Variability Management with Feature-Oriented Programming and Aspects

Mira Mezini, Klaus Ostermann
Darmstadt University of Technology
D-64283 Darmstadt, Germany
{mezini,ostermann}@informatik.tu-darmstadt.de

ABSTRACT

This paper presents an analysis of feature-oriented and aspect-oriented modularization approaches with respect to variability management as needed in the context of system families. This analysis serves two purposes. On the one hand, our analysis of the weaknesses of feature-oriented approaches (FOAs for short) emphasizes the importance of crosscutting modularity as supported by the aspect-oriented concepts of pointcut and advice. On the other hand, by pointing out some of AspectJ's weaknesses and by demonstrating how Caesar, a language which combines concepts from both AspectJ and FOAs, is more effective in this context, we also demonstrate the power of appropriate support for layer modules.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*; D.2.11 [**Software**]: Software Engineering—*Software Architectures*

General Terms

Design, Languages

Keywords

Variability Management, Product Lines, Aspect-Oriented, Feature-Oriented

1. INTRODUCTION

Classes as the traditional units of organization of object-oriented software have proved to be insufficient to capture entire *features* of the software in a modular way. As a result, the last decade has seen quite a number of approaches that concentrate on a more appropriate representation of features in the source code. In this paper, we analyze two families of such approaches, with the goal of illustrating their strengths and weaknesses and emphasizing that a best-of combination of their concepts is needed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT'04/FSE-12, Oct. 31–Nov. 6, 2004, Newport Beach, CA, USA.
Copyright 2004 ACM 1-58113-855-5/04/0010 ...$5.00.

On the one hand, there is a class of approaches that concentrate on encapsulating features as increments over an existing base program, together with a mechanism for combining different features on demand. Approaches in this class include GenVoca [2], mixin layers [28], delegation layers [25], and AHEAD [3] and will be referred to as feature-oriented approaches in the reminder of this paper (FOAs for short). On the other hand, a number of aspect-oriented approaches have been proposed that deal with the localization and modularization of crosscutting concerns [12, 30, 13, 4]. Due to their diversity, we will not discuss aspect-oriented approaches as a homogeneous family. We will only consider AspectJ [11], the most mature approach today, and Caesar [22, 23], a language which combines concepts from both AspectJ and FOAs. The discussion of other aspect-oriented approaches is postponed until we present related work at the end of the paper.

FOAs are superior to framework technology since they introduce a *layer* module with two distinctive capabilities. First, a layer encapsulates multiple abstractions, respectively deltas of them, which together pertain to the definition of a feature, into a single modular unit. A layer nicely localizes the definition of a feature that would otherwise be scattered around the definition of several classes into a single code unit. Second, a layer is a mixin-like module, i.e., it abstracts over the concrete variant of the base definition it applies to. Individual abstractions encapsulated within a layer are defined as mixins to their respective base abstractions and the layer plays a similar role to them as classes play to their method definitions. This is the key to FOAs' support for variability management: Variants of a base behavior can be composed in a plug-and-play fashion.

The analysis in this paper shows, however, that FOAs are weak with regard to capturing crosscutting features in a modular way. They require the definition of a feature delta to be "purely hierarchical" to the base definition[1] - the delta is a refinement of the base. With FOAs, we are forced to express features in terms of the basic hierarchical structure by mapping the abstractions of a given feature to existing base classes via refinements. However, there might be features that rather crosscut the modular structure of the base. There are two forms of such crosscutting, static and dynamic.

Static crosscutting means that several feature abstractions may map to the same base abstraction, or vice versa, that there might be feature abstractions that do not have any

[1]We will also use the term "basic modular structure" to refer to the modular structure chosen for the base.

direct correspondence to abstractions in the basic modular structure. We will discuss that it is hard to encode such features as refinements in a modular manner.

Dynamic crosscutting refers to the specification of how the feature interacts with the dynamic control flow of the base program. The "join points" that can be expressed with FOAs are limited to individual method calls: A feature can join the execution by overriding method calls. There are no means to specify general sets of related join points that may crosscut the given module structure. We will show that this damages the scalability of the divide-and-conquer technique underlying FOAs.

Due to the notions of join points and advice, as exemplified by AspectJ, the interaction of a feature with the base program can be expressed much more conveniently than with method overriding only as in FOAs. However, we will argue that there are also some important deficiencies of AspectJ when considered from the perspective of feature modeling, namely limited means to structure the aspect itself and little support for variants of a program.

We also discuss disadvantages that are shared by both FOAs and AspectJ. First, both features and aspects are encoded relative to a fixed base program; hence, it is hard to reuse these implementations in a different program. Second, there is only little support for dynamic configuration of software. In most FOAs (except [25]), once instantiated, the configuration of a product line does not change at runtime. On the other side, aspects in AspectJ are activated at compile time. This is not appropriate for features whose most specific variant to use cannot be determined before runtime.

Recognizing that neither FOAs nor AspectJ is fully satisfactory, we show how Caesar [22, 23] copes with the outlined problems. Caesar actually combines the best of both worlds: It is an advancement over both existing feature-oriented and aspect-oriented approaches in that it combines their relative strengths and also provides solutions to the common problems stated above.

To summarize, the contribution of this paper is an analysis of feature-oriented and aspect-oriented modularization approaches with respect to variability management as needed in the context of system families. This analysis serves two purposes. On the one hand, it is our goal to emphasize the importance of crosscutting modularity as supported by the aspect-oriented concepts of pointcut and advice. To this end, we hope to shed light on the applicability of AOP beyond traditional examples of logging, debugging, authorization control, and the like. On the other hand, by pointing out some of AspectJ's weaknesses and by demonstrating how Caesar, which combines concepts from both AspectJ and FOAs, is more effective in this context, we also demonstrate the power of appropriate support for layer modules.

