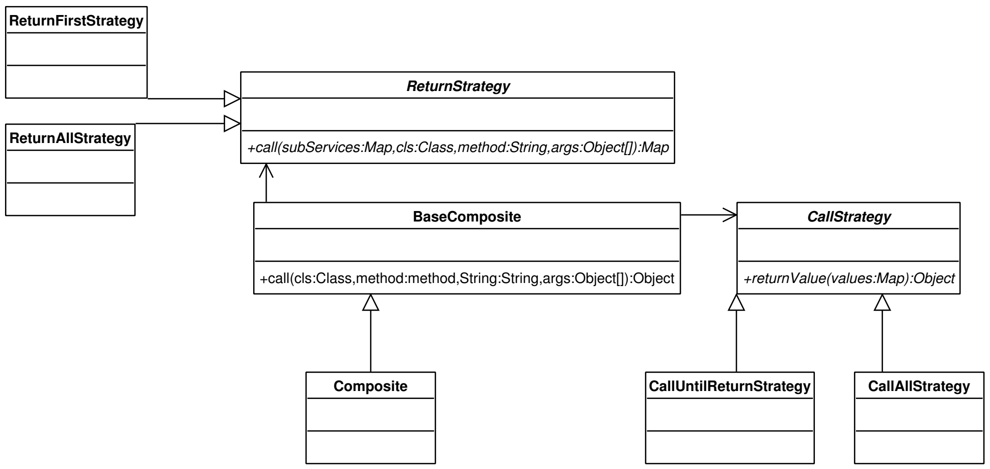
Composite Call/Return Strategy

William Lee
wwlee1@uiuc.edu
CS427 Pattern Essay

13th November 2003


ReturnFirstStrategy
ReturnAllStrategy
ReturnStrategy
+call(subServices:Map,cls:Class,method:String,args:Object[]):Map
BaseComposite
+call(cls:Class,method:method,String:String,args:Object[]):Object
CallStrategy
+returnValue(values:Map):Object
Composite
CallUntilReturnStrategy
CallAllStrategy

Eliminates redundant code in Composites that have complex multiplexing behavior.

1 Problem

Traditionally, Composites multiplex their calls to all its “children” or “sub-services.” When clients call the interface methods in the Composites, the methods would call the same methods for the children stored in the list. Although this model is simplistic and easy to understand, it also limits the power of the Composite. In particular, here are the limitations:

1. Composites often need to have custom code for each function to perform the multiplexing operations.
2. Composites are usually designed for write operations (functions that do not return any value). It works poorly on read operations (functions that returns values from the Composite’s children).

2 Context

Whenever a *Composite* tries to do more than simple multiplexing, it faces two problems above. For example, in a stock exchange system, the client may want to talk to a *Composite* in order to submit an order. In addition, the client also wants to retrieve the stock price from the same interface. For instance, when the buy order is submitted, the *Composite* needs to multiplex its calls to a RDBMS, ODBMS, and a notification service. However, in the read case (get stock price), the *Composite* may choose to only read from the RDBMS (certain stocks are only stored on the RDBMS, but not on the ODBMS). In a heterogenous environment, *Composite* may even need to make the following choices:

1. Retrieves the quotes from all services, and return the average price.
2. Retrieves the quotes from all services, and return all prices.
3. Retrieve the first quote returned from any services.

As we can see, there are many ways that a Composite can return a value. However, the implementation for each choices may be shared by a common infrastructure.

3 Forces

1. When a *Composite* extends beyond its default multiplexing behavior, custom code needs to be written. If you have multiple *Composites* that have similar multiplexing behavior, there may be duplicated code.
2. The implementation of a *Composite* is somewhat trivial. A base implementation is probably sufficient to implement most *Composite's* requirements. We want to share as much common code as possible among multiple *Composites* that may have different multiplexing behavior. It would be ideal to minimize the multiplex and values gathering code.

4 Solution

A *Composite* can use *Composite Call Strategy* to multiplex its call to its sub-services. Consequently, *Composite Return Strategy* returns a single value selected from the list of return values returned by each sub-services. *Base Composite* implements most functionality for a *Composite* and serves as the central coordinator.

When used together, a specialized *Composite*, which extends from the *Base Composite* class, would choose the *Call Strategy* and *Return Strategy* for each operation. The *Composite* then uses the *Base Composite*'s *call()* function (see the inital UML diagram for the pattern) to invoke the appropriate functions for the sub-services. The *call()* function in the *Base Composite* uses reflection to figure out the appropriate calls to make on the sub-services. The *Call Strategy* then determines the sub-services to multiplex to. In the simple case, the *CallAll* implementation of the *Call Strategy* will invoke the operation on all sub-services.

