Conquering Aspects with Caesar

Mira Mezini
Darmstadt University of Technology
D-64283 Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

Klaus Ostermann
Darmstadt University of Technology
D-64283 Darmstadt, Germany
ostermann@informatik.tu-darmstadt.de

ABSTRACT

Join point interception (JPI), is considered an important cornerstone of aspect-oriented languages. However, we claim that JPI alone does not suffice for a modular structuring of aspects. We propose CaesA, a model for aspect-oriented programming with a higher-level module concept on top of JPI, which enables reuse and componentization of aspects, allows us to use aspects polymorphically, and introduces a novel concept for dynamic aspect deployment.

1. INTRODUCTION

A popular view of aspects is one of modules that define (i) points in the execution of a base program to intercept (joinpoints), and (ii) how to react at these points. We believe, however, that more powerful means for structuring aspect code are needed on top of join point interception (JPI), namely, better support (a) for expressing an aspect as a set of collaborating abstractions, comprising the modular structure of the world as seen by the aspect, and (b) for structuring the interaction between two parts of an aspect: aspect implementation, and aspect binding (integration) into a particular code base.

To clarify the terminology, let us consider a simple and well-known example: the subject-observer pattern [6]. As far as (a) is concerned: The world as seen by this aspect consists of two abstractions, subject and observer, which are mutually recursive in that the definition of each of them refers to the other one. The definition of the observer aspect should clearly define these two abstractions as two modules that interact with each other via well defined interfaces. As far as (b) is concerned: The implementation part comprises in this case the implementation of methods such as addObserver(), removeObserver() and changed(), say by means of a LinkedList. Of course, other implementations are possible, e.g., one that executes the observer notifications asynchronously, or one that employs buffering to eliminate duplicated notifications. The binding part, on the other hand, comprises details about how to integrate the observer protocol into a particular context mapping the roles “Subject” and “Observer” to particular application classes, e.g., JButton and MyActionListener. An example for such binding details would be the extraction of the part of the subject state (e.g., JButton) to be passed over to the observers along a change notification, as well as how the notification is performed in terms of the method to call on the observer site.

The advantage of supporting the definition of an aspect as a set of mutually recursive abstractions that interact via well-defined interfaces is more or less a direct derive of the advantages of the object-oriented approach to modeling a world of discourse; for this reason it does not require particular justification at this stage of the discussion.

A short discussion is needed, though, to justify the requirement for decoupling aspect implementation from aspect binding. An aspect implementation that is tightly coupled with a particular aspect binding, by the virtue of being defined within the same module, cannot be reused with other possible bindings. Hence, this particular aspect implementation must be rewritten for every meaningful binding, thereby rendering the application tangled, since the aspect implementation becomes itself crosscutting. Especially for non-trivial aspects with complex implementations, this rewriting of the aspect implementation is tedious and error-prone.

An aspect binding that is tightly coupled to a specific aspect implementation is also undesirable. A binding translates the concepts, terms, and abstractions of the application’s world into the world of the particular aspect domain; its usage is not limited to a specific aspect implementation. Consider e.g., an aspect binding that transforms a particular business application data model to the domain of graphs with nodes and edges. Such a graph view is useful with different graph algorithms.

Without dedicated language support it is rather difficult to separate aspect implementation and binding properly. We will elaborate on this claim in Sec. 2, where we investigate the AspectJ approach to separation of aspect implementation and binding via abstract aspects. The discussion in Sec. 2 will also reveal the deficiencies of AspectJ’s JPI-based approach with respect to modeling multiple mutually recursive abstractions.

To solve these problems, we propose the CaesA model in Sec. 3, which is based on the notion of collaboration interfaces (CI) presented in [11] as a means to better support a-posteriori integration of independent components into existing applications. We show that CIs and the related no-
In this section we discuss the deficiencies of a JPI-based approach to aspect structuring. Please note that the discussion in this section is by no way a critique on the notions of JPIs and ad\-\ces. On the contrary, recognizing them as pivotal concepts of aspect-oriented languages, we emphasize the need for higher-level module concepts on top of them.

For illustrating the problems, we use as an example the implementation of the observer pattern in AspectJ proposed in [7] by Hannemann and Kiczales, as shown in Fig. 1 and Fig. 2, whereby Fig. 1 shows a reusable implementation of the observer protocol in AspectJ, while Fig. 2 binds it to particular classes.

The basic idea in Fig. 1 is that the aspect ObserverProtocol declares an abstract pointcut that represents change events in the Subject classes. The empty interfaces Subject and Observer are marker interfaces that are used in the binding to map the application classes to their roles. The observers for each subject are stored in a global WeakHashMap (the weak references are required in order to prevent a memory leak) that maps a subject to a list of observers. In case of a subject change all observers are notified by means of the abstract method updateObserver(), which is overridden in the binding aspect in order to fill in the appropriate update logic.