Our analysis will be driven by the example of a software for requesting stock information, whose simplified class structure is shown in Fig. 1. The central abstraction in Fig. 1 is the class `StockInformationBroker` that implements two methods. When called, the method `collectInfo` receives a `StockInfoRequest`, looks up the information for all stocks in the request by using the `DBBroker` database and wraps all collected information into a `StockInfo` object, which it returns. In the remainder of the paper, we will use the abbreviation *SIB* to refer to the stock information broker application. We will discuss variability by means of a particular feature of this software, namely *pricing*: Clients of

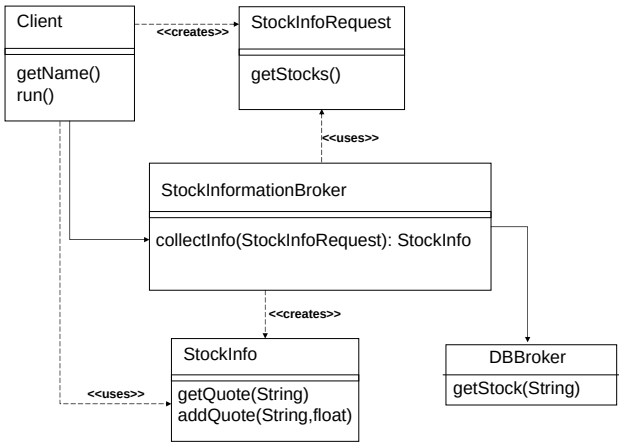

Figure 1: Stock information broker (SIB)

the SIB application are supposed to pay for using this service. We will discuss different requirements that may arise in the context of the pricing feature and discuss how they are addressed by both FOAs and aspects.

The remainder of this paper is structured as follows. Sec. 2 analyzes feature-oriented approaches. In Sec. 3, we sketch the alternative designs possible with AspectJ and analyze their strengths and weaknesses. In Sec. 4, we show how Caesar can be used to realize the scenario described before and how it addresses the problems of FOAs and AspectJ. Sec. 5 discusses related work. Sec. 6 summarizes the paper and outlines areas of future work.

2. VARIABILITY WITH FOA

With FOAs, a feature is encoded as a delta over an existing base structure. This delta is usually expressed in a subclass/mixin-like style. Using FOA in our example, we basically have two different options: (a) implement pricing functionality as a layer on top of SIB, or (b) implement SIB as a layer on top of pricing. Hence, either the pricing functionality has to be encoded in terms of the modular structure of the SIB, or vice versa. Since we assume that we already have a running version of the SIB application, we will discuss the first case. In the second case, the problems we identify are basically the same, hence, this choice does not affect the discussion.

Choosing the first approach for illustration means that we consider the SIB software as the base element (a *constant* in terms of [3]) and define the pricing feature as a delta on top of it, as shown in Fig. 2 and 3. The delta definition in Fig. 3, which uses the syntax of AHEAD [3], contains extensions of `Client`, `StockInfoRequest`, and `StockInformationBroker`[2]. The first two refinements map pricing abstractions to base types, while the refinement for `StockInformationBroker` is needed to integrate the pricing feature into the control flow of the base. The refinement for `StockInformationBroker` actually contains an error because the original signature of the overridden method `collectInfo` does not declare a parameter of type `Client`. We will later discuss this point further.

[2]In AHEAD [3], the modifier `refine` denotes extensions.

```
class Client {...}
class StockInformationBroker {...}
class StockInfo {...}
class DBBroker {...}
class StockInfoRequest {...}
```

Figure 2: Base layer

```
refines class Client {
  float balance;
  public float balance() { return balance; }
  void charge(StockInfoRequest r) { balance -= r.price(); }
}

refines class StockInfoRequest {
  float price() { return basicPrice() + calculateTax(); }
  float basicPrice() { return 5 + getStocks().length*0.2; }
  float calculateTax() { ... }
}

refines class StockInformationBroker {
  StockInfo collectInfo(Client c, StockInfoRequest r) {
    super.collectInfo(c, r);
    c.charge(r);
  }

}
```

Figure 3: Pricing delta

Class refinements are similar to subclasses (we can add fields like in `Client` or override methods like in `StockInformationBroker`) with the additional flexibility that different layers can be freely combined. For example, we could have another layer that adds a security feature to the SIB application and combine it with the pricing layer. Every combination of features can be made available in a separate namespace[3], hence it is possible to have multiple different configurations of a product line in the same application.

Now, let us consider our claim that "FOAs are purely hierarchical". In our example, pricing is defined in terms of abstractions dictated by the basic modular structure, such as `Client`, `StockInfoRequest`, etc., although it would be more naturally organized in terms of abstractions such as `Product` and `Customer`. For example, the implicit pricing abstraction `Product` is mapped to the base abstraction `StockInfoRequest` by encoding the product-specific properties as a refinement of `StockInfoRequest`; similarly, the implicit pricing abstraction `Customer` is mapped to `Client`.

The problem with this encoding is that hierarchical modularity encoded by refinement is not appropriate to modularizing features whose modular structures are not in a hierarchical relationship to each other. This is the case in our example: The modular structure of pricing is not a refinement of a modular structure of a SIB software. This makes the mapping difficult to express and maintain. There are two facets of the problem.