After the *Call Strategy* gathers a list of return values from each sub-services, the *Base Composite* will use the *Return Strategy* to pick from the return values to return. Note that the *call()* method in the *Call Strategy* returns a Map from the service name to object return. The Map would be passed to the *Return Strategy* to pick the right return value.

5 Rationale

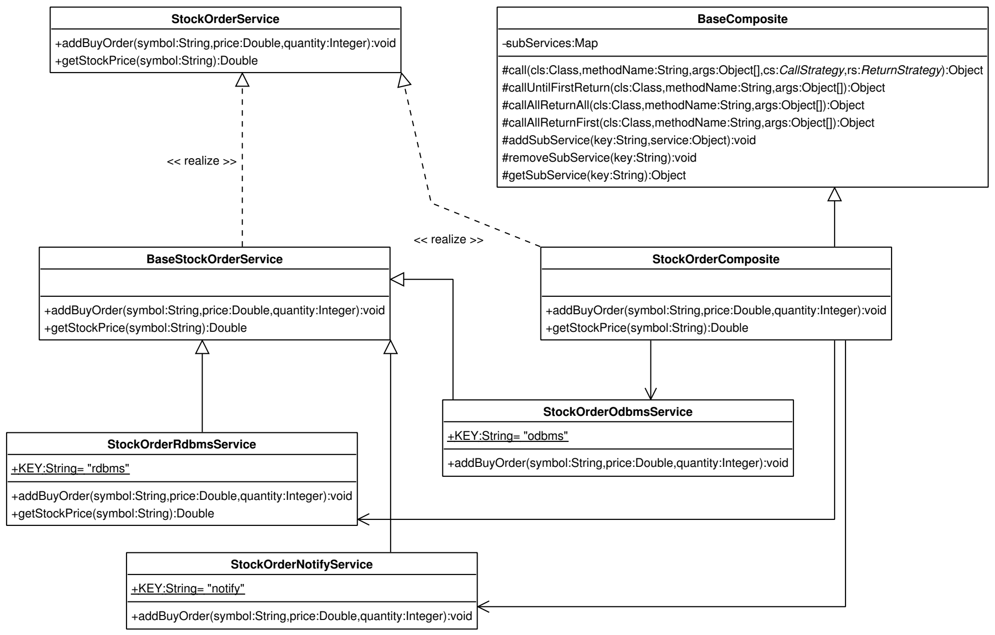
Composite Call/Return Strategy provides a very flexible framework and is especially appealing whenever there is a special case for multiplexing behavior. One can simply write a different *Call/Return Strategy* to change how the *Base Composite* call the sub-services and returns values from them.

One can even use a *Plugin*[1] pattern to implement *Base Composite.* Since it uses reflection, it can apply to all kinds of *Composites.* This means much code can be shared. Performance can also be improved when the *Composite* is not required to call all services in order to return a value.

6 Related Patterns

- *Plugin* can be used to implement *Base Composite*
- *Composite* is the main pattern that *Composite Call/Return Strategy* depends on.

7 Example


StockOrderService
+addBuyOrder(symbol:String,price:Double,quantity:Integer):void
+getStockPrice(symbol:String):Double
BaseComposite
-subServices:Map
#call(cls:Class,methodName:String,args:Object[],cs:CallStrategy,rs:ReturnStrategy):Object
#callUntilFirstReturn(cls:Class,methodName:String,args:Object[]):Object
#callAllReturnAll(cls:Class,methodName:String,args:Object[]):Object
#callAllReturnFirst(cls:Class,methodName:String,args:Object[]):Object
#addSubService(key:String,service:Object):void
#removeSubService(key:String):void
#getSubService(key:String):Object
<< realize >>
<< realize >>
BaseStockOrderService
+addBuyOrder(symbol:String,price:Double,quantity:Integer):void
+getStockPrice(symbol:String):Double
StockOrderComposite
+addBuyOrder(symbol:String,price:Double,quantity:Integer):void
+getStockPrice(symbol:String):Double
StockOrderOdbmsService
+KEY:String= "odbms"
+addBuyOrder(symbol:String,price:Double,quantity:Integer):void
StockOrderRdbmsService
+KEY:String= "rdbms"
+addBuyOrder(symbol:String,price:Double,quantity:Integer):void
+getStockPrice(symbol:String):Double
StockOrderNotifyService
+KEY:String= "notify"
+addBuyOrder(symbol:String,price:Double,quantity:Integer):void

Suppose we have a *StockOrderService* that modifies both a RDBMS and an ODBMS. In addition, we also want to send a notification to the auditor that is responsible to verify the order. We want to create a simple way to multiplex the interface calls to multiple services.