This proposal has two main advantages. First, Fig. 1 is indeed a reusable implementation of the observer protocol: Nothing in the implementation is specific to a particular binding of this functionality. This is because the authors [7] recognize the need to separate aspect implementation and aspect binding. Second, the same role, e.g., Subject, can be mapped to multiple different classes, e.g., Point and Line as in Fig. 2. It would also be no problem to assign two roles, e.g., Subject and Observer, to the same class, or assign the same role twice to the same class in two different bindings. For example, a Point can be simultaneously a subject concerning coordinate changes ias well as color changes. In terms of [14], the observer “component” in Fig. 1 is independently extensible.

These features are probably the rationale for the author’s decision against an alternative (simpler) implementation of the observer protocol in AspectJ. The alternative solution of which we speak is to declare addObserver() and removeObserver() in the interface Subject and then (in the binding) inject these methods into the corresponding classes by means of a so-called introduction. - AspectJ’s open class mechanism. Similarly, a LinkedList could be introduced into every Subject class, thereby rendering the perSubj- ectObservers map unnecessary. However, with this solution, a class could not have two different instantiations of the Subject role, because then the class would have multiple implementations of the same method (e.g., addObserver()), hence resulting in a compiler error. In other words, we would lose independent extensibility.

Now, let us take a critical look on this solution. We identify the following problems.

Lacking support for multi-abstraction aspects

Note that all methods in Fig. 1 and 2 are top-level methods of the enclosing aspect class. For example, addObserver(), which is conceptually a method of the subject role, is a top-level method whose first parameter is the respective Subject object. This design is conceptually questionable leading to a poor separation of concerns inside the aspect: The enclosing class contains all methods of all abstractions that are defined in the particular aspect and therefore becomes easily bloated. In a way, this is a rather procedural style of programming, contradictory to one of the fundamentals of object-oriented programming, according to which a type definition contains all methods that belong to its interface. It
is also contradictory to the aspect-oriented vision of defining crosscutting modules in terms of their own modular structure. The structure of the aspect in Fig. 1 is one of empty abstractions and unstructured method definitions, and as such not particularly modular.

The implications of this design decision are not only of a conceptual, but also of a practical nature. First, we cannot pass objects that play a role to other classes that expect an instance of that role. Envisage, for illustration, a role \texttt{Composite} with a method \texttt{compareTo()}. If we want to pass an object as a \texttt{Comparable} to another class, e.g., a sorting class, then the approach in Fig. 1 and 2 based on introducing an empty interface and encoding all methods as top-level methods of the enclosing class, does not work. The alternative would be to use AspectJ’s introduction mechanism to introduce the interface and its methods directly into the respective class but then again we would loose independent extensibility, as discussed above. For example, a \texttt{Point} could be compared to another \texttt{Point} by means of their geometrical distance $\sqrt{x^2 + y^2}$ as well as their Manhattan distance $|x| + |y|$ to the origin, which would require two independent implementations of the \texttt{Comparable} abstraction.

A similar problem shows up, if some interaction between the abstractions that build up the aspect’s model of the world - \texttt{Subject} and \texttt{Observer} in our example - is needed. The interaction in Fig. 1 is very simple: a subject passes itself on calling the \texttt{notify} method on each observer, but the parameter gets never used in the binding of the aspect in Fig. 2. It is more realistic that observers would want more detailed information of what state change actually happened on the subject’s site. This would require some query methods in the interface of the subject. Using the AspectJ’s introduction mechanism to “pattern” exemplified in Fig. 1 and Fig. 2, where abstractions are typeless, we would have to declare such query methods also at the top level, e.g., \texttt{getState(Subject s)}. The query methods would have to be declared abstract in Fig. 1 since their implementation is binding specific and should be implemented by the concrete binding subsaspect in Fig. 2. However, it is not possible to implement different query methods for \texttt{Point} and \texttt{Line}, i.e., it is not possible to dynamically dispatch with regard to the type of the base objects being decorated with the subject functionality.

With the solution in Fig. 1 and 2 it is also pretty awkward to associate state with the individual abstractions in the definition of the aspect. For example, the observers of all subjects are stored in a global hash map \texttt{perSubjectObservers}. Besides the dangers of such a global bottleneck, the access and management of state becomes pretty clumsy. The example in Fig. 1 is relatively simple because state is associated with only one of the abstractions (\texttt{Subject}) and this state consists of only one “field”. However, the general case is that multiple abstractions in the module structure of the aspect may declare multiple fields. A simple example would be an implementation where observers maintain a history of the observed state change, e.g., when they need to react on change bundles rather than on individual changes. If we consider the case that all roles need many different fields then the code might very easily become a mess, if all these fields are hosted by the outer aspect.

The problem with modeling state becomes even worse, once we consider the case of role inheritance, e.g., \texttt{SpecialSubject} inheriting from \texttt{Subject}. In this case, we would end up simulating shared data fields manually. This problem with modeling state applies to the aspect binding as well. There we might also want to associate state with the objects that are mapped to the aspect roles, e.g., in order to cache computed values.

Summarizing the problems so far, what we would like to have is a nested class structure of aspect implementation and aspect binding within which we can assign methods and state to every aspect role in isolation.

