URL: http://www.elsevier.nl/locate/entcs/volume82.html 13 pages

The development of generic definitions of hyperslice packages in Hyper/J

Youssef Hassoun [1]

School of Computer Science and Information Systems
Birkbeck College, University of London
London, United Kingdom

Constantinos A. Constantinides [2]

School of Computer Science and Information Systems
Birkbeck College, University of London
London, United Kingdom

Abstract

In this paper we investigate the notion of reusability of aspect definitions. We discuss the development of generic aspects in Hyper/J and compare it with the AspectJ approach. In doing that, we follow the design principle of "developing with hyperslice packages" and we show that hyperspace structure, concern mapping, hyperslice defintions and merging stategies exhibit well-defined patterns. An approach to constructing and merging generic aspects with base core concerns in Hyper/J is presented.

1 Introduction

Applying well-established principles of software engineering in order to improve separation of concerns, like modularity, abstraction, anticipation of change and incremental development will increase the level of reusability and adaptability and consequently the quality of software. AspectJ[3] [2] [7] and Hyper/J [9] seem to be two of the most notable technologies that provide mechanisms to support a disciplined way to applying the principle of separation of concerns. However, each technology follows a different approach.

[1] Email: yhassoun@dcs.bbk.ac.uk
[2] Email: cc@dcs.bbk.ac.uk
[3] AspectJ is a trademark of PARC.

On one hand, AspectJ is a general-purpose aspect-oriented language[4] and it constitutes an extension to the Java[5] language by providing new language constructs to explicitly capture crosscutting concerns (aspects). In the AspectJ language one models aspects as separate modules that are combined with components (regular Java classes) at compile-time using a weaver tool. There are three general constructs supported by AspectJ:

- Joinpoint specifications refer to well defined points in the execution of the program. Joinpoints may be grouped into pointcuts.
- An advice forms an action to be performed once a corresponding set of joinpoints is reached.
- Introductions may enhance components with state and behavior.

On the other hand, Hyper/J does not offer new language constructs, but it follows a different approach to support multi-dimensional separation and composition of concerns. Developers build regular Java classes by choosing the most appropriate decomposition for a program. Java classes are then combined at compile-time by specifying a series of composition rules.

Of interest to this study is the notion of reusability of aspect definitions. The importance of reuse lies on the fact that it can speed up the development process, it cut down costs thus increasing productivity as well as it can improve the quality of software.

The popularity of AspectJ resulted in a relatively large number of proposals to provide a higher level of reuse of aspect definitions, ranging from proposals to modify the language (the structure of aspect definitions) to language frameworks[6]. In [6] we discussed the problem of visibility of aspect definitions over components in the context of AspectJ by presenting different types of visibility relationships, and proposed a language framework that decreases the level of coupling between aspects and components.

In this paper we will address the development of generic reusable aspects (hyperslice definitions) in the context of Hyper/J. The rest of the paper is organized as follows: Section 2 illustrates a motivating example that involves an aspect that introduces bean behavior to a component, where we show how aspect reusability is addresed by AspectJ. In Section 3 we take the example of the previous section and we discuss the notion of generic aspects in the context of Hyper/J. In Section 4 we discuss another example where we address the aspect of exception handling. In Section 5 we provide a discussion on our approach. In Section 6 we discuss related work. We conclude this study in Section 7.

[4] The popularity of AspectJ and its influence over the design dimensions of other general-purpose aspect-oriented languages make it a representative technology for this study.
[5] Java is a registered trademark of Sun Microsystems.
[6] In [6] we present a literature survey of the different proposals to increase the level of reusability of aspect definitions in AspectJ.

2 Motivating example

Consider an example application where a requirement dictates that bean behavior is to be added to an abstract data type (a class) with setter and getter fields. Our goal would be to be able to catch and fire events of a change of state on an object whenever a `set` method is executed.

A traditional object-oriented solution to this requirement would be to insert the necessary code to fire the event in every setter function in order to inform all listener objects of the state change. Immediately one can realize that the outcome of this would be code scattering and code tangling, which is highly undesirable.

