet getResourcePropertySet() {
                        return this.propSet;  } }
//Client program
package
org.globus.examples.clients.QueueService_instance;
//import all required packages;
public class Client {
            public static void main(String[] args) {
            QueueServiceAddressingLocator locator =
             new QueueServiceAddressingLocator();
            try { String serviceURI = args[0];
// Create endpoint reference to service (omitted)
QueuePortType queue =
locator.getQueuePortTypePort(endpoint); . . . }}}
```

6.5 Coupling

Since the Grid service provides platform and language independency via. SOAP protocol and its interface is specified in XML format that loose the coupling between the client and service component very. The coupling metric is estimated by $M_4 = w_0 + w_2$. So $M_4 < M_3$ since $w_0 < w_1$.

7. Comparisons

The following figure shows the decoupling metric comparison of class inheritance, object aggregation, component interface, service-component implementation of the adapter design pattern.

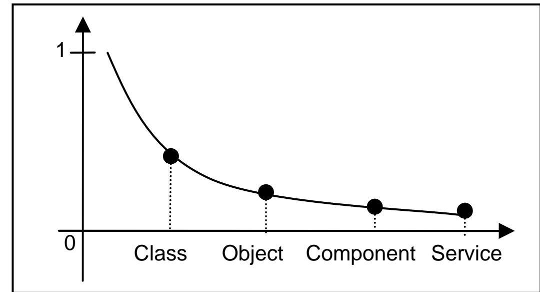


Figure 5. Coupling comparisons.

Also the decoupling attribute benefits the other software quality attributes such as system reusability, portability, interoperability, scalability, flexibility, extensibility, maintainability but here is quality attribute trade-off between these implementations. The better decoupling attribute may demote the system performance.

8. Conclusion

Class, object, component-based, and service-based implementations of a design pattern may result in a significant difference in terms of decoupling metrics between clients and designed software. The component-based and service-oriented implementations of a design pattern may lead much better decoupling and reusability attributes, but it may increase the software complexity [6][7].

9. References

[1] James W. Cooper, The Design Patterns. Addison-Wesley., Oct 2, 1998, 81-89.

[2] " Patterns Central". website: http://www.patternscentral.com/modules.php

[3] " Object Oriented Pattern Digest". website: http://patterndigest.com/patterns/ObjectAdapter.html

[4] Partha Kuchana, Software Architecture Design Patterns in Java, Auerbach.

[5] Sinan Si Albir, UML In A Nutshell. Oreilly, Sept 1998, 211-214

[6] Clemens Szypersk, “Component Software”, Addison-Wesly, 2002

[7] Andy Wang, Kai Qian, “Component-Oriented Programming”, Wiley, 2005