**Lacking support for sophisticated mapping**

The second kind of problem with the solution in Fig. 1 and 2 is that the mapping from aspect abstractions to base classes by means of the \texttt{declare parents} construct works only when each aspect abstraction has a corresponding base class to which it is mapped directly. However, this is not always the case. Consider e.g., a scenario in which there is no class \texttt{Line} and every \texttt{Point} object has a collection of neighbor points. If we want to map this data structure to a graph aspect defined in terms of \texttt{Node} and \texttt{Edge} abstractions, then an edge would be represented by two adjacent points, but there is no abstraction in the base application to which we can map the \texttt{Edge} abstraction. The latter is only implicitly and indirectly represented by the collections of adjacent points.

**Lacking support for reusable aspect bindings**

Third, every aspect binding is coupled to one particular implementation. For example, the \texttt{ColorObserver} binding in Fig. 2 is hardwired to the observer pattern implementation in Fig. 1, although the binding itself is not dependent on the implementation details of the observer pattern. The observer pattern is not a very good example to illustrate the usefulness of a binding that can be used with many different implementations; a better example is that of an aspect binding that maps an arbitrary data structure, e.g., the classes of an abstract syntax tree, to a general tree representation. Many different implementations of a tree make sense in conjunction with such a binding, e.g., one that displays trees on the screen or one that performs algorithms on trees. That is, one might want to be able to write some functionality that is parameterized with a particular binding type, but is polymorphic with respect to the implementation. This is, however, not possible, if the binding is coupled to the implementation.

**Lacking support for asp{u}ctual polymorphism**

The fourth deficiency concerns aspect deployment. We say that the \texttt{ColorObserver} aspect in Fig. 2 is statically deployed. By this we mean that once compiled together with the package containing the figure classes, the changes in the particular points in the execution of point and line objects implied by \texttt{ColorObserver} aspect are effective. Which is to say that it is not possible to determine at runtime, whether to apply the aspect at all, or which implementation of the aspect to apply, e.g., a \texttt{LinkedList} version, or one with asynchronous notifications. We say that \texttt{aspectual polymorphism} is missing, in the sense that the code is not polymorphic with respect to the types and implementations of the aspects affecting it after compilation.

3. THE CAESAR MODEL

A core feature of CAESAR is the notion of an \texttt{aspect collaboration interface (ACI for short)} – an interface definition for
3.1 Aspect Collaboration Interfaces

An ACI consists, in general, of several mutually recursive nested ACIs - one for each abstraction in the modular structure of the aspect. The ACI ObserverProtocol in Fig. 3, for example, has two nested ACIs, Subject and Observer, that are mutually recursive in that the name of one type is used to define the other one and vice versa. A simple ACI that does not contain other nested ACIs, e.g., Subject, is a special kind of interface that lays down a bidirectional communication protocol between any possible implementation and binding of the corresponding abstraction. It does so by distinguishing between two part-interfaces: the provided and the expected facets of the abstraction, consisting of methods declared with the modifiers provided and expected, respectively. Hence, we can redefine an ACI as consisting of expected and provided declarations for the aspect as a whole as well as a set of mutually recursive nested ACIs - one for each abstraction in the modular structure of the aspect.

The provided facet of an aspect lays down what the aspect provides to any context in which it is applied. The observer ACI in Fig. 3 specifies that any implementation of ObserverProtocol must provide an implementation of the three provided methods of Subject. On the other side, the expected facet of an aspect makes explicit what the aspect expects from the context in which it will be applied, in order to be able to supply what the provided facet promises. Hence, the expected facet declares methods whose implementation is binding specific.

Consider for instance, the part of the observer protocol concerned with communicating relevant state from the subject to observers, when a change is notified. What part of subject’s state is relevant, and how this state should be extracted for being passed to observers is highly dependent on what classes play the the subject and observer roles in a particular context. Furthermore, the operation to be called on the observer as part of the notification is also binding specific. This is why notify() and getState() are declared with the modifier expected in Fig. 3.

An ACI’s provided and expected facets are implemented in different modules, called aspect implementations and aspect bindings respectively. However, all implementations and bindings of the same ACI are indirectly connected to each other, since they implement two facets of the same whole. The common ACI serves as a medium for bidirectional communication between them: Any module that implements one of the facets can freely use declarations in the other facet. This loose coupling is the key to independent reuse of implementations and bindings.

3.2 Aspect Implementations

An aspect implementation must implement all methods in the provided facet of the corresponding ACI, i.e., all aspect level provided methods, as well as provided facets of all nested ACIs. Fig. 4 shows a simple implementation of the ObserverProtocol ACI. Similarly, we could write another implementation of ObserverProtocol, say, a class AsyncObserverProtocolImpl that implements ObserverProtocol and realizes a notification strategy with asynchronous updates.

As illustrated in Fig. 4, an aspect implementation is a class that declares itself with an implements clause. Provided facets of the nested ACIs are implemented in nested classes which have the same names as their respective nested ACIs (see e.g., ObserverProtocolImpl.Subject in Fig. 4).

The implementation of provided methods can call expected methods of the same or of other abstractions of the same aspect. For example, Subject.changed() calls notify(), which is declared in the expected facet of ObserverProtocol. Observer. Nested implementation classes are free to define additional state and behavior (as, e.g., the observers field in Subject). Since ObserverProtocol.Observer has no provided methods, there is no Observer class in Fig. 4, but we could have added additional state and behavior with Observer, if necessary.