In [6] we proposed an AspectJ approach to implement this requirement in order to support a low level of coupling between aspect and component. The abstract aspect `BeanAspect` in Figure 1 includes an abstract pointcut and provides an advice where a change of state is fired based on calls to setter methods of any object that implements the `IPropertyChangeSupport` interface. Aspect `BaseBeanAspect` in Figure 2 inherits from `BeanAspect` and it is assigned two responsibilities:

- To provide a concrete pointcut that refers to the execution of any setter method on any arbitrary type `BaseObjectType` (part of the system core concerns)
- To extend the behavior of objects of type `BaseObjectType` by introducing bean behavior (semantics).

The AspectJ implementation supports a level of genericity in the sense that the realized functionality of catching and firing an event of a change of state is independent of the specifics of the clients (base code objects). The only provision for base objects is that they implement `IPropertyChangeSupport` which is handled by the introduction of this interface over the target class in AspectJ.

The question we now would like to address is how to implement bean behavior in pure Java (i.e. without adopting new language constructs as those provided by AspectJ) and how to merge this behavior with a base object using the composition rules of Hyper/J. We will address this question in the subsequent sections.

3 Hyper/J and generic aspects

To support multi-dimensional separation and integration of concerns in Java, Hyper/J introduces the notion of a hyperspace. In essence, a hyperspace is a multidimensional concern space where each dimesion represents one concern and contains the units needed to address that concern. A concern is a module, which deals with and encapsulates a particular area of interest whereas a unit

```
import java.lang.reflect.*;
public abstract aspect BeanAspect {
  abstract pointcut setter(IPropertyChangeSupport obj);
  void around(IPropertyChangeSupport p): setter(p) {
    String property =
          thisJoinPoint.getSignature().getName().substring("set".length());
    Method[] meths=p.getClass().getDeclaredMethods();
    for (int i=0; i<meths.length; i++) {
      if (meths[i].getName().toLowerCase().indexOf("set")!=-1) {
        Method getMeth=getGetMethod(p, property);
        Object oldVal=null;
        Object newVal=null;
        try {
           oldVal = getMeth.invoke(p, null); // getMethods have no parameters
           proceed(p);
           newVal=getMeth.invoke(p, null);
        } catch (java.lang.IllegalAccessException iae) {
           iae.printStackTrace();
        } catch (java.lang.reflect.InvocationTargetException ite) {
           ite.printStackTrace();
        }
        p.firePropertyChange(property, oldVal, newVal);
      }
    }
  }
  protected Method getGetMethod(IPropertyChangeSupport p, String property) {
    Method[] meths=p.getClass().getDeclaredMethods();
    for (int i=0; i<meths.length; i++) {
      if ((meths[i].getName().toLowerCase().indexOf("get")!=-1) &&
          (meths[i].getName().indexOf(property)!=-1)) {
          return meths[i];
      }
    }
    return null;
  }
}
```

Fig. 1. Aspect `BeanAspect` as a generic realization of catching and firing an event of a change of state

is a syntatctic constuct in Java, as for example, a method, an instance variable, a class or a package.

Generic aspects can be conceived as reusable pieces of code that can be added to a base program through a weaving mechanism (as in AspectJ) or through the specification of composition rules (as in Hyper/J). As a result the hyperspace involving such one aspect is, in its most general form, a two-dimensional space, where the aspect, as one concern, is represented by one dimension and the base code, as the sum of all units of the application components, is considered as one concern and is represented by the other dimension in the hyperspace. Figure 3 illustrates the structure of such a hyperspace. Note that the base code and the aspect code are also physically separated in

```
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;
public aspect BaseBeanAspect extends BeanAspect {
  pointcut setter(IPropertyChangeSupport obj): execution(* set*(*)) &&
                                              target(obj);
  declare parents: <BaseObjectType> implements IPropertyChangeSupport;
  PropertyChangeSupport <BaseObjectType>.support =
                                      new PropertyChangeSupport(this);
  public void <BaseObjectType>.addPropertyChangeListener
                                       (PropertyChangeListener listr){
    support.addPropertyChangeListener(listr);
  }
  public void <BaseObjectType>.removePropertyChangeListener
                                      (PropertyChangeListener listr) {
    support.removePropertyChangeListener (listr);
  }
  public void <BaseObjectType>.firePropertyChange (String p,
                                                  Object oldval,
                                                  Object newval){
  support.firePropertyChange(p, oldval, newval);
  }
}
```