The first facet concerns the structural mapping between feature and base abstractions when there is static crosscutting between the two: There is no one-to-one relation between the respective abstractions, e.g., there is no class in the base to which we can map an abstraction in the feature as a subclass.

[3]This means that we would have classes like `base.Client`, `base.DBBroker`, ..., as well as `pricing.Client`, ..., `security.Client`, ..., `pricingsecurity.Client`, ...etc.

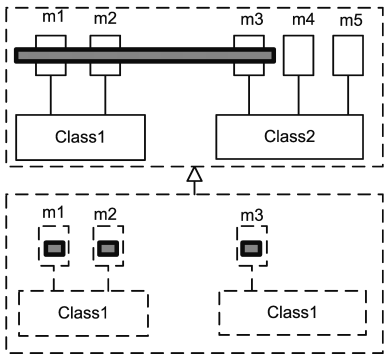

Figure 4: Scalability problem of refinement

It could be possible that the SIB software consults different databases that must be paid separately; hence, we would like to apply the `Customer` role to a *pair* consisting of a client and a database. There is no abstraction in the SIB layer to which we could map such a role, i.e., from which to inherit. One possibility to work around this problem is as follows. We could add a separate class that stores the current balance for a client/database pair. But now we need an additional structure to maintain the relation between client/database pairs and instances of this class; this adds more accidental complexity to the system.

The second facet concerns the integration of a feature into the control flow of the base, which in general involves dynamic crosscutting. We first consider the problem as it appears in our example; subsequently, we generalize. In Fig. 3, we shadowed the definition of `StockInformationBroker` to express the crosscutting structure of pricing. In our simplified world, pricing is triggered in a single point in the execution of SIB, namely after the call to `collectInfo`; in general, it is easily conceivable that triggering the execution of a feature has to be performed in several places, which requires the redefinition of the base layer in various places.

In fact, Fig. 3 contains an error which was introduced deliberately: The original version of `collectInfo` does not have a parameter of type `Client`. Such an argument is needed, though, in order to call `charge` on it. Some workaround is needed to get a reference to the `Client` instance calling `StockInformationBroker.collectInfo`. We could try to override all callers of `collectInfo` inside `Client` and elsewhere in order to get hold of this reference.

In general, FOAs do not scale when crosscutting is involved, because in the worst case the effort required to encode a dynamic crosscut grows linearly with the program size. Consider, e.g., a feature like logging that affects large parts of the system. All affected classes and methods would have to be shadowed in order to trigger the logging feature. With FOAs, we can refine a whole hierarchy of entities (layers, classes, methods). But, in order to encode a crosscutting feature, we have to shadow all parts of the hierarchy above the nodes involved in triggering the crosscutting feature at runtime. This is illustrated in Fig. 4. The bar affecting methods `m1`, `m2`, and `m3`, stands for a crosscut that needs to be refined in order to integrate a feature in the bottom layer. To override the respective definitions, however, a large part of the given hierarchy has to be "shadowed" in the bottom layer.

An encoding of the dynamic crosscutting by overriding individual methods has serious problems with respect to maintenance and evolution. Since the specification of the shadow is not modularized, a change becomes necessary in several places of the feature delta, whenever we add to the base a class with methods that, when invoked, should trigger the execution of the pricing functionality, or eventually when we modify the methods that in the current structure trigger an execution of the pricing feature.

Besides of being "purely hierarchical", a further problem of FOAs is lack of appropriate support for reuse. One can easily envisage other usage scenarios of pricing in other application families, or even within the same application family. Unfortunately, we cannot reuse the algorithm encoded in the implementation of the pricing functionality in Fig. 3. This is because the algorithm does actually not exist in its own, but is tangled with the issues of its particular deployment for SIB pricing. Again, the problem boils down to the lack of what could be called "crosscutting modularity" that would allow the definition of a generic "pricing feature" in terms of its own abstractions, which could then be superimposed over the basic modularity structure in different ways by means of some mechanism to express the crosscut between independent modular structures.

Finally, the last problem with FOAs is the lack of support for dynamic configuration. For illustration, consider that there are different pricing policies such as "flat rate", "volume-based", etc., that are implemented as separate layers. Every user of the system may have chosen a different pricing policy. This kind of dynamic configurability is not supported by the FOAs we are aware of.

3. VARIABILITY WITH ASPECTJ

In this section we consider how the same problem could be addressed in AspectJ [11]. We have two main options:

- We encode the complete pricing semantics in methods and state of the aspect using the pointcut and advice constructs only. Pricing is triggered by means of an advice that intercepts calls to `collectInfo`. This option is illustrated in Fig. 5.
- We add the pricing related methods and state to the module structure of the base program by means of inter-type declarations. This option is illustrated in Fig. 6.