3.3 Aspect Bindings

An aspect binding implements all expected methods in the aspect’s CI and in its nested interfaces. Fig. 5 shows a binding of ObserverProtocol which maps the subject role to Point and Line and the observer role to Screen. The class ColorObserver declares itself as a binding of ObserverProtocol by means of a binds clause.
class ColorObserver binds ObserverProtocol {
    class PointSubject binds Subject wraps Point {
        String getState() {
            return "Point colored " + wrappee.getColor();
        }
    }
    class LineSubject binds Subject wraps Line {
        String getState() {
            return "Line colored " + wrappee.getColor();
        }
    }
    class ScreenObserver binds Observer wraps Screen {
        void notify(Subject s) {
            wrappee.display("Color changed: "+s.getState());
        }
    }
    after(Point p): (call(void p.setColor(Color)))
        { PointSubject(p).changed(); }
    after(Line l): (call(void l.setColor(Color)))
        { LineSubject(l).changed(); }
}

Figure 5: Sample binding of observer protocol

For each nested ACI of ObserverProtocol, i.e., Subject and Observer, there might be zero, one, or more nested bindings inside ColorObserver. The latter are also declared with a binds clause and must implement all expected methods in the corresponding interface. The relation between nested types in an ACI and their binding classes is not established by name identity, since there might be more than one binding for the same abstraction within the same binding class, as in Fig. 5.

Aspect binding is almost pure OO: A binding class refers to one or more base objects and uses their interface for implementing the expected facet of the aspect abstraction. The aspect binding in Fig. 5 uses only three non-OO features: (a) the wrap clause and the wrappee keyword, (b) wrapper recycling and (c) pointcuts/advises. Features (b) and (c) will be explained in Sec. 3.4, and 3.5. The wraps clause and the keyword wrappee are syntactic sugars for the common case, when each aspect abstraction is mapped to exactly one base class. For example,

class PointSubject binds Subject wraps Point {...}

is syntactic sugar for
class PointSubject binds Subject {
    Point wrappee;
    PointSubject(Point wrappee) { this.wrappee = wrappee; }
...}

In general, a wrapper class may have an arbitrary number of “wrappees” that can be initialized or computed in the constructor. Due to bindings being almost pure OO in CAESAR, the programmer is able to encode more complicated cases, where the relation to application objects has to be computed or is represented by multiple application objects (see [11] for more details).

3.4 Wrapper Instantiation

A subtle issue when using wrappers is how to avoid that multiple wrappers are created for the same base object (called wrapper identity hell [11]). Our solution is a mechanism called wrapper recycling. Syntactically, wrapper recycling refers to the fact that, instead of creating an instance of a wrapper W with a standard new W(constructorargs) constructor call, a wrapper is retrieved with the construct outerClassInstance.W(constructorargs). For illustration consider the expressions PointSubject(p) and LineSubject(l)3 in the after-advices in Fig. 5. We use the usual Java scoping rules, i.e., PointSubject(p) is just an abbreviation for this.PointSubject(p).

The semantics of wrapper recycling is that it guarantees a unique wrapper for every (set of) wrappees in the context of an outerClassInstance. The call to the wrapper recycling operation PointSubject(p) is equivalent to the corresponding constructor call only if a wrapper for p does not already exist. That is, two subsequent wrapper retrievals for a point yield the same PointSubject instance - the identity and state of the wrapper are preserved. For more details on wrapper recycling semantics we refer to [11].

Another interesting feature of CAESAR is its notion of most specific wrappers: A mechanism that determines the most specific wrapper for an object based on the object’s runtime type, when multiple nested binding classes with the same name are available. Consider, e.g., MovableFigures in Fig. 6, which contains three nested classes named MovableFigure. These classes have different constructors, though (recall that the wraps clause is just syntactic sugar for a corresponding constructor). On a constructor- or wrapper recycling call, the dynamic type of the argument determines the actual nested binding to instantiate/recycle. For example, if Test.move(Figure) in Fig. 6 is called with a Point as the actual parameter f, the wrapper recycling call mv.MovableFigure(f).moveBy(5,7) returns an instance of the MovableFigure implementation that wraps Point.

The mechanism of most specific wrapper is very similar to multiple dispatch in languages such as CLOS, Cecil [3], or MultiJava [4]. More precisely, if one thinks of the constructors of nested classes as factory methods of the enclosing instance, then our mechanism is an application of multiple dispatch at these factory methods.

3 Recall that the clauses wraps Point and wraps Line imply corresponding constructors.
3.5 Pointcuts and Advices

As illustrated in Fig. 5, CAESAR also have advices and pointcuts, which while being similar to AspectJ, differ from it in two points. The first difference concerns the decoration of executing (target) objects at a join point with aspect types. This decoration is implicit in AspectJ. For illustration, consider the pointcut `subjectChange` in Fig. 2: The base object, `s`, brought into the scope of `ColorObserver` by the join point `target`, whose type is either `Line` or `Point`, is automatically seen as being of type `Subject` within `ColorObserver` (see the parameter type of the pointcut).