Fig. 2. Pseudocode for a concrete aspect needed to weave in the advice code of its super abstract aspect

```
Hyperspace genericAspectBaseHyperspace
  composable class base.*;
  composable class genericAspect.*;
```

Fig. 3. Two-dimensional hyperspace including base and generic aspect code

```
package base : Feature.base
package genericAspect : Feature.genericAspect
```

Fig. 4. Base and generic aspect concern file

two locations, i.e. in two separate Java packages.

The concern mapping (Figure 4) reflects the existence of two concerns, the base concern and the aspect concern. Each mapping indicates that all classes and interfaces and all of their members that are part of the package address the one concern in the corresponding dimension. The concern mapping adds no new dimensions to the hyperspace. The base code is considered as one concern and we are interested in merging this concern with a generic aspect, which constitutes the other concern.

Figure 5 illustrates the definition of bean behavior in pure Java. The code wraps every `set` method with the corresponding `get` method and to subsequently fire the event with the obtained value.

We notice that in order to be able to use the generic function of Figure 5, the client object on the base side must be of `IPropertyChangeSupport`

```
pakage bean;
import java.lang.reflect.*;
public class BeanAspect {
  public void invokeMethod(IPropertyChangeSupport p,
    Method setMethod, Object[] params) {
    String property =setMeth.getName().substring("set".length());
    // use introspection to find the corresponding get method;
    Method getMethod= getGetMethod(p, property);
    Object oldVal=null; Object newVal=null;
    try {
      oldVal = getMethod.invoke(p, null);
      setMethod.invoke(p, params);
      newVal=getMethod.invoke(p, null);
      } catch (java.lang.IllegalAccessException iae) {
      } catch (java.lang.reflect.InvocationTargetException ite) {
    }
    p.firePropertyChange(property, oldVal, newVal);
  }
}
```

Fig. 5. Implementing base method calls wrapped within try-catch block

type. Since this is part of a bean feature, we need to add this type to our base as an abstract interface. Moreover, because every `set` method call is to be substituted by method `invokeMethod()`, the base client must provide a dummy implementation for this method. This way, we secure separation of concerns and guarantee “declarative completeness”, which is an important requirement to encapsulate concerns and to eliminate coupling between hyperslices. Declarative completeness is related to the notion of a hyperslice and it is supported by the mechanism of abstract declarations provided by Hyper/J. Declaring `IPropertyChangeSupport` as an abstract interface and providing a dummy implementation for `invokeMethod()` in the base hyperslice are examples of applying the mechanism of abstract declarations. This way, base and aspect hyperslices remain self-contained. Furthermore, each concern is confined to a Java package and at the same time it defines a hyperslice. In [9] a design approach with such hyperslice packages is referred to as “developing with hyperslice packages.”

Having established the hyperspace structure and concern mapping, we now address the question of defining the hyperslices and the composition strategies and rules that will merge them together. A hyperslice is a set of concerns that is declaratively complete, which means that it must declare everything which it refers to. Since we are dealing with only two concerns, it is reasonable to define two hyperslices, one for each concern (Figure 6). The next step is to write the Hyper/J hypermodule file to integrate the bean class code with the base code. A merging mechanism is proposed in Figure 6 (Note that `Feature.genericAspect` of Figure 3 is renamed to `Feature.bean` to fit the example implementation).

The concern mapping identifies two features to be composed, a base and

```
hypermodule beanHyperModule
  hyperslices:
    Feature.bean, Feature.base;
  relationships:
    mergeByName;
    equate class Feature.base.XPoint, Feature.bean.BeanAspect;
end hypermodule;
```

Fig. 6. Hypermodule to merge base and generic aspect code

a generic aspect. The code of each concern is confined to one package and implemented completely separately from the code of the other concern. Each package defines a hyperslice and encapsulates one concern. To merge both concerns, we first need to extend the `Point` class by adding bean behavior to it, which essentially amounts to implementing the interface `IPropertyChangeSupport`, and then to provide "dummy" implementation for needed methods, which marks them as required by the base hyperslice package but not defined within it.