An advantage of the AspectJ solutions is their support for expressing crosscutting by means of pointcuts and advice. In contrast to the FOA solution, no shadowing is necessary in order to trigger pricing in the base application. Also, it is not a problem that the `collectInfo` method does not have a `Client` parameter; we can use the `call` pointcut designator in order to retrieve the caller of the request. Even if the client would not call `collectInfo` directly but via some intermediate objects, we could still retrieve the source of the request by means of the `cflow` pointcut designator. With more advanced mechanisms such as wildcards, field getters/setters, `cflow`, etc., a pointcut definition also becomes more stable with respect to changes in the base structure than the corresponding set of overridden methods in FOAs. The use of pointcuts instead of shadowing parts of an inherited base structure avoids the scalability problem mentioned in the FOA discussion. The key point is that

```
aspect Pricing {
  HashMap clientBalance = new HashMap();

  void charge(Client c, StockInfoRequest r) {
    clientBalance.put(c,
      new Integer(
       ((Integer) clientBalance.get(c)).intValue()
          - price(r) )
      );
  }

  int balance(Client c) {
    return ((Integer) clientBalance.get(c)).intValue();
  }

  int price(StockInfoRequest r) {
    ... // analogous to charge-method
  }

  after( Client c, StockInfoRequest request ):
    (call (StockInfo collectInfo(StockInfoRequest))
    && this(c) && args(request)) {
     charge(c,request);
    }
}
```

Figure 5: Pricing with pointcuts and advice

```
aspect Pricing {

  private float Client.balance;

  float Client.balance() { return balance; }

  void Client.charge(StockInfoRequest r) {
    balance -= r.price();
  }

  float StockInfoRequest.price() { ... }

  after( Client c, StockInfoRequest request ):
    (call (StockInfo collectInfo(StockInfoRequest))
    && this(c) && args(request)) {
     c.charge(request);
    }
}
```

Figure 6: Pricing with inter-type declarations

with pointcuts we can abstract over details in the control flow that are irrelevant to the feature integration. Equivalent abstraction mechanisms are missing in FOAs.

Now, we discuss the weaknesses of AspectJ solutions. A major disadvantage of the first option is that the whole conceptual structure of the pricing functionality and its mapping to the SIB application is defined in a flat "global" method-space; furthermore, clumsy hashtable-like workarounds are needed to associate state with the pricing abstractions (see the `clientBalance` hashtable in Fig. 5). The second option in Fig. 6 is similar to the FOA solution in that pricing functionality is superimposed on the existing modular structure of the base application. Compared to the first option, this second option is less attractive with respect to the principle of *independent extensibility* [29], as argued in [23].

A problem of AspectJ-like aspects, in general, is their lack of support for variability. We can either compile the pricing aspect and have the pricing functionality in *all* instances of the SIB application, or not use it at all; in contrast to FOAs, it is not possible to have different variants of the SIB application in the same system. Adding a FOA layer is similar to creating a subclass: it does not change the

semantics of existing code, and instances of both the class and the subclass can co-exist in the same system.

The same semantics is not directly supported in AspectJ: An AspectJ aspect, when compiled, changes the semantics of the affected classes in an invasive way. A more fine-grained control is actually possible by the following pattern. One can define (empty) subclasses of the classes to be affected and apply aspects only to these subclasses; another aspect can then be defined that redirects the creation of the original classes. This pattern also enables some form of dynamic configuration: The aspect controlling the instantiation can decide on a per-object basis whether an aspect applies or not by instantiating the appropriate class.

However, there are two problems with this pattern. First, as any pattern and idiom, it requires some effort to implement it correctly and consistently; furthermore, its correct and consistent use cannot be verified. Second, it is not obvious how to combine different aspects that apply to the same classes because both aspects apply only to separate subclasses that would somehow need to be combined.

Another disadvantage which AspectJ shares with FOAs is the lack of support for reuse - aspects are defined in terms of the base code. To this end, AspectJ's mechanism for specifying effects on the inheritance structure of a base application (the `declare parents` construct) ought to be mentioned. Following the coding pattern for AspectJ used in [8] to define a reusable publish-subscriber protocol, one might be able to define a reusable pricing implementation in an abstract aspect. The latter could then be bound to a concrete base modular structure by means of `declare parent` statements in a concrete sub-aspect. In [23], we show that this implementation pattern suffers from basically the same problems with regard to the structure of the aspect as the first option discussed above.

4. VARIABILITY WITH CAESAR

In this section, we analyze how features are modeled in the programming language Caesar [23].

4.1 Crosscutting Layers

The Caesar implementation[4] of a simple pricing policy for SIB is shown in Fig. 7. In Caesar, the concept of a class is generalized to serve also as a layer module capable of encapsulating the definition of multiple interdependent types (nested classes) contributing to a feature, in such a way that these types constitute a family [7].

The layer `SimpleSIBPricing` in Fig. 7 defines two nested classes, `ClientCustomer` and `StockInfoRequestProduct`. These classes wrap SIB `Client`s into customer, respectively `StockInfoRequest`s into product functionality, as needed for the price calculation. The declaration `ClientCustomer wraps Client` states that any instance of `ClientCustomer` has a reference to an instance of `Client` assigned at object creation and available via the special keyword `wrappee`. As can be seen from this example, arbitrary pricing specific state and behavior can be associated with the abstractions from SIB.

It should be noted that `ClientCustomer` and `StockInfoRequestProduct` are virtual classes [7], i.e., they are features of an object of the enclosing class `SimpleSIBPricing`,

[4]The Caesar compiler and the source code for the example can be downloaded from http://caesarj.org

```
class SimpleSIBPricing {

  class ClientCustomer wraps Client {
    float _balance;
    float balance() { return _balance; }
    void charge(StockInfoRequestProduct p) {
      _balance -= p.price();
    }
    public String getInfo() { wrappee.getName(); }
  }

  class StockInfoRequestProduct wraps StockInfoRequest {
    float price() {
      return basicPrice() + calculateTax(); }
    float basicPrice() { return 5 +
     wrappee.getStocks().length * 0.2; }}
    float calculateTax() { ... }
  }

  after( Client c, StockInfoRequest r ):
    (call (StockInfo collectInfo(StockInfoRequest))
    && this(c) && args(r)) {
        ClientCustomer(c).charge(StockInfoRequestProduct(r));
    }

}
```

Figure 7: An implementation of `Pricing` for SIB

can be redefined in subclasses of `SimpleSIBPricing`, and are subject of late binding, just as methods are. Virtual classes already provide a good framework to encode refinements of related groups of classes, as supported by FOAs. Since Caesar also supports propagating class composition [6, 1], it is also possible to encode the semantics of FOAs' mixin-like composition of layers.