On the contrary, the conversion is explicit in CAEASR, via wrapper recycling calls. In Fig. 5, we avoided type conversions in a pointcut, in order to avoid mingling the discussion on wrapper recycling with that on pointcuts and advices. For this reason, we defined different pointcuts for `Point` and `Line`. A shorter variant of the same binding, where we use conversions in the pointcuts, in given in Fig. 7. Note the explicit calls to wrapper recycling operators within the `with` clauses in Fig. 7; they allow us to decorate basis objects with different aspect facets in each “case” of the pointcut. We prefer the explicit variant because it increases programmer’s expressiveness: H/she can choose among several constructors of the binding classes, if more than one is available (see [11] for more details).

The second and more important difference between CAEASR and AspectJ pointcuts and advices is at the semantic level. Compiling a binding class that contains advice definitions does not have any effect on the base application’s semantics. This is because an aspect (its implementation and binding) must be explicitly deployed in Caesar.

On the contrary, the conversion is explicit in Caesar, via wrapper recycling calls. In Fig. 5, we avoided type conversions in a pointcut, in order to avoid mingling the discussion on wrapper recycling with that on pointcuts and advices. For this reason, we defined different pointcuts for `Point` and `Line`. A shorter variant of the same binding, where we use conversions in the pointcuts, in given in Fig. 7. Note the explicit calls to wrapper recycling operators within the `with` clauses in Fig. 7; they allow us to decorate basis objects with different aspect facets in each “case” of the pointcut. We prefer the explicit variant because it increases programmer’s expressiveness: H/she can choose among several constructors of the binding classes, if more than one is available (see [11] for more details).

The second and more important difference between CAEASR and AspectJ pointcuts and advices is at the semantic level. Compiling a binding class that contains advice definitions does not have any effect on the base application’s semantics. This is because an aspect (its implementation and binding) must be explicitly deployed in Caesar. Only the advice definitions of explicitly deployed aspects are executed, as elaborated in the following.

3.6 Weavelets and Deployment

In order to gain a complete realization of an aspect type, an implementation-binding pair needs to be composed into a new unit called a weavelet. An example of a weavelet is the class `CO` in Fig. 8, which represents a complete realization of the `ObserverProtocol` interface that combines the implementation `ObserverProtocolImpl` with the binding `ColorObserver`, denoted by the declaration after the `extends` clause.

A weavelet is a new class within which the respective implementations of the `expected` and `provided` methods from the binding and implementation parts are composed. The composition takes place recursively for the nested classes: All nested classes with a `binds` declaration are combined with the corresponding implementation from the implementation class.

A weavelet has to be `deployed` in order to activate its pointcuts and advices. A weavelet deployment is syntactically denoted by the modifier `deploy` and comprises basically two steps: (a) create an instance of the weavelet at hand and (b) call the deploy operation on it. One can choose between `static` (load-time) and `dynamic` deployment.

**Static deployment**

Static deployment is expressed by the `deploy` keyword be a modifier of a `final static` field declaration. Semantically, it means that the advices and pointcuts in the instance that has been assigned to the field become active. For example, `co` is deployed when `Test` is loaded in the following code extract:

```java
class Test ... {
    deploy public static final CO co = new CO();
    ...
}
```

The object assigned to `co` could also be computed in a `static` method; hence, the weavelet that is actually deployed might also be a subtype of `CO`, thereby enabling static aspectual polymorphism. The `deploy` keyword can also be used as a class modifier. This variant should be regarded syntactic sugar in the sense that

```java
deploy class CO ... { ... }
```

is equivalent to declaring a deployed field named `THIS` as in:

```java
class CO ... {
    deploy public static final CO THIS = new CO();
    ...
}
```

Fig. 9 shows the declaration of a statically deployed color observer protocol together with sample code which shows how the deployed weavelet instance can be accessed (`.register()`). Since `CO.THIS` is deployed, the pointcuts of the observer protocol are active, i.e., color changes in points and lines will be propagated to `CO.THIS`.

Using `deploy` as a class modifier is appropriate if we need only one instance of the aspect and if aspectual polymorphism is not required. By means of `deploy` as a field modifier we can create and deploy multiple instances of the same weavelet and select from different weavelets using aspectual polymorphism. Having two instances of, say, the `CO` weavelet in the observer example would mean that every `Point` and `Line` would have two independent facets as subject with independent lists of observers. An example that
class Logging {
  after(): (call(void Point.setX(int)) ||
    call(void Point.setY(int))) {
    System.out.println("Coordinates changed");
  }
}
class VerboseLogging extends Logging {
  after(): (call(void Point.setColor(Color)) {
    System.out.println("Color changed");
  })
class Main {
  public static void main(String args[]) {
    Logging l = null;
    Point p[] = createSamplePoints();
    if (args[0].equals("-log")) {
      l = new Logging();
    } else if (args[0].equals("-verbose")) {
      l = new VerboseLogging();
    }
    deploy(l) {
      modify(p);
    }
  }
  public static void modify(Point p[]) {
    p[2].setColor(Color.RED);
    p[3].setX(5);
  }
}

Figure 10: Polymorphic aspect deployment

makes more sense is the association of color to elements of a
data structure which can be seen as nodes of a graph. Multi-
ple independent instances of the corresponding weavelet
would represent multiple independent colorings of the graph.
Other examples can be derived from role modeling, where
frequently one object has to play the same role twice, for
example, a person is employee in two independent com-
panies. Static aspectual polymorphism is useful if we want
to select a particular weavelet based on conditions that are
known at load-time. For example, based on the number of
processors or the multi-threading support, one might either
choose a usual observer pattern implementation or one with
asynchronous updates.