As an example, *StockOrderService* is defined as:

```
public interface StockOrderService
{
    public void addBuyOrder(String symbol, Double price, Integer quantity);
    public Double getStockPrice(String symbol);
}
```

We then want to implement a new Service called *StockOrderComposite* that extends from *BaseComposite*. This is an example on how one may implement *BaseComposite*:

```
public class BaseComposite
{
    // stores the sub-services
    private Map subServices = new Hashtable();
    // the generic call function that uses a CallStrategy
    // and ReturnStrategy
    protected Object call(
        Class cls,
        String methodName,
        Object[] args,
        CallStrategy cs,
        ReturnStrategy rs)
    {
        Map rtn = cs.call(subServices, cls, methodName, args);
        return (rs.returnValue(rtn));
    }
```

```
// Helper functions that children can use.  This will call until the
// first sub-service returns a non-null value, then it'll return
// that value immediately.
protected Object callUntilFirstReturn(
    Class cls,
    String methodName,
    Object[] args)
{
    return call(
        cls,
        methodName,
        args,
        CallStrategy.getCallUntilFirstReturn(),
        ReturnStrategy.getReturnFirst());
}

// Calls all sub-services, and return all values
protected Object callAllReturnAll(
    Class cls,
    String methodName,
    Object[] args)
{
    return call(
        cls,
        methodName,
        args,
        CallStrategy.getCallAll(),
        ReturnStrategy.getReturnAll());
}

// Calls all sub-services, but only return the first value
protected Object callAllReturnFirst(
    Class cls,
    String methodName,
    Object[] args)
{
    return call(
        cls,
        methodName,
        args,
        CallStrategy.getCallAll(),
        ReturnStrategy.getReturnFirst());
}
protected void addSubService(String key, Object service)
{
    subServices.put(key, service);
}
protected void removeSubService(String key)
{
    subServices.remove(key);
}
protected Object getSubService(String key)
{
    return subServices.get(key);
    }
}
```

The *StockOrderComposite* looks like. Note the use of different strategies for *addBuyOrder()* and *getStockPrice()*. It does it so that the *addBuyOrder()* call will multiplex to all sub-services, but we only need one service to return the value in *getStockPrice()*. The *Return Strategy* in the *getStockPrice()* returns the first non-null return value from any sub-service.

```
public class StockOrderComposite
    extends BaseComposite
    implements StockOrderService
{
    public void addBuyOrder(String symbol, Double price, Integer quantity)
    {
        Object[] args = { symbol, price, quantity };
        callAllReturnFirst(StockOrderService.class, "addBuyOrder", args);
    }
    public Double getStockPrice(String symbol)
    {
        Object[] args = { symbol };
        // Use just the RDBMS service to retrieve the price
        return (Double) callUntilFirstReturn(
            StockOrderService.class,
            "getStockPrice",
            args);
    }
}
```

The *CallStrategy* includes code for calling the sub-services using reflection:

```
public abstract class CallStrategy
{
    private static CallStrategy callAll = null;
    private static CallStrategy callUntilReturn = null;
    protected Class[] getClasses(Object[] args)
    {
        Class[] classes = new Class[args.length];
        for (int i = 0; i < args.length; i++)
        {
            classes[i] = args[i].getClass();
        }
        return classes;
    }
    public static CallStrategy getCallAll()
    {
        if (callAll == null)
            callAll = new CallAllStrategy();
        return callAll;
    }

    public static CallStrategy getCallUntilFirstReturn()
    {
        if (callUntilReturn == null)
            callUntilReturn  = new CallUntilReturnStrategy();
        return callUntilReturn;
    }
   public abstract Map call(
        Map subServices,
        Class cls,
        String methodName,
        Object[] args);
   /**
     * Uses reflection to invoke the methods stored for all the sub services
     * that it needs to multiplex to.
     */
    protected Object invokeCallOnService(
        Class cls,
        Object subService,
        String methodName,
        Object[] args,
        Class[] classes)
    {
        Method m;
        try
        {
            m = cls.getMethod(methodName, classes);
            return m.invoke(subService, args);
        }
        catch (Exception e)
        {
             // handles the exception
        }
        return null;
    }
}
```