To define how a particular feature interacts with a base system at runtime, pointcuts and advice constructs are available in Caesar. Currently, Caesar uses the same pointcut model as AspectJ, hence, the pointcut definition in Fig. 7 is the same in the AspectJ solution in Fig. 6. However, the advice definitions in the respective solutions are different. The expression `ClientCustomer(c)` in the advice - syntactically similar to a constructor call without the `new` keyword - is a so-called *lifting*. Its semantics is different from a constructor: Lifting ensures that there is a unique canonical wrapper instance for every "wrappee". In particular, we do not want to create a new wrapper every time we need to navigate to a wrapper; otherwise, we would lose the state and identity of a previously created wrapper. Our lifting calls do exactly this: The expression `ClientCustomer(c)` will always yield the same instance of `ClientCustomer` for a given instance `c` of type `Client`.

Lifting expressions serve as a translator between the SIB- and the pricing world. After we have *lifted* the client `c` to its customer role, we can call the `charge` method on it. Since the method expects an instance of `StockInfoRequestProduct` as a parameter, we do the same kind of transformation (from `StockInfoRequest` to `StockInfoRequestProduct`) with the parameter `r`. Lifting fulfills in Caesar a role similar to `pertarget` and `perthis` declarations together with `aspectOf` calls in AspectJ, but with a more fine-grained control over how state is associated with object identifiers.

The discussion so far shows how Caesar combines a layer module concept similar to that of FOAs with pointcut/advice constructs similar to those of AspectJ, thereby inheriting their respective strengths. Due to the layer concept, AspectJ's problem with the flat structure of the aspect code is

avoided. At the same time advice/pointcut constructs allow to express the interaction of the features at runtime in a modular way, avoiding the scalability problem of FOAs.

In contrast to FOAs, in Caesar the relation between a layer and the base is not purely hierarchical, i.e., the relation between class definitions in a layer and the classes in the base does not need to be a one-to-one relation. In the example in Fig. 7, every nested class of `SimpleSIBPricing` happens to have a single wrappee class. In general, however, zero or more wrappees are allowed in Caesar. For example, our application could have multiple different abstractions that play the product role in the pricing feature; wrapping them into a product role adds a uniform interface to these potentially very different abstractions[5].

On the other hand, if there is no abstraction in the base application to which we can map a feature role via the `wraps` clause, Caesar offers the possibility to define arbitrarily complex mappings via user-defined constructors instead of `wraps` clauses (the `wraps` clause is just syntactic sugar for creating a single-argument constructor). For example, if we liked to map the customer role to a `Client/DBBroker` pair (as discussed in Sec. 2), we could create our own constructor that takes two parameters, namely a `Client` and `DBBroker`, respectively, and stores them in its own fields, instead of using the `wraps` clause (see [23] for more details).

4.2 Support for Variability

Caesar's pointcut/advice concept comes with a more expressive deployment mechanism, which address the variability problem of AspectJ. The advice definition in Fig. 7 makes sense only in conjunction with an *instance* of the class within which the advice is defined, hence, we need a mechanism to specify in which context the advice definitions are activated with respect to which instance of the class. This mechanism is called *deployment* in Caesar. An instance of a class that contains pointcuts and advice has to be *deployed* in order to activate its pointcuts and advice. Two different types of deployment are available: *static* (load-time) and *dynamic* (runtime) deployment.

Static deployment is expressed by the `deployed` class modifier; it means that a singleton instance of the class is created at load-time (explicit creation via `new` is not allowed), and all advice definitions refer to this singleton instance. The instance itself is available as a static field `THIS` of the class. Fig. 8 illustrates static deployment and the use of the singleton. The expression `p.ClientCustomer(cl)` is again a lifting, this time called outside the enclosing class. The type declaration `p.ClientCustomer` in Fig. 8 is related to the type system of Caesar, which uses virtual classes [16] and family polymorphism [7] in order to retain static type safety. Details of the type system are described elsewhere [22, 1]; for this paper it is sufficient to understand that `p.ClientCustomer` denotes the variant of the type `Client-Customer` valid inside the instance `p` of the enclosing pricing class. Similar to aspects in AspectJ, the activation of statically deployed aspects can only be controlled by their presence or absence in the build.

Dynamic deployment, denoted by the keyword `deploy` used as a block statement, allows to determine which variant of an aspect should be applied (or whether we need the aspect at all) at runtime. For illustration, assume a slightly different design of our pricing functionality, consisting of an abstract layer class `SIBPricing` with an abstract nested wrapper `StockInfoRequestProduct`, and two refinements of it, `SimpleSIBPricing` and `DiscountSIBPricing`, which implement the simple pricing policy from Fig. 7 and some discount pricing policy, respectively.

[5]One can define an abstract nested class `Product` which implements `price()`, while leaving abstract the functionality depending on the wrappee, e.g., `basicPrice()`. Different subclasses can refine `Product` wrapping different base classes.

```
deployed class SimpleSIBPricing { ... };
...
Client cl;
SimpleSIBPricing p = SimpleSIBPricing.THIS;
p.ClientCustomer cu = p.ClientCustomer(cl);
float f = cu.balance();
```

Figure 8: Static Aspect Deployment

```
...
Client c = ...; SIBPricing p = null;
if (...) p = new SimpleSIBPricing();
if (...) p = new DiscountSIBPricing();

deploy(p) { c.run(); }
...
```

Figure 9: Dynamic aspect deployment

Now, consider the requirement that every client can subscribe to a different pricing policy. With dynamic deployment, we can choose one of these variants at runtime and deploy them during the execution of a block, as illustrated by the code in Fig. 9. The meaning of `deploy(p)` is that the aspect `p` is active during the execution of `c.run()`. Later or concurrent executions of `Client.run` are not affected. In other words, the advice is lately bound, similarly to late method binding; hence, we call this kind of polymorphism *aspectual polymorphism*. Note that `null` is also a possible value that can be deployed, denoting no pricing at all.

If our SIB application is supposed to be completely independent of pricing, the deployment code in Fig. 9 should not be part of the base application. In Caesar, this is achieved by using another, statically deployed, helper aspect whose purpose is to deploy the correct pricing policy at the appropriate points in the dynamic control flow. For illustration, consider Fig. 10: This aspect intercepts calls to `Client.run` and continues the execution (`proceed` statement) in the context of a dynamically chosen pricing policy.

After having shortly presented the deployment mechanism

```
deployed class PricingDeployment {

  static Map pricingMap = new HashMap();

  static { // User <-> Pricing Mapping
    pricingMap.put("Klaus", new SimpleSIBPricing());
    pricingMap.put("Mira", new DiscountSIBPricing());
  }

  void around( Client c ) :
    (execution( void Client.run() ) && this(c)) {
      deploy(pricingMap.get(c.getName())) { proceed (c); }
  }
}
```

Figure 10: A deployment aspect

```
interface Pricing {

  interface Customer {
    provided void charge(Product p);
    provided float balance();
  }

  interface Product {
    expected float basicPrice();
    expected float calculateTax();
  }
}
```

Figure 11: Pricing interface

```
class SimplePricing implements Pricing {

  class Customer {
    float _balance;
    float balance() { return _balance; }
    void charge(Product p) { _balance -= p.price();}
  }

  class Product {
    float price() {return basicPrice()+calculateTax();}
  }

}
```

Figure 12: A sample implementation of `Pricing`

of Caesar, let us close this subsection by a short discussion of how it address the variability issues of AspectJ and FOAs. FOAs support variability by enabling the free composition of layers representing features of a product line. AspectJ supports only limited variability in that only one variant can exist in a system. In Caesar, features can be freely composed by instantiating and deploying instances of the compound classes representing a bound feature. With dynamic deployment and sub-typing, we can choose one variant of a feature at runtime and use it in a polymorphic way. An equivalent dynamic configurability is not possible with FOAs.

4.3 Multi-Model Decomposition and Reuse

As argued in Sec. 2 and 3, FOAs and AspectJ force the programmer to implement all features in terms of a primary decomposition as illustrated in Fig. 3 and Fig. 6. Hence, the programmer of a feature needs to know about the structure of the base in order to squeeze in the additions necessary for the feature. This is also the case with the Caesar implementation discussed so far.

However, Caesar also supports reusable features encoded in their own model and ontology and provides language constructs to express combinations of these different models. A central concept is the notion of *bidirectional interfaces* (*BI* for short) [22]. A BI serves to specify the abstractions that together make up an feature/aspect independent of the context in which the feature/aspect will be deployed[6]. BIs differ from standard interfaces in two ways. First, BIs exploit interface nesting in order to express the abstractions of an aspect and their interplay. Second, BIs divide methods into `provided` and `expected` contracts. Provided methods describe what every component that is described in terms of this model (i.e., *implements* the BI), must implement. Expected methods represent variation points of the model that are used to integrate features into a concrete system.

For illustration, the BI `Pricing` that bundles the definition of the generic pricing functionality is shown in Fig. 11. As an example for the reification of provided and expected contracts, consider `Customer.charge` and `Product.basicPrice` in Fig. 11. The ability to charge a customer for a product is at the core of pricing; hence, `Customer.charge` is marked as `provided`. The calculation of the basic price of a product, on the other hand, is specific to the context of usage which determines what will be the products to charge for; hence, `Product.basicPrice` is marked as `expected`.

The categorization of the operations into expected and provided comes with a new model of what it means to implement a BI[7]: We explicitly distinguish between *implementing* a BI's provided contract and *binding* the same BI's expected contract. Two different keywords are used for this purpose, `implements`, respectively `binds`. In the following, we refer to classes that are declared with the keyword `implements`, respectively `binds`, as *aspect implementations*, respectively *aspect bindings*.

An implementation must (a) implement all `provided` methods of the BI and (b) provide an implementation class for each of the BI's nested interfaces. In doing so, it is free to use respective `expected` methods. Furthermore, an implementation may or may not add methods and state to the BI's abstractions it implements. Fig. 12 shows a sample pricing implementation. The class `Customer`, which is identified with the interface of the same name in `Pricing`, implements the respective provided operations from the BI. The nested class `Product` adds a method `price` that computes the total price of a product. It calls expected methods from the BI for this purpose.

An aspect binding must provide zero or more nested binding classes (declared via `binds` clauses) for each of the BI's nested interfaces (we may have multiple bindings of the same interface). In these binding classes, all `expected` methods have to be implemented. Just as implementation classes can use their respective expected facets, the implementation of the expected methods of a BI and its nested interfaces can call methods declared in the respective provided facets.