Dynamic Deployment

Dynamic deployment is denoted by the keyword deploy used
as a block statement. The rationale behind dynamic deploy-
ment is that frequently we cannot determine which variant
of an aspect should be applied (or whether we need the as-
pect at all) until runtime. Consider e.g., a program with dif-
f erent logging options, i.e., without logging, with standard
logging, and with "verbose" logging. In CAESAR, this can be
implemented as in Fig. 10: We have two different logging
aspects related by inheritance, Logging and VerboseLog-
ing), and we choose one of them at runtime, depending on
the command line arguments with which the program has
been started.

The interesting point is the deploy block statement in
Main.main, which means that the advices defined in the an-
notated aspect instance l become active in the control flow
of the deploy block, in this case, during the execution of
modify(f). In particular, other independent threads that

Figure 11: Aspect deployment aspects

eexecute the same code are not be affected by the deploy
clause. Please note that the advices and pointcuts that will
be activated in the deploy block are not statically known; l
is only known by its upper bound Logging (l could have also
been passed as a parameter). In other words, the advices
are late bound, similarly to late method binding, hence our
term aspectual polymorphism. If l is null the deploy clause
has no effects at all.

The usefulness of dynamic deployment becomes clear if
we consider a "simulation" of this functionality by means
of static deployment. With static deployment, we would
have to encode the different variants by conditional logic in
the aspect code, 5. The structure of the aspect would get
awkward because all variants of the aspect are tangled inside
a single aspect module. In a way, this is similar to simulating
late binding in a non-OO language, hence we see dynamic
aspectual polymorphism as an imperative consequence of
integrating aspects into the OO concept world. Also, such
programs would be very fragile with respect to concurrent
programs and additional synchronization measures would be
required.

An interesting question is whether the aspect deployment
code should also be separated from the rest of the code. If
desired this can easily be done with another aspect whose
responsibility is the deployment of the logging aspect, as
indicated in Fig. 11. In this figure, the aspect LoggingDe-
ployment (which is itself deployed statically) computes and
deploys an appropriate logging aspect by means of an around
advise, i.e., the proceed() call is executed in the context of
the logging aspect.

3.7 Virtual Classes and Static Typing

In CAESAR, all nested interfaces of a CI and all classes that
implement or bind such interfaces, are virtual types/classes,
as in the family polymorphism approach [5]. Similar to fields
and methods, virtual types also become properties of objects
of the class in which they are defined. Hence, their deno-
tation can only be dynamically determined in the context
of an instance of the enclosing class. The rationale behind
using family polymorphism lies in its power with respect to
reuse and polymorphism at the level of multiple related

5Our example also uses conditional logic in Main.main. However, we select the logging version once and never have
to do any checks again (a factory object could have been used, as well) whereas without dynamic deployment the
check would be redone at every joinpoint.
public class LazyColorObserver extends ColorObserver {

    override class ScreenObserver {
        int count = 0;
        void notify(Subject s) {
            count++;
            if (count >= 10) {
                super.notify(s);
                count = 0;
            }
        }
    }

    Figure 12: Lazy color observer

    97

    97

abstractions.

    If we want to have a variant of a binding, weavelet, or
    CI, we can refine the respective entity by creating an ex-
    tension within which the nested virtual types/classes can
    be overridden. LazyColorObserver in Fig. 12 refines the
    behavior of ColorObserver in Fig. 5 by using virtual class
    overriding (declared with the keyword override) - a lazy
    ScreenObserver reacts only after being notified ten times
    about a change. The important observation to make here is
    that even if the definition of PointSubject and LineSubject
    are inherited unchanged, references to Observer within their
    respective implementations will automatically be bound to
    LazyColorObserver.ScreenObserver during the execution
    of any method on an instance of LazyColorObserver.

    However, this flexibility is not paid for with loss of static
typing: An improvement of the type system proposed in [5]
preserves the ability to detect type errors at compile time.
The integration of virtual classes [9] and family polymor-
phism [5] with collaboration interfaces and their implementa-
tion and binding units has already been described in [11].
Hence, for more details on reuse and typing issues we refer
to [11] and [5].

4. EVALUATION

    This section discusses how CAESAR copes with the prob-
lems outlined in Sec. 2. In addition, we will elaborate on
how CAESAR’s explicit aspect instantiation and deployment
relate to AspectJ-like languages, where aspects are only im-
plcitly created and which do not have a notion of aspect
deployment.

