# Independent Extensibility – an open challenge for AspectJ and Hyper/J

Klaus Ostermann, Günter Kniesel

Universität Bonn
Institut für Informatik III
Römerstr. 164, D-53117 Bonn, Germany
{osterman,gk}@cs.uni-bonn.de

March 30, 2000

**Abstract**

AspectJ and Hyper/J provide different ways to modify classes that are used by existing clients. We think that a significant share of their differences can be attributed to the use of complementary ways to implement changes and to propagate them to clients: AspectJ uses *in-place modification* whereas Hyper/J uses *client migration*. Whereas these techniques are largely complementary in many respects, they share the fact that the combination of independent extensions of one base class poses major problems.

Object-oriented programmers frequently face one of the following problems:

- All subclasses of an existing class should exhibit some extended behavior.
- Objects created by existing code should exhibit some extended behavior.

As a rather simple example, consider a billing system with a central class `Invoice`. Invoices can be delivered by calling the `deliver()`-method in the `Invoice` class. Instances of `Invoice` are created in multiple locations of the system and multiple subclasses of `Invoice` exist. We will assume two users of `Invoice`, a subclass `SpecialInvoice` and a client `InvoiceClient`:

```
package billingsystem;

class Invoice {
  void deliver() { ... }
  ...
}
class SpecialInvoice extends Invoice {
 ...
}
class InvoiceClient {
  Invoice invoice = new Invoice();
  void foo() {
    invoice.deliver();
  }
}
```

Now assume that, within the scope of a shift to e-business, invoices should also be delivered by e-mail prior to normal delivery.

Neither subclassing nor wrapper-based solutions can be applied to this problem, because existing clients refer to the existing class instead of the subclass or the wrapper [Höl93]. So mainstream object-oriented systems only leave us with the choice to modify either the `Invoice` class or to modify its clients. Neither alternative is particularly appealing – it is generally agreed that changes should be incremental, rather than invasive, because invasive changes require source-code access and entail a major maintenance problem [OH92], [Höl93]. The maxime that "adding new code is good, modifying existing code is bad" dates back at least to [SLU89].

Meanwhile, AspectJ [Asp00] and Hyper/J [TO99] recommend themselves as two novel and promising approaches to cope with this problem.

## 1 AspectJ

We can tackle the problem with AspectJ by defining an aspect which has a `sendMail()`-Method and augments the `deliver()`-method of `Invoice` by a call to this method:

```
package billingsystem;

class MailExtensionAspect {
  introduction Invoice {
    void sendMail() {
   ...
    }
  }

  static advice(Invoice invoice): void print() & invoice {
    before {
       invoice.sendMail();
    }
  }
}
```

The result of a compilation of these classes is an `Invoice` class that is equivalent to writing the appropriate modifications directly into the class. We call this kind of modification an *in-place modification.*

## 2 Hyper/J

The same problem can be handled with Hyper/J by providing an independend class `extension.Invoice` for the mail-functionality. The "old" `Invoice` class together with its clients is then merged with the extension into a new hypermodule (for brevity reasons, the concerns and hyperslice specifications are left out, the definitions of our hyperslices below are just enumerations of the incorporated classes):

```
package extension;

public class Invoice {
      public void sendMail() {
        ...
      }
      public void deliver() {
        sendMail();
      }
}

hypermodule extendedbillingsystem
   hyperslices:
      Feature.billingsystem, // consists of billingsystem.*;
      Feature.extension;     // consists of extension.*
   relationships:
      mergeByName;
end hypermodule;
```

After performing the composition, the package `extendedbillingsystem` contains the classes `Invoice`, `SpecialInvoice` and `InvoiceClient`.

The interesting point in this case is that in the scope of the new package, `SpecialInvoice` and `InvoiceClient` *forward* [OKK+96] to `extendedbillingsystem.Invoice`. This means, that they use `extendedbillingsystem.Invoice` wherever `billingsystem.Invoice` was used previously. Instead of changing the *original* `Invoice` class, new clients are created that refer to a *new* `Invoice` class.

We call this kind of program transformation *client migration.*

## 3 Discussion

Both, in-place modification and client migration, can be used to solve our sample programming task. However, these solutions are not equivalent.

In-place modification is destructive in the sense that the original class is no longer available after composition. So in-place modification is appropriate if *all* known (and unknown) clients of this class should participate in the modification. If only selected clients should use the modified class, in-place modification fails.

Client migration is better suited for selective modification, because existing classes are not changed. On the other hand, this strategy imposes a maintenance problem, if the modified class should be used by all (existing and future) clients, because all clients have to be listed explicitly in the composition specification.