Class `XPoint` is a base client of type `IPropertyChangeSupport`, which extends the abstract data type `Point` and provides a dummy implementation for method `invokeMethod()` (See Figure 7). Furthermore, since `IPropertyChangeSupport` is part of the aspect concern, a corresponding abstract interface type with the same name must be declared on the base side.

Calling a `set` method on an `XPoint` instance triggers a change of state and the concrete and generic method `invokeMethod()` of the aspect will be called with the actual parameters defined in the set method on the base side. Identifying the concrete method `invokeMethod()` and matching the concrete `IPropertyChangeSupport` with the corresponding method and abstract type in the other hyperslice is realized with the merging mechanism of Hyper/J.

To merge the base concern with the bean concern, we need to build a correspondence between the units `XPoint` and `BeanAspect` belonging to different hyperslices. The merging relationship `equate` is used to build such correspondence and the merging strategy `mergeByName` establishes the merging link between the two classes as well as between the abstract and concrete interface `IPropertyChangeSupport` types (Figure 6).

4 Addressing exception handling

In this section we consider the provision of exception handling in a Java program and we apply the same merging procedure of the previous section. We are particularly interested in `RuntimeExceptions` of Java, which can be thrown during the normal operation of the Java Virtual Machine. Essentially, such exceptions are bugs and the Java compiler does not enforce exception specifications for them. However, if the program is supposed to handle some of them, such as catching `NumberFormatExceptions` and `IndexOutOfBoundsExceptions` for methods involving vector or array operations and expecting an integer pa-

```
package base;
import java.beans.*;
import java.lang.reflect.*;
public class XPoint extends Point implements IPropertyChangeSupport
  protected PropertyChangeSupport sup=new PropertyChangeSupport(this);
  public void addPropertyChangeListener(PropertyChangeListener lis) {
    sup.addPropertyChangeListener(lis);
  }
  public void removePropertyChangeListener(PropertyChangeListener lis) {
    sup.removePropertyChangeListener(lis);
  }
  public void firePropertyChange( String pty, Object oval, Object nval) {
    sup.firePropertyChange(pty, oval, nval);
  }
  public void xsetx(int newX) {
    try {
      Method setx=getClass().getMethod("setx", new Class[]{Integer.TYPE});
      invokeMethod(this, setx, new Object[]{new Integer(newX)});
    } catch (Exception e) { e.printStackTrace();}
  }
  public void invokeMethod (IPropertyChangeSupport support,
                            Method setMeth, Object[] params) {
    throw new com.ibm.hyperj.UnimplementedError();
  }
}
```

Fig. 7. Class `XPoint` provides a dummy implementation of `invokeMethod()` on the base side

rameter, the code becomes tangled with repetitive `try-catch` blocks. To avoid this kind of code repetition, which is a consequence of the traditional object-oriented solution, we may follow the same aproach as with the previous example.

Initially, we look for a Java function, which wraps every method call (involving vector-array operations) in a `try-catch` block. The code is shown in Figure 8. Method `newInteger()` handles the cases where the input string is of a wrong format.

The hyperspace and concern mapping are similar to those of the bean example of the previous section. Figure 9 sketches a base class `Demo` using the services of the generic aspect class `RTExceptionHandling`, which represents the runtime exception handling concern.

Note that `Demo` provides dummy implementations of two methods, `invokeMethod()` and `newInteger()` and it includes no handling code for runtime `NumberFormatExceptions` and `IndexOutOfBoundsExceptions`. Method `setIndex()` needs to call method `initElementAt()` with an integer value, which is expected to be within the range of `1..10`.