Problems Revisited

    Recall that we identified the following problems in Sec. 2:
(1) lacking support for multi-abstraction aspects, (2) lacking
support for sophisticated mapping of aspect abstractions to
base classes, (3) lacking support for reuse of aspect bindings,
(4) lacking support for aspectual polymorphism. In the fol-
lowing we will explain how each of these problems is solved
in CAESAR.

Ad 1:. As was shown in the code in Fig. 3, 4, and 5, each ab-
straction in the vocabulary of the world as it is decomposed
from the point of view of an aspect, is defined in its own
full-fledged module with a well-defined interface. Methods
in the interface of one abstraction can be called by methods
of other abstractions within the same aspect, or from the
outside. Consider e.g., the call of Subject.notify(...) in
the implementation of ObserverProtocolImpl in Fig. 4, or
the invocation of CO.THIS.addObserver(...) in Fig. 9.

    Due to this finer-grained modularization of the aspect it-
self, the runtime system is able to dispatch methods not only
based on the instance of the aspect, but also based on the
particular abstraction in execution. Consider, for example,
the getState() method in the definition of Subject, which
was implemented differently for point-subjects and for line-
subjects, while being uniformly used in the update logic (cf.
Fig. 5). As was pointed out in Sec. 2, the same polymor-
phism would not be possible, if there were only aspect-level
methods. Furthermore, due to the incorporation of virtual
classes, it is easy to encode different variants of a multi-
abstraction aspect, as exemplified in Fig. 12.

    Let us now consider the issue of defining state for the
individual abstractions pertaining to an aspect. As it was
shown by examples in the previous section, each abstrac-
tion in the modular structure can declare its own state, e.g.,
observers in Subject. Hence, there is no need for defin-
ing data structures that "globally" maintain aspect-related
state of all base objects in a single place, as e.g., perSub-
jectObservers in Fig. 1. Similarly, state can be added to
the abstractions at the binding side, such as e.g., the count
field in Fig. 12.

Ad 2:. In our model bindings are Java classes with some ad-
ditional features. As such, the definition of mappings from
aspect abstractions to the classes of a base application can
make use of the full expressiveness of an general purpose OO
language. There is nothing to prevent a CAESAR program-
er in coding any mapping no matter how sophisticated.
A more detailed discussion on this issue supported also by
better examples can be found in [11].

Ad 3:. Different weavelets can combine an aspect binding
with different aspect implementations. On the other hand,
different weavelets can combine (and reuse) a particular as-
pect implementation with multiple different bindings. For
example, we can combine the observer protocol binding to
MyActionListener with the LinkedList or the
AsyncronousUpdate observer implementation, and on the
other hand combine the same observer implementation, say
AsyncronousUpdate, with multiple different bindings, e.g.,
to JButton/MyActionListener and ListModel/ListItem.
As a consequence, one can define functionality that is polymor-
phic with respect to (a) aspect implementations by being
written to a certain aspect binding type, (b) aspect bind-
ings by being written to a certain aspect implementation
type, or (c) both of them, by being written to an ACI.

Ad 4:. As already discussed in Sec. 3.6, our approach
does support aspectual polymorphism. For example, the
modify(Point p[]) method in Fig. 10 is polymorphic with
respect to aspects that might be defined in the future. It is
even possible to run the same method concurrently within
two different threads with and without the logging aspect.

Explicit vs. Implicit Aspect Instantiation/Deployment

    The question we pose here is: How does our notion of ex-
plcit aspect instantiation and deployment relate to AspectJ-
like languages, within which aspects are only implicitly cre-
ated and which do not have a notion of aspect deploy-
ment? In AspectJ, aspect instantiation can be controlled by
means of the aspect modifiers isSingleton (this is the de-
fault), perThis/perTarget, and percflov/percflowbelow.
In Caeser, these aspect instantiation strategies turn out to be special cases or “patterns” of the more general model in Caeser.

Tab. 1 describes how the AspectJ instantiation strategies can be simulated in Caeser. The isSingleton case is obvious. The perThis modifier can be simulated by creating a wrapper class and using wrapper recycling in order to refer to the state that is associated with each point. Simulating perTarget is identical to perThis, except that we would have to exchange this(p) by target(p). More interesting is AspectJ’s perflow modifier, which means that an instance of the aspect is created for each flow of control of the join points picked out by the annotated pointcut. The semantics of perflow can be simulated by using a deployment aspect ADegi that uses dynamic deployment at the respective starts of control flow.

What do we gain if all the cases in Tab. 1 can already be handled very well by AspectJ? To answer this question recall that AspectJ instantiation strategies are just special cases of a more general model in Caeser. This has two implications. First, we do not need special new keywords to express the semantics of AspectJ instantiation, thereby rendering the conceptual model more slender. Second and more importantly, our model allows us to express instantiation and deployment semantics that cannot easily be expressed in AspectJ.

When using AspectJ’s perThis of perTarget modifiers, state can be only associated with objects that are caller or callee, respectively, in a pointcut. In Caeser, state can be associated with arbitrary objects and arbitrary relations between objects. For example, we could associate state with every pair of this and target, or with any argument of a method call. In the perflow case we can either simulate the AspectJ semantics but we could also do something more sophisticated, e.g., deploy an instance of an optimization aspect only if the number of calls to the method to be optimized is executed more than a certain threshold.