The *CallAll* strategy will call all the sub-services:

```
public class CallAllStrategy extends CallStrategy
{
    public Map call(
        Map subServices,
        Class cls,
        String methodName,
        Object[] args)
    {
        Map rtnMap = new Hashtable();
        Class[] classes = getClasses(args);
        // Make calls to the sub services.
        for (Iterator i = subServices.entrySet().iterator(); i.hasNext();)
        {
            Object rtn = null;
            Map.Entry en = (Map.Entry) i.next();
            Object subService = en.getValue();
            rtn =
                invokeCallOnService(cls, subService, methodName,
                                    args, classes);
            if (rtn != null)
                rtnMap.put(en.getKey(), rtn);
        }
```

```
            return rtnMap;
        }
    }
```

For the *CallUntilReturnStrategy*, we would continue calling the next sub-service if the current one returns null:

```
    public class CallUntilReturnStrategy extends CallStrategy
    {
        public Map call(
            Map subServices,
            Class cls,
            String methodName,
            Object[] args)
        {
            Map rtnMap = new Hashtable();
            Class[] classes = getClasses(args);
            // Make calls to the sub services.
            for (Iterator i = subServices.entrySet().iterator(); i.hasNext();)
            {
                Object rtn = null;
                Map.Entry en = (Map.Entry) i.next();
                Object subService = en.getValue();
                rtn =
                    invokeCallOnService(cls, subService, methodName, args, classes);
                if (rtn != null)
                {
                    rtnMap.put(en.getKey(), rtn);
                    return rtnMap;
                }
            }
            return rtnMap;
        }
    }
```

Here is the *ReturnStrategy* class:

```
    public abstract class ReturnStrategy
    {
        public static ReturnStrategy returnAll = null;
        public static ReturnStrategy returnFirst = null;

        public abstract Object returnValue(Map valuesMap);

        public static ReturnStrategy getReturnAll()
        {
            if (returnAll == null)
                returnAll = new ReturnAllStrategy();
            return returnAll;
        }
        public static ReturnStrategy getReturnFirst()
        {
            if (returnFirst == null)
                returnFirst = new ReturnFirstStrategy();
            return returnFirst;
        }
    }
```

The *ReturnAllStrategy* just returns the Map that the call returns, *ReturnFirstStrategy* returns the first argument in the Map.

```
    public class ReturnFirstStrategy extends ReturnStrategy
    {
        public Object returnValue(Map valuesMap)
        {
            for (Iterator i = valuesMap.entrySet().iterator(); i.hasNext();)
            {
                Map.Entry me = (Map.Entry) i.next();
                return me.getValue();
            }
            return null;
        }
    }
    public class ReturnAllStrategy extends ReturnStrategy
    {
        public Object returnValue(Map valuesMap)
        {
            return valuesMap;
        }
    }
```

The sub-services performs the actual action. Note that the *addBuyOrder()* call would apply to all three services, but the *getStockPrice()* would returns the RDBMS stock price, since it is the only sub-service that does not return null:

```
    public class StockOrderNotifyService
        extends BaseStockOrderService
    {
        public static final String KEY = "notify";
        public void addBuyOrder(String symbol, Double price, Integer quantity)
        {
            // Notify the person who adds this order.
        }
    }
    public class StockOrderOdbmsService
        extends BaseStockOrderService
    {
        public static final String KEY = "odbms";
        public void addBuyOrder(String symbol, Double price, Integer quantity)
        {
            // Add the order in the ODBMS
        }
    }
    public class StockOrderRdbmsService
        extends BaseStockOrderService
    {
        public static final String KEY = "rdbms";
        public void addBuyOrder(String symbol, Double price, Integer quantity)
        {
            // Add the order in the RDBMS
        }

        public Double getStockPrice(String symbol)
        {
            // returns the stock price from the RDBMS.
        }
    }
```

The *BaseStockOrderService* serves as a stub for the sub-services:

```
    public class BaseStockOrderService implements StockOrderService
    {
        public void addBuyOrder(String symbol, Double price, Integer quantity)
        {
            // does nothing.  Children should override this.
        }
        public Double getStockPrice(String symbol)
        {
            // does nothing.  Children should override this.
            return null;
        }
    }
```

In order to use the *Composite,* one can do:

```
        public static final void main(String[] args)
        {
            StockOrderComposite som = new StockOrderComposite();
            som.addSubService(
                StockOrderRdbmsService.KEY,
                new StockOrderRdbmsService());
            som.addSubService(
                StockOrderOdbmsService.KEY,
                new StockOrderOdbmsService());
            som.addSubService(
                StockOrderNotifyService.KEY,
                new StockOrderNotifyService());
            StockOrderService sos = som;
            sos.addBuyOrder("IBM", new Double(39.2), new Integer(1000));
            Double stockPrice = sos.getStockPrice("IBM");
        }
```

References

[1] Martin Fowler. *Patterns of Enterprise Application Aarchitecture*. 2003.