The process of binding a BI instantiates its nested types for a concrete usage scenario. Hence, it is natural that in addition to their provided facets, binding classes also use the interface of abstractions from that concrete usage scenario. We say that bindings wrap abstractions from the world of the concrete usage scenario and map them to abstractions from the generic aspect world. The class `SIBPricing` in Fig. 13 shows an example of binding the interface `Pricing` from Fig. 11 for the concrete SIB usage scenario. Consider e.g., the binding of `Product` in the nested class `StockInforRequestProduct`. The latter implements all expected methods of `Product` by using the interface of the class `StockInfoRequest`.

Both classes defined in Fig. 12 and 13 are not operational, i.e., cannot be instantiated; the respective contracts implemented by them are only parts of a whole and make sense only within a whole. Operational classes that completely implement an interface are created by composing an

[6]In the following, we will use the terms feature and aspect equivalently.

[7]BIs also have consequences from a static typing point of view, but this is out of the scope of this paper (see [23] for details).

```
class SIBPricing binds Pricing {

  class ClientCustomer binds Customer wraps Client { }

  class  StockInfoRequestProduct binds Product
                              wraps StockInfoRequest {
    float basicPrice() {
        return 5 + wrappee.getStocks().length*0.2;
    }
    float calculateTax() { ... }
  }

  after( Client c, StockInfoRequest r ):
    (call (StockInfo collectInfo(StockInfoRequest))
    && this(c) && args(r)) {
        ClientCustomer(c).charge(StockInfoRequestProduct(r));
    }

}
```

Figure 13: Binding of `Pricing` for SIB

```
class SimpleSIBPricing extends
  Pricing<SimplePricing, SIBPricing>;
...
SimpleSIBPricing p = new SimpleSIBPricing();
```

Figure 14: Combining implementation and binding

implementation and a binding class, syntactically denoted as *aBI*`<`*anImpl*`,`*aBinding*`>`. This is illustrated by `SimpleSIBPricing` in Fig. 14, which composes `SimplePricing` and `SIBPricing`. Combining two classes as in Fig. 14 means that we create a new compound class within which the respective implementations of expected and provided methods are combined. The combination also takes place recursively for the nested classes: All nested classes with a `binds` declaration are combined with the corresponding implementation from the compound class. Only compound classes can be instantiated, as illustrated by the constructor call in Fig. 14.

To summarize, in Caesar every feature can be implemented with respect to its own model as described by the corresponding BI. This model can then be composed with other crosscutting models by creating an appropriate binding that describes how the two models interact which each other. The bindings describe how the abstractions of the models relate to each other structurally by creating adapters such as `StockInfoRequestProduct` in Fig. 13. This structural mapping is then used in the behavioral mapping (pointcuts and advice) that describe how the models interact in the dynamic control flow.

Since every feature can be implemented in terms of its own model, it is independent of any possible binding of the feature, hence, the implementation of a feature is reusable. This is in contrast to FOAs and AspectJ, where a feature, respectively an aspect, is tightly coupled to a specific base application. For example, the pricing algorithm indicated in Fig. 12 is reusable whereas the code in Fig. 3 can only be used in the context of the stock information broker software.

Furthermore, since aspect bindings as in Fig. 13 are independent of concrete implementations of the aspect interface, these bindings are reusable, too: They can, similarly to Fig. 14, be combined with a different implementation of `Pricing`. The separation of the two contracts, and their independent implementation allows to reuse implementations of the two contracts in arbitrary compositions.

5. RELATED WORK

Caesar is related to *Hyper/J* and its notion of multi-dimensional separation of concerns (MDSOC) [30]. 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 hyperplane. Apart from that, Caesar 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; context dependencies are declared as abstract methods. 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. Hyper/J [30] has no notion of bidirectional 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.

Adaptive Plug and Play Components (APPCs) [21] and *Aspectual Components* [13] are related to our work in that both approaches support the definition of multi-abstraction features/aspects and have a vague definition of expected and provided interfaces. However, the latter feature was not integrated well with the type system. Recognizing this deficiency, the successor model of *Pluggable Composite Adapters (PCAs)* [24] even dropped this notion and reduced the declaration of the expected interface to a set of standard abstract methods. With the notion of bidirectional interfaces, Caesar represents a qualitative improvement over all three models, as far as support for multi-abstraction aspects is concerned. Due to the lack of a BI notion, connectors and adapters in these models cannot be reused. In addition, [21] and [13] rely on a dedicated mapping sub-language that is less powerful than our object-oriented wrappers with lifting. Finally, the lack of the notion of virtual types is another drawback of these approaches compared to Caesar. Similar observations can be made about the aspectual collaborations and Object Teams approaches [14, 9], both successors of aspectual components [13].

Some aspect-oriented approaches like Prose [26] support asynchronous dynamic activation of aspects but we believe that it is very hard or even impossible to reconcile these approaches to dynamic aspects with multi-threading. Dynamic deployment is directly supported on the virtual machine layer in the Steamloom VM [5], thereby enabling a very efficient execution of dynamic deployment.

Lasagne [31] 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 offered, enabling client-specific customization of systems. Support for crosscutting models and independence of features is not in the scope of this work, though.

Hölzle [10] analyses some problems that occur when com-

bining independent components. Our proposal can be seen as an answer to the problems and challenges discussed in [10]. Mattson et al [18] also indicate the problems with framework composition, analyze reasons for these problems and investigate the state of the art of available solutions.

Our work is also related to *architecture description languages* (ADL) [27], for example Rapide [15], Darwin [17], C2 [20], and Jiazzi [19]. The building blocks of an architectural description are components, connectors, and architectural configurations. A component is a unit of computation or data store, a connector is an architectural building block used to model interactions among components and rules that govern those interactions, and an architectural configuration is a connected graph of components and connectors that describe an architectural structure. Compared to our approach, ADLs are less integrated into the common OO framework and do not have a dedicated notion of crosscutting models in order to provide a new virtual interface to a system.

We think that bidirectional interfaces might also prove very useful in the context of ADL. In ADL, components also describe their functionality and dependencies in the form of required and provided methods (so-called *ports*). The goal of these ports is to render the components reusable and independent of other components. However, although the components are syntactically independent, there is a subtle semantic coupling between the components, because a component `A` that is to be connected with a component `B` has to provide the exact counterpart interface of `B`. The situation becomes even worse if we consider multiple components that refer to the same protocol. This problem could be adressed by bidirectional interfaces.

6. SUMMARY AND FUTURE WORK

In this paper, we argued that support for crosscutting modularity is crucial for variability management. We showed how the lack of such support in most feature-oriented approaches today leads to code scattering, with the effect that the divide-and-conquer technique underlying these approaches does not scale with the increased complexity of the product line. Furthermore, we also pointed out that joinpoint interception as the mechanism for expressing crosscutting in aspect-oriented languages of the AspectJ family is not sufficient, since it lacks module support for capturing multi-abstraction slices of behavior, an important prerequisite for modularizing features whose definition generally involves several cooperating abstractions. In addition, we pointed out some problems that feature-oriented and aspect-oriented approaches have in common. Finally, we showed how Caesar supports both multi-abstraction modules and joinpoint interception and solves the problems outlined in this paper.

There are several areas of future work. First, we are working on an advanced joinpoint model for expressing the intention of a crosscut more precisely than it is the case with the more syntax-based model of Caesar today. This will make the bindings less fragile with respect to potential changes in the syntax of the base definition. Second, we are working on a formal definition of Caesar's dependent type system [1]. Finally, in the context of the TOPPrax project[8], we will use Caesar for a large-scale real-world case study.

[8]www.topprax.de

Acknowledgment

This work is supported by the TOPPrax project financed by the German Ministry of Education and Research (BMBF). We would like to thank the anonymous reviewers for their very valuable comments.

7. REFERENCES

[1] C. Anderson, S. Drossopoulou, E. Ernst, and K. Ostermann. Virtual classes with dependent types. *In preparation*, 2004.

[2] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

[3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *International Conference on Software Engineering (ICSE '03)*, 2003.

[4] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters, 2001. Available at trese.cs.utwente.nl/composition_filters/.

[5] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD 2004 Proceedings*. ACM Press, 2004.

[6] E. Ernst. Propagating class and method combination. In *Proceedings ECOOP'99*, LNCS 1628, pages 67–91. Springer-Verlag, 1999.

[7] E. Ernst. Family polymorphism. In *Proceedings ECOOP '01*, LNCS 2072, pages 303–326. Springer, 2001.

[8] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings OOPSLA '02. ACM SIGPLAN Notices 37(11)*, pages 161–173. ACM, 2002.

[9] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Proceedings of Net.ObjectDays*, Erfurt, Germany, 2002.

[10] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings ECOOP '93*, LNCS 707, pages 36–56. Springer, 1993.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings ECOOP'97*, LNCS 1241, pages 220–242. Springer, 1997.

[13] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University, March 1999.

[14] K. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations – combining modules and aspects. *Journal of British Computer Society*, 2003.

[15] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[16] O. L. Madsen and B. Møller-Pedersen. Virtual classes:

A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89. ACM SIGPLAN Notices 24(10)*, pages 397–406, 1989.

[17] J. Magee and J. Kramer. Dynamic structure in software architecture. In *Proceedings of the ACM SIGSOFT '96 Symposium on Foundations of Software Engineering (FSE)*, 1996.

[18] M. Mattson, J. Bosch, and M. E. Fayad. Framework integration problems, causes, solutions. *Communications of the ACM*, 42(10):80–87, October 1999.

[19] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *Proceedings of OOPSLA '01*, ACM SIGPLAN Notices 36(11), pages 211–222, 2001.

[20] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in C2-style architectures. In *Proceedings of International Conference on Software Engineering (ICSE) '97*, pages 692–700. IEEE Computer Society, 1997.

[21] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98. ACM SIGPLAN Notices 33(10)*, pages 97–116, 1998.

[22] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA '02, ACM SIGPLAN Notices 37(11)*, pages 52–67, 2002.

[23] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings Conference on Aspect-Oriented Software Development (AOSD) '03*, pages 90–99. ACM, 2003.

[24] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer, 2001.

[25] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of ECOOP '02. LNCS 2374*, pages 89–110. Springer, 2002.

[26] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings AOSD '02*, pages 141–147. ACM, 2002.

[27] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. PrenticeHall, 1996.

[28] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *Proceedings of ECOOP '98, LNCS 1445*, pages 550–570, 1998.

[29] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings 19th Australian Computer Science Conference*. Australian Computer Science Communications, 1996.

[30] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings International Conference on Software Engineering (ICSE) '99*, pages 107–119. ACM Press, 1999.

[31] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Joergensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the International Conference on Software Engineering (ICSE) '01*, pages 233–242. IEEE Computer Society, 2001.