5. RELATED WORK

Open classes: An open class is a class to which new fields or methods can be added without editing the class directly. For example, in MultiJava [4] additional methods can be attached to a class. In AspectJ, methods as well as fields can be added to a class by means of introductions. As already discussed in Sec. 2, open classes are in contrast to the concept of independent extensibility [14], an essential prerequisite for reusable and extensible software. On contrary, Caeser offers an alternative to open classes that is even more powerful and does not violate independent extensibility.

Adaptive Plug and Play Components (APPs) [10] and their aspect-oriented variant of Aspectual Components [8] are related to our work in that both approaches support the definition of multi-abstraction components/aspects and have a vague definition of required and provided interfaces. However, the latter feature was not well integrated with the type system. Recognizing this deficiency, the successor model of Pluggable Composite Adapters (PCAs) [12] even dropped this notion and reduced the declaration of the expected interface to a set of standard abstract methods. With the notion of collaboration interfaces, Caeser represents a qualitative improvement over all three models, as far as support for multi-abstraction aspects is concerned. Due to the lack of a CI notion, connectors and adapters in APPC, Aspectual Components, and PCA models are bound to a fixed implementation of an aspect and cannot be reused. In addition, [10] and [8] rely on a dedicated mapping sublanguage that is less powerful than our object-oriented wrappers with wrapper recycling. Finally, the lack of the notion of virtual types is another drawback of these approaches as compared to the work presented here.

Delegation layers [13] are an approach to decompose a collaboration into layers and compose these layers dynamically at runtime. We plan to integrate delegation layers with Caeser in order to organize aspect implementations and bindings in layers and compose them dynamically.

Caeser is also related to Hyper/J and its notion of multidimensional separation of concerns (MDSOC) [15]. Our aspect bindings, which serve as a translator from one domain to another domain, allow to view and use a system from many different perspectives. This is similar to the MDSOC idea of having multiple concern dimensions such that the program can be projected on each concern hyper plane. Apart from that, Caeser is very different from Hyper/J. In Hyper/J, one can define an independent component in a hyperslice. Hyperslices are independent of their context of use by the feature of being declaratively complete, i.e., they declare as abstract method everything that they need, but cannot implement themselves. A hyperslice is integrated into an existing application by means of composition rules specified in a hypermodule. As the result, new code is generated by mixing the hyperslice code into the existing code. Similar to PCAs, Hyper/J [15] also lacks the notion of collaboration interfaces and the reuse of bindings related to it: Either the modules to be composed are not independent due to the usage of the “merge-by-name” composition strategy or the modules are independent but then the non-reusable composition specification gets very complex. Similar to APPC and Aspectual Component models, Hyper/J’s approach is class-based: it is not possible to add the functionality defined in a hyperslice to individual objects. Furthermore, Hyper/J’s sublanguage for mapping specifications from different hyperslices is fairly complex and not well integrated into the common OO framework.

Lasagne [16] is a runtime architecture that features aspect-oriented concepts. An aspect is implemented as a layer of wrappers. Aspects can be composed at run-time, enabling dynamic customization of systems, and context-sensitive selection of aspects is realized, enabling client-specific customization of systems. Although Lasagne is an architectural approach focusing on middleware (instead of a general purpose language extension as Caeser), it has some similarity with Caeser. In particular, Lasagne also features extensions that are created and deployed at runtime, and it also provides means to restrict the visibility of an extension to a particular scope (as our deploy block statement).

In [2] an extension of the composition filter model [1] geared more towards aspect-oriented programming is discussed. With composition filters, it is possible to define various filters for incoming and outgoing messages of an object. By means of superimposition [2], it is possible to apply these filters to objects that are specified via a join point declaration similar to AspectJ pointcuts. Composition filters have no dedicated means to separate aspect implementation and binding, and there is notion of deployment or aspectual polymorphism. In comparison with Caeser, where almost
everything is specified as usual OO code, composition filters are more declarative. On one hand, this makes it easier to express kinds of concerns that are easily expressible with the declarative sublanguage, but on the other hand it restricts applicability to arbitrary kinds of concerns.

6. SUMMARY

In this paper, we argued that join point interception (JPI), that is, intercepting and eventually modifying the execution of running code at certain points, alone does not suffice for a modular structuring of aspects, resulting in tangled aspect code. We discussed several problems resulting from the lack of an appropriate higher-level module construct on top of join points and advices. We proposed Caesar, a model for aspect-oriented programming with a higher-level module concept on top of JPI, which enables reuse and componentization of aspects, allows us to use aspects polymorphically, and introduces a novel concept for dynamic aspect deployment. Caesar is based on the notion of an aspect collaboration interface (ACI) presented in [11]. In this paper we show that ACIs and the related notions of separated ACI implementations and ACI bindings, once properly adopted to the needs of aspect-orientation, can also be applied to support a more modular structuring of aspect code and better aspect reuse.

7. REFERENCES