Additionally, client migration poses the question of how to handle clients of the migrated clients. Either these clients-of-clients are migrated, too, or they have to be invasively modified, or they continue to use the old version of the modified class. The first case can be iterated until the main entry into the system, e.g. the starting class, is migrated – in this case, the deployment has to be changed invasively (with of course less weighty consequences).

## 4 Independed extensibility

A useful property of an extensible component is *independend extensibility*: A system is independently extensible if it is extensible and if independently developed extensions can be combined [Szy96].

The difficulty in achieving independend extensibility when using in-place modifications is that different extensions might conflict and no central instance (like the inheriting class in a multiple inheritance hierarchy) is available to resolve this conflict.

AspectJ suffers from this problem in case of conflicting introductions, which result in compiler errors. One could be tempted to limit the visibility of introductions by using `public`, `protected`, `private` modifiers. However, this will only reduce the number of conflicts but it will not help to deal with the remaining ones, e.g. two conflicting `public` introductions.

Advice conflicts can be influenced by defining a partial order on the aspects, which introduces a low coupling among them. However, ordering aspects will often be too coarse grained, because an ordering that is appropriate for conflicting advices related to one particular method might be fully inappropriate for advices referring to another method. For instance, one might have an original class with methods `start()` and `end()` and two aspects: the one advices that a new method `openTheDoor()` should be invoked before `start()` and `closeTheDoor()` before `end()`; the other one advices that a new method `goInside()` should be invoked before `start()` and `goOutside()` before `end()`. Clearly, the only sensible sequence of events is: `openTheDoor()`, `goInside()` and `goOutside()`, `closeTheDoor()`, which corresponds to `different` orderings of the advices from different aspects.

A more subtle problem is related to updates: If an aspect updates state of its object, state hold or introduced by other aspects will in general become inconsistent.It is impossible to detect automatically, let alone to reconcile, different (implicit) invariants assumed by different aspects. Therefore, the only possible solution seems to be to let each aspect enforce its own invariants. This would be possible if aspects were informed about state changes in the shared object (e.g. as listeners of corresponding events). However, this is no general solution either because the actions taken by one aspect to enforce its invariants might lead to further changes of shared state that might lead to other actions by other aspects, and so on *ad infinitum*.

Similarly, Hyper/J suffers from difficulties when it comes to combination of extensions. This requires either the modification of a hypermodule specification (which is an invasive change) in order to insert another extension into the hypermodule, or the creation of a new hypermodule containing all clients, which introduces the client-of-client problem.

## 5 Conclusions

Although we have talked mainly about AspectJ and Hyper/J the essence of the discussion is the comparison of in-place-modification with client migration.

It turns out that AspectJ and Hyper/J can be viewed as powerful front-ends that automate the traditional solutions for changing existing classes. AspectJ performs in-place-modification, which invasively changes the original classes. Hyper/J performs client migration, which performs the changes in duplicates of the original class and of its clients.

Due to their invasive nature, independently developed aspects of one original class will possibly conflict with respect to introductions, advices, and state changes. We have shown that conflict resolution of advices by orderings of aspects is not general enough. There is no general solution for conflicting introductions either. Finally, the issue of conflicting state changes is notoriously difficult, too.

Hyper/J, on the other hand, does a good job in case only a fixed set of selected clients should be migrated. Composition of independent extensions would be possible in Hyper/J, if a solution to the client-of-client problem were available. Indeed, we think that there is an obvious solution: An optional migration / duplication of all classes in the transitive closure of the client-of relation, ie an automatic migration of all clients, clients of clients etc.

However, this would result in an exponential growth of class versions and program size. In addition, this feature requires a global view of the compiler: All application classes have to be known at the time of compilation (*closed-world assumption*, eg [PS94]), a problem that AspectJ suffers from, too.

## References

[Asp00] AspectJ homepage, 2000. http://aspectj.org.

[Höl93] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings ECOOP '93, LNCS*, 1993.

[OH92] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *Proceedings OOPSLA '92*, 1992.

[OKK$^+$96] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–292, 1996.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

[SLU89] Lynn Andrea Stein, Henry Lieberman, and David Ungar. A shared view of sharing: The treaty of orlando. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press and Addison-Wesley, 1989.

[Szy96] Clemens Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings 19th Australian Computer Science Conference*, number 18(1), pages 203–212. Australian Computer Science Communications, 1996.

[TO99] Peri Tarr and Harold Ossher. Hyper/J user and installation manual, 1999. http://www.research.ibm.com/hyperspace.