The hypermodule file needed to integrate the base concern with the exception handling concern is similar to that of the previous section (Figure 6) and is illustrated in Figure 10. There are two hyperslices where each contains

```
package eh;
import java.lang.reflect.Method;
public class RTExceptionHandling {
  public void invokeMethod(Object obj, String mname,
                           Class[] ptypes, Object[] params) {
    try {
      Method method =obj.getClass().getDeclaredMethod(mname, ptypes);
      method.invoke(obj, params);
      } catch (Exception e) {
      System.out.println("Exception in " + mname + ": " + e);
      // handle the exception
    }
  }
  public Object[] newInteger(String param) {
    try {
      Object[] parameters=new Object[]{new Integer(param)};
      return parameters;
      } catch (Exception e) {
      System.out.println("Invalid parameter " + param": " + e);
      return new Object[] {new Integer(0)};
    }
    return null;
  }
}
```

Fig. 8. Generic methods to avoid tangled runtime exception handling code

one concern and both are self-contained. The base hyperslice needs to provide dummy implementations for methods `invokeMethod()` and `newInteger()`. The correspondence between the units `Demo` and `RTExceptionHandling`, each of which belongs to a different hyperslice, is realized by using the merging relationship `equate` and the merging strategy `mergeByName` establishing the link between the two units. This way, Hyper/J identifies the methods with dummy implementations on the base side with the corresponding concrete methods on the aspect side.

5 Discussion

In general, the approach to merge generic aspect code with an application code followed in the previous examples consists of two steps:

- Deploy the reflection API of Java in order to encapsulate generic behavior within an aspect class.
- Follow the design principle of "developing with hyperslice packages."

With the reflective capabilities of the Java language like introspection and method invocation, it is reasonable to assume that the first step is applicable in general. The second step involves the definition of a two-dimensional concern hyperspace and a concern mapping that builds the package structure one-to-one as two Features, where each concern is associated with one

```
package base;
import java.lang.reflect.Method;
public class Demo {
  public Demo() {}
  public void setIndex(String sint) {
    MyInteger mii=new MyInteger();
    Class[] paramtypes=new Class[]{Integer.TYPE};
    Object[] params=newInteger(sint);
    invokeMethod(mii, new String("initElementAt"), paramtypes, params);
  }
  public static void main(String[] args) {
    Demo de=new Demo();
    de.setIndex(new String("95"));
  }
  public void invokeMethod(Object o, String mname, Class[] ptypes,
    Object[] params) {
    throw new com.ibm.hyperj.UnimplementedError();
  }
  public Object[] newInteger(String param) {
    throw new com.ibm.hyperj.UnimplementedError();
  }
}
class MyInteger {
  Integer[] v=new Integer[10];
  ...
  public void initElementAt(int i) {
    v[i]=new Integer(0);
  }
}
```

Fig. 9. A base class using the exception handling package after merging

```
hypermodule expHandlingHyperModule
  hyperslices:
    Feature.base, Feature.eh;
  relationships:
    mergeByName;
    equate class Feature.base.Demo, Feature.eh.RTExceptionHandling;
end hypermodule;
```

Fig. 10. Hypermodule to merge base and generic aspect code of exception handling

hyperslice. Furthermore, to satisfy the requirement of declarative completeness of hyperslices, the base concern must, if necessary, be supplemented with abstract classes and it must provide dummy implementations of the generic methods of the aspect class.

We notice that the framework suggested in [6] is also applicable in case of Hyper/J as an implementation language providing a merging mechanism to support multi-dimensional separation of concerns. According to this framework and in the case of the bean example, the "aspect domain" is represented by the `BeanAspect` and is related to the "base domain" through the usage of the `IPropertyChangeSupport` type. Class `XPoint` establishes the contact between the two domains, first by being of IPropertyChangeSupport type (through the implements relationship) and second by providing a dummy implementation of the method invokeMethod().

In this study we followed the design principle of developing with "hyperslice packages" to keep the generic aspect and the base code separated. To satisfy the requirement of declarative completeness within this context, we needed to supplement the base code with abstract classes and methods. More specifically, in the case of the bean aspect, we had to provide the base package with an abstract data type IPropertyChangeSupport and a dummy implementation of the aspect method invokeMethod().

The integration mechanism of Hyper/J consists of a merging tool together with a script-language in order to specify the merging strategy and relationships between the various elements of the hyperslices. This mechanism seems to be relatively straightforward and it has the advantage of programming in one language (Java). However, Hyper/J is still under development and the existing limitations on some merging strategies and relationships lead to a decrease in the possible implementation choices. On the other hand, the joinpoint model of AspectJ provides a flexible and powerful weaving mechanism.

6 Related work

In [4] Chavez, Garcia and Lucena suggest a transformation algorithm from AspectJ into Hyper/J, where they explicitly specify the steps necessary to transform an AspectJ code into a Java code that fits the integration mechanism of Hyper/J. They also outline the structure of the hypermodule file. However, this algorithm is confined to some initial conditions that restrict its applicability to generic aspects. Among these conditions are, for example, the non-existence of around advices and the definition of the pointcuts being limited to the execution to single methods (with no wildcards).

In [5] Clarke and Walker stress the principle of separation of concerns throughout the software lifecycle. They consider the Observer design pattern as a composition pattern at the design level and implement it in AspectJ and Hyper/J at that level as a step before applying it to a library design application at the base level. The authors notice that the mapping of the Observer design pattern to Hyper/J as an implementation language reduces the reusability and extensibility of the code because of the restrictions imposed by the merging mechanism.

7 Conclusion

There are a number of different approaches to implement the principle of separation of concerns. In this paper we considered AspectJ and Hyper/J as possible tools that support the implementation of this principle. AspectJ is an extension to the Java language that provides linguistic constructs that can explicitly capture aspects. Hyper/J, on the other hand, provides a framework, where the system units, implemented in Java, can be distributed over different concerns and hyperslices. By specifying merging rules, the different concerns can be composed to provide the desired functionality. In this study we investigated the issue of using Hyper/J to separate and subsequently merge generic aspects with base code and compared this approach with AspectJ. Hyper/J supports a clean separation between generic aspects, as reusable pieces of code, and base code components. AspectJ, on the other hand, being a programming language with a powerful joinpoint model and new constructs provides a flexible mechanism to weave in aspect code into base components.

Among a large number of proposals, other aspect-oriented technologies include Composition Filters [3] and XML-based (DOM-tree) manipulation [8]. Interested readers may refer to the Aspect-Oriented Software Development (AOSD) community home page [1] that maintains links to AOSD resources.

Further developments to this study will include investigation of reusability issues into other approaches like the Composition Filters approach and the XML-based (DOM-tree) manipulation system.

References

[1] AOSD community home page located at http://aosd.net

[2] AspectJ home page located at http://www.eclipse.org/aspectj/

[3] Bergmans, B., and M. Aksit, *Composing multiple concerns using composition filters*, Communications of the ACM, 44(10):51–57, October 2001.

[4] Chavez, G. F. C., A. F. Garcia, and C. J. P. Lucena, *Some insights on the use of AspectJ and Hyper/J*, Tutorial and Workshop on Aspect-Oriented Programming and Separation of Concerns, Lancaster, August 23-24, 2001.

[5] Clarke, S., and R. J. Walker, *Mapping composition patterns to AspectJ and Hyper/J*, ICSE 2001 Workshop on Advanced Separation of Concerns in Software Engineering, Toronto, May 15, 2001.

[6] Hassoun, Y., and C. A. Constantinides, *Considerations on component visibility and code reusability in AspectJ*, 3rd Workshop on Aspect-Oriented Software Development, Essen, March 4-5, 2003.

[7] Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, *Getting started with AspectJ*, Communications of the ACM, 44(10):59–65, October 2001.

[8] Schonger, Stefan, Elke Pulvermüller, and Stefan Sarstedt, *Aspect-oriented programming and component weaving: using XML representations of abstract syntax trees*, 2nd Workshop on Aspect-Oriented Software Development, Bonn, February 21-22, 2002.

[9] Tarr, P., and H. Ossher, "Hyper/J User and Installation Manual", IBM Research, 2000.
URL: http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm