An Overview of CaesarJ

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, Klaus Ostermann

Darmstadt University of Technology, D-64283 Darmstadt, Germany
{aracic,gasiunas,mezini,ostermann}@informatik.tu-darmstadt.de

Abstract. CaesarJ is an aspect-oriented language which unifies aspects, classes and packages in a single powerful construct that helps to solve a set of different problems of both aspect-oriented and component-oriented programming. The paper gradually introduces the concepts of the language and illustrates them by showing how they can be used for non-invasive component refinement and integration, as well as for development of well modularized flexible aspects. In this way we demonstrate that the combination of aspect-oriented constructs for join-point interception with advanced modularization techniques like virtual classes and propagating mixin composition can open the path towards large-scale aspect components.

1 Introduction

Aspect-oriented programming is mostly perceived as a technology for localizing crosscutting concerns by means of a mechanism to intercept execution at relevant events in order to trigger aspect-specific functionality. More recently [1, 27, 44], more attention has been given to other software engineering properties attributed to good modularization such as robustness against changes, well-defined interfaces and information hiding, or reusability.

CaesarJ[1] is an aspect-oriented language with a strong support for reusability. It combines the aspect-oriented constructs, pointcut and advice, with advanced object-oriented modularization mechanisms. From an aspect-oriented point of view, this combination of features is particularly well-suited to make large-scale aspects reusable - one can say, it enables aspect components. From a component-oriented view, on the other hand, CaesarJ is addressing the problem of integrating independent components into an application without modifying the component to be integrated or the application.

In this paper, we will give an overview of CaesarJ's features. Previous publications have focused on one of the viewpoints in isolation when presenting CaesarJ features. In [37], the language features that are relevant to component integration have been discussed, while the focus in [38] has been on features for improving the modularity and reusability of aspect code. This paper unifies the two viewpoints mentioned above and is the first comprehensive overview of CaesarJ. In particular, we will show how enabling reusable large-scale aspect

---

[1] CaesarJ can be downloaded from `caesarj.org`

components and supporting non-invasive integration of independently developed components are actually facets of the same problem, which can be addressed by the same set of language features. By doing so, the paper also contributes an in-depth presentation of CaesarJ features that have not or only sparsely been discussed in previous works in their interplay with the rest of the language.

The structure of the paper is as follows. In the next section, we illustrate the problem we want to address with a concrete example and give a rough overview of how a solution to this problem in CaesarJ would look like. In Sec. 3, we introduce the main module construct of CaesarJ - a generalized notion of classes that unifies them with the notion of packages (in terms of: sets of collaborating classes) and aspects - and demonstrate how it can be used to capture, extend and compose large-scale software components. In Sec. 4, we show how CaesarJ addresses the crosscutting integration problem by providing means for reconciling independently modularized parts of a system. In Sec. 5, we introduce the notion of *dynamic aspect deployment*, a flexible mechanism that enables control over the scope, in which an aspect component is active. The implementation of CaesarJ as an extension of Java [2] that produces JVM-compatible bytecode is outlined in Sec. 6. Related and future work is described in Sec. 7 and 8, respectively.

2 Problems Addressed by CaesarJ in a Nutshell

This section briefly surveys the limitations of mainstream object-oriented programming that CaesarJ addresses, so as to establish the frame within which to understand the in-depth technical discussion in the following sections. It does so by an example that will subsequently be used throughout the paper.

2.1 Large-scale units of modularity beyond individual classes

A significant body of research has raised the concern that classes are a too small unit of modularity [50, 36, 48, 15, 43]. We think that any large-scale piece of functionality involves a *group* of related classes; hence, abstraction, late-binding, and subtype polymorphism should be supported at the level of groups of interrelated classes. Different terminology has been used in the literature to denote such groups of interrelated classes, such as collaborations [53, 43, 36, 32, 22], layers [48, 43], teams [22], and families [14]. In this paper, the notion of a group of interrelated classes corresponds to that of a *class family* [14]. Hence, this term will be used.

To illustrate the need for carrying over the notions of abstraction, late binding, and subtype polymorphism to the level of class families, consider the class diagram in Fig.1. It shows the structure of a software for displaying hierarchical data structures (see the screenshot in Fig. 2). As indicated by Fig. 2, the data model assumed by the display component is one of a composite structure, where nodes are randomly labeled `childA`, `childB`, etc.; the implemented layout is one in which boxes displaying nodes in the hierarchy have a fixed size, independent

of the length of the displayed text; connections between nodes are shown as straight lines between the middle points of the boxes.

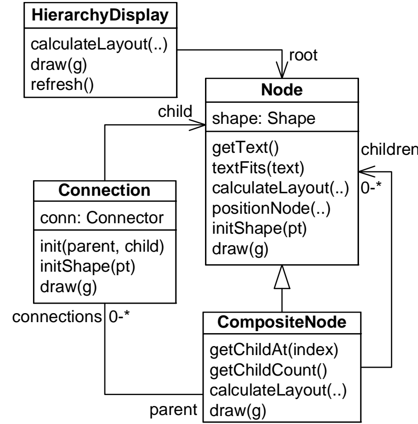

Fig. 1. Hierarchy display class diagram

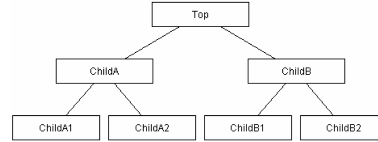

Fig. 2. Hierarchy display

Now consider some simple variations of this display functionality. One is to enable boxes capable of adjusting their size to the displayed content (screenshot in Fig. 3). Another variation would be to have right-angled connections (screenshot Fig. 4); yet another would use colors to encode the hierarchical levels. Each of these variations makes sense in isolation and in combination with others; it is reasonable to require that in scenarios where variability is important, e.g., in product line development, all of them co-exist. This calls for an incremental style of programming and flexible composition mechanisms.

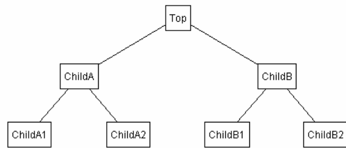

Fig. 3. ... adjusted nodes

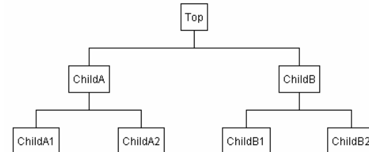

Fig. 4. ... plus right-angled connections

We can incrementally define different variations of the node and connection abstractions by subclassing `Node` and `Connection`. However, in addition, we need to make sure that any reference to `Node` and/or `Connection` is (re)bound to the respective new definition. For this, we have to redefine all classes that refer to `Node` and/oder `Connection` either by calling their constructor or by being a subclass of them. For constructor calls, we need to redefine all methods where instances of `Node` respectively `Connection` are created[2]. For subclasses of the refined classes the problem is even harder - most object-oriented languages provide no obvious solution.

This problem is well-known [24] - to cope with it, CaesarJ supports *virtual classes*, a concept that stems from the programming languages Beta [35] and

[2] Some design patterns can help with this in dynamic languages like Smalltalk.

has been refined and generalized in more recent work [12, 16]. Just like a virtual method, a virtual class is also an abstraction that has different meanings depending on the dynamic context of use. Virtual classes are defined as inner classes of an enclosing family class; just like methods and fields, they are also members of instances of their enclosing family class, called *family objects*. Hence, at any time during the execution their meaning is relative to the dynamic type of the family object.

With family classes, we can group sets of collaborating classes into a new unit (which is again a class). Fig. 5 shows that the classes of the hierarchy display component are now members (*virtual classes*) of an enclosing class `HierarchyDisplayType`. The name `HierarchyDisplayType` suggests that an instance of this family class represents a particular configuration of hierarchy display, by being a repository for the inner classes.

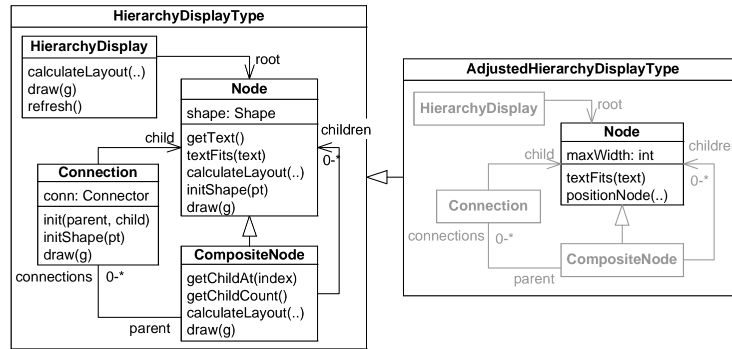

Fig. 5. Hierarchy display extension with virtual classes

Subclasses of a family class can refine inherited inner classes. Fig. 5 shows how `AdjustedHierarchyDisplayType` extends `HierarchyDisplayType` with text fitting functionality; it contains only of a refinement (a so-called *further-binding*) of the virtual class `Node`[3]. In such a further-binding, we can override inherited methods, add new methods or new state, as well as add additional superinterfaces and superclasses (the latter leads to multiple inheritance, which will be explained later).

There is a significant difference between a further-binding and a conventional subclass of `Node`: All references to the type `Node` in the other virtual classes are automatically re-bound to the refined `Node` class, when they are referred to during the execution of an object of type `AdjustedHierarchyDisplayType`. This is indicated by the gray shadows of the other virtual classes in Fig. 5. For example, in the context of an instance of `AdjustedHierarchyDisplayType`, a `CompositeNode` is a subclass of the refined `Node` class. Similarly, instance creation expressions are also late bound.

[3] One could similarly refine `Connection` in a subclass `AngularHierarchyDisplayType` to extend `HierarchyDisplayType` with angular connections.

A related problem addressed by CaesarJ is how to compose different variations of some basic functionality. In mainstream object-oriented languages, a subclass is defined to a particular superclass. This lack of abstraction over the implementation of the superclass hinders reusability: The variation defined by the subclass cannot be reused with other superclasses. For illustration, consider that it makes sense to compose different variants of hierarchy displays, e.g., `AdjustedHierarchyDisplayType` and `AngularHierarchyDisplayType` to have a layout strategy with both, adjusted nodes and angular connections (see screenshot in Fig. 4). However, such a composition is not possible, if both subclasses are defined to a concrete implementation of `HierarchyDisplayType`.

To solve this problem CaesarJ shares with gbeta [12] a mixin-based class composition mechanism [13]: (a) classes (simple or families) are mixins, i.e., their superclass can be exchanged [9], and (b) mixin composition of family classes automatically propagates into their inner classes. By being mixins defined to a common supertype, modules that implement the display layout strategy with adjustable nodes and with angular connections can be composed with each other; superclasses in the inheritance hierarchy are replaced according to specific composition rules. Mixin composition propagation ensures that the composition structure is propagated from families to their inner classes. Unambiguousness of the composition is ensured by the composition order and a linearization algorithm to be discussed later in this work.

2.2 Crosscutting composition mechanisms

The composition mechanism outlined so far is hierarchical: in order to compose different modules in a non-trivial way, they must have common ancestors because only those inner class definitions are merged that are further-bindings of a common class definition. In our example, all variations inherit the structure of `HierarchyDisplayType`. It is this shared structure that makes them composable with each other; the composition of differently structured class families is still possible, but not very useful, because it would not compose any inner classes.

In many cases, however, one would like to compose independent (family) classes that do not have a hierarchical relationship and hence no common ancestor, in a meaningful way. For illustration, consider the class diagram in Fig. 6 - part of a software system for automating the administration of companies. Assume that we are involved in implementing a GUI, which is capable of displaying the company structure. Given the two components we already have, `HierarchyDisplayType` (or any of its variations) and `Company`, it is desirable to "simply" compose them. The composition cannot, however, be performed automatically by mixin-composition, because the operands of the composition are not in a hierarchical relationship as the variations of the hierarchy display functionality.

There are two issues involved in integrating hierarchy display and company components. On the one hand, the generic visualization functionality of the display component must be customized to the specifics of the company administration structure. On the other hand, the functionality of the company component

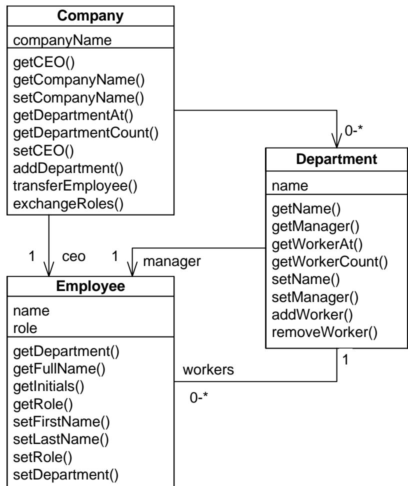

Fig. 6. Company data model

must be tuned in such a way that changes to the company state, e.g., moving of employees from one department to another, are signaled to the display component so that the latter can refresh itself.

Let us quickly discuss why this is a problem in conventional OO languages. One could use the adapter pattern[17] for customizing the hierarchy display functionality to the company structure and the observer pattern[17] for the display refresh aspect of the composition, as outlined in Fig. 7 and Fig. 8 respectively. However, the resulting composition exhibits crosscutting structure.

First, the adaptation of the generic display functionality to the structure of the company software requires a lot of infrastructural logic: hash tables to maintain adapter identity [24] and ubiquitous type casts in the adapter code.

Second, the adaptation logic cuts across various display variations. Adapters are implemented in subclasses of concrete implementations of the types `Node` and `CompositeNode`, e.g., those encoding the standard layout; hence, they only work with that specific implementation. The adaptation logic must be duplicated for all variations of the display implementation[4].

Third, the observation logic for refreshing the display cuts across the modular structure of the company component. Notification logic is not explicit and is mixed within data model operations. Besides, the composition is not incremental: adding observation support requires changing existing code. The lack of means for explicit expression of the crosscutting structure of the display refreshing aspect results in a lot of infrastructural code for observer registration and event dispatch.

[4] A part of the adaptation code can be made reusable in a language that supports multiple inheritance.

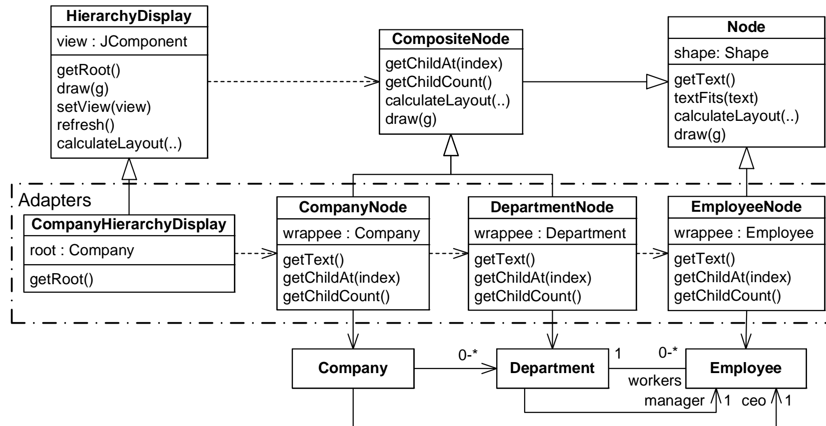

Fig. 7. Integrating components with adapters

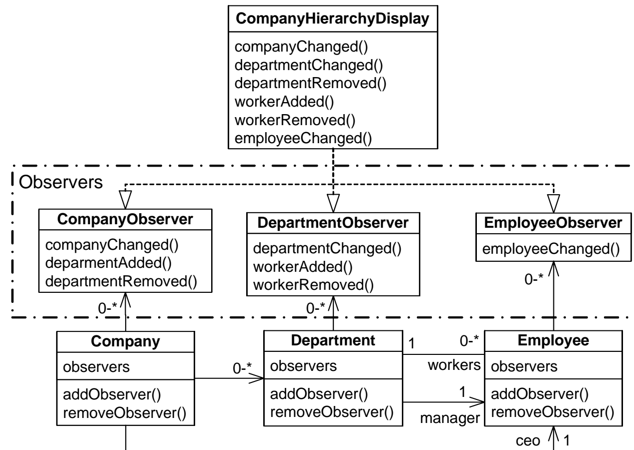

Fig. 8. Integrating components with observers

To cope with the outlined problems, CaesarJ provides two dedicated mechanisms for expressing crosscutting compositions. First, an AspectJ-like pointcut-advice mechanism [26] is available for expressing modifications of existing behavior incrementally. Second, CaesarJ provides a mechanism for automatic management of associations between company objects and their adapters to roles in the hierarchy display concept world. The integration logic is expressed in so-called binding classes.

Similar to the hierarchical variations, bindings are also expressed in family classes as variations on the display functionality, i.e., they rely on the concepts of virtual classes and propagating mixin composition. This has a twofold effect: (a) type casts present in adapters are no longer needed, and (b) a binding of

the hierarchy display can be reused (composed) with any hierarchy display variation. Details on bindings as well as the variability enabled by CaesarJ will be discussed in Sec. 4.

3 Class Families, Refinement, and Mixin Composition

In this section, we will first introduce the notion of virtual classes as realized in CaesarJ, then we will talk about composing class hierarchies and how the type system supports polymorphic usage of class families, and finally we will explain the semantics of abstract virtual classes and introduce the notion of collaboration interfaces.

3.1 Virtual Classes, Type System, and Family Polymorphism

With virtual classes, we can group sets of collaborating classes into a new unit (which is again a class), and subclasses of such a unit can refine inherited inner classes.

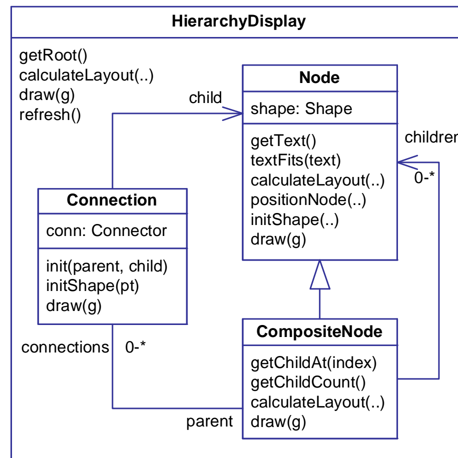

Fig. 9. Alternative design of hierarchy display with confined types

To understand the effect of making a class a virtual member of another class, consider another version of the example from Fig. 5. Fig. 9 shows an alternative design for the hierarchy display component: the state and methods of the virtual class `HierarchyDisplay` from Fig. 5 are moved to the top-level. These two designs differ from each other in an important way: The `Node`, `CompositeNode`, and `Connection` classes are members of individual `HierarchyDisplay` instances in Fig. 9, whereas in Fig. 5 all `HierarchyDisplay` instances share the same `Node`, `CompositeNode`, and `Connection` classes.

That is, in Fig. 5 different instances of `HierarchyDisplay` could share or exchange parts of the displayed data, whereas in Fig. 9 the family class acts as a unit of *confinement*: The type system prevents that nodes or connections that

```
1  cclass HierarchyDisplay {
2      cclass Node { ... }
3      cclass CompositeNode extends Node { ...
4          calculateLayout() { ...
5              Connection c = new Connection(); ... }
6      }
7      cclass Connection { ...
8          void initShape(Point pt) { ... }
9      }
10     Node root; ...
11 }
12 cclass AdjustedHierarchyDisplay extends HierarchyDisplay {
13     cclass Node { ...
14         int maxwidth;
15     }
16     void foo(Node n) {... n.maxwidth ... }
17 }
18 cclass AngularHierarchyDisplay extends HierarchyDisplay {
19     cclass Connection { ...
20         void initShape(Point pt) { ... }
21     }
22 }
```

Listing 1. Code for HierarchyDisplay

stem from different families (in this case the hierarchy display instance is the family) will ever be mixed. This is also the reason why the name of the family class in Fig. 5 is `HierarchyDisplayType`: an instance of it represents a particular configuration or type of hierarchy displays, whereby multiple instances of `HierarchyDisplay` might be instances of the same hierarchy display type.

Choosing between these two design alternatives is an important design decision to be made on a case-by-case basis. Such design considerations are out of the scope of this paper - the distinction between the two alternatives was done with the sole purpose to highlight the effect of making a class a virtual member of another class. In the remainder of the paper we will use the design from Fig. 9 in our examples. Listing 1 shows source code that corresponds to the design in Fig. 9 as well as two extensions, `AdjustedHierarchyDisplay` and `AngularHierarchyDisplay`, of the base component. The keyword `cclass` is used instead of `class` in order to differentiate pure Java classes from (virtual) CaesarJ classes.

Since all virtual classes depend on their family, all types that refer to virtual classes are implicitly (or explicitly) annotated with a path to their owner family object. For example, the type `Connection` in line 5 of List. 1 implicitly means `HierarchyDisplay.this.Connection`[5], because the actual definition of this type depends on the owner family object. Similarly, the superclass declaration in line 3 should be read as `extends HierarchyDisplay.this.Node`, meaning that the actual superclass definition depends on the family object. The effect of late-bound types is also illustrated in line 16: A type cast is not necessary to access the `maxwidth` property, because it is known that all nodes of an `AdjustedHierarchyDisplay` have this property.

[5] In Java, as well as in CaesarJ, the owner object is referenced by qualifying ẗhisẅith its class name.

```
1  hd.Node findChild(final HierarchyDisplay hd, hd.CompositeNode n, String text) {
2      for (int i = 0; i < n.getChildCount(); i++)
3        hd.Node m = n.getChildAt(i);
4        if (m.getText().equals(text)) return m;
5      }
6      return null;
7  }
8  ...
9  final HierachyDisplay hda = new AdjustedHierarchyDisplay();
10 hda.CompositeNode cna = ...;
11 final HierachyDisplay hdb = new AngularHierarchyDisplay();
12 hdb.CompositeNode cnb = ...;
13 hda.Node n1 = findChild(hda, cna, someString); // ok
14 hdb.Node n2 = findChild(hdb, cnb, someString); // ok
15 hda.Node n3 = findChild(hdb, cnb, someString); // static error
```

Listing 2. Illustration of path-dependent types and family polymorphism

If virtual classes are used as types outside their family classes, the implicit scoping must be replaced by an explicit specification of the owner family object. This is illustrated in List. 2, which shows a method defined in some class outside `HierarchyDisplay` as well as some code that uses this method. The type declaration `hd.CompositeNode` in the signature of the method `findChild` means that only instances of `CompositeNode` that belong to the family `hd` may be passed as the second parameter to the method. Similarly, the return type `hd.Node` means that the returned node instance belongs to the family `hd`.

Hence, the calls in line 13 and 14 are correct, whereas the call in line 15 causes a static type error, because the variable `n3` belongs to the family `hda` rather than `hdb`. In general, the type checker makes sure that families are never mixed, i.e., an object `o1` can only be compatible to an object `o2`, if `o1` and `o2` belong to the same family.

Listing 2 also illustrates the concept of *family polymorphism* [14]: The `findChild` method can be used polymorphically with different families (in the example `hda` and `hdb`). The type checker for these kinds of dependent types is highly-non trivial but not in the focus of this work. A full formalization of the core constructs of virtual classes as used in CaesarJ (operational semantics, type system, and a soundness proof) can be found in [16].

3.2 Composing Class Hierarchies

As illustrated in List. 3, CaesarJ classes can be composed with the operator `&`. The class `AdjustedAngularHierarchyDisplay` composes `AdjustedHierarchyDisplay` and `AngularHierarchyDisplay`. The composition operator realizes a variant of multiple inheritance that linearizes the superclasses, thereby avoiding ambiguities w.r.t. method dispatch and w.r.t. to sharing or duplicating of inherited state. The composition uses a variant of *C3 linearization* [3, 13], which produces a unique and predictable linearization of the inheritance graph. In the case of `AdjustedAngularHierarchyDisplay`, the linear order of superclasses produced by the linearization algorithm is `[HD,AngHD,AdjHD,AdjAngHD]`, whereby `HD` is an ab-

```
1 cclass AdjustedAngularHierarchyDisplay extends
2                  AdjustedHierarchyDisplay & AngularHierarchyDisplay {}
```

Listing 3. Composing variants of hierarchy display

breviation for `HierarchyDisplay`, `Ang` is an abbreviation of angular, and `Adj` is an abbreviation of `Adjusted` and the last mixin is the most specific one.

The order of the mixin operands of the operator `&` is important in determining the order of the mixins in the linearized chain. The operator `&` is not commutative and the operand on the left hand side is more specific than the one on the right hand side. The left most mixin is the most specific one. The same linearization algorithm is also used if a further-binding of a class declares additional superclasses.

In the context of virtual classes, this composition operator propagates the composition into inner classes. This means that all inner classes of the composed classes that are further-bindings of a common class are automatically composed via linearization, whereby the linearization of the enclosing family class determines the linearization of the inner classes. This composition works recursively with arbitrary levels of nesting. In our example, this means that `AdjustedAngularHierarchyDisplay` combines further-bindings of both `AdjustedHierarchyDisplay` and `AngularHierarchyDisplay`. Since the inner classes of a class can represent an entire class hierarchy, the `&` operator can effectively be used to extend and compose class hierarchies.

$$\mathcal{A}ssemble(\overline{\mathsf{p}}, \mathsf{C}) = \mathcal{L}inearize([\,\mathcal{E}xpand(\overline{\mathsf{p}}, \mathsf{p}) \mid \mathsf{p} \leftarrow \mathcal{D}efs(\overline{\mathsf{p}}, \mathsf{C})\,])$$

$$\mathcal{D}efs(\overline{\mathsf{p}}, \mathsf{C}) = [\,\mathsf{p.C} \mid \mathsf{p} \leftarrow \overline{\mathsf{p}}, \mathcal{C}lassDef(\mathsf{p.C}) \neq \bot\,]$$

$$\mathcal{E}xpand(\overline{\mathsf{p}}, \mathsf{p}) = \mathcal{L}inearize([\,\mathcal{A}ssemble(\overline{\mathsf{p}}, \mathsf{C}') \mid \mathsf{C}' \leftarrow \mathsf{C}_1 ... \mathsf{C}_n\,])\,\mathsf{p}$$
$$\text{where } \mathcal{C}lassDef(\mathsf{p}) = \textbf{cclass } \mathsf{C} \textbf{ extends } \mathsf{C}_1 \& ... \& \mathsf{C}_n \;\texttt{\{ ... \}}$$

$$\begin{aligned}
&\mathcal{L}inearize(\mathsf{nil}_{\overline{\mathsf{p}}}) = \mathsf{nil}_{\mathsf{p}} \\
&\mathcal{L}inearize(\overline{\overline{\mathsf{p}}}\;\overline{\mathsf{p}}) = \mathcal{L}in2(\mathcal{L}inearize(\overline{\overline{\mathsf{p}}}), \overline{\mathsf{p}}) \\
&\mathcal{L}in2(\mathsf{nil}_{\mathsf{p}}, \mathsf{nil}_{\mathsf{p}}) &&= \mathsf{nil}_{\mathsf{p}} \\
&\mathcal{L}in2(\overline{\mathsf{p}}\;\mathsf{p}, \overline{\mathsf{p}}'\;\mathsf{p}) &&= \mathcal{L}in2(\overline{\mathsf{p}}, \overline{\mathsf{p}}')\;\mathsf{p} \\
&\mathcal{L}in2(\overline{\mathsf{p}}, \overline{\mathsf{p}}'\;\mathsf{p}') &&= \mathcal{L}in2(\overline{\mathsf{p}}, \overline{\mathsf{p}}')\;\mathsf{p}' \text{ if } \mathsf{p}' \notin \overline{\mathsf{p}} \\
&\mathcal{L}in2(\overline{\mathsf{p}}\;\mathsf{p}, \overline{\mathsf{p}}') &&= \mathcal{L}in2(\overline{\mathsf{p}}, \overline{\mathsf{p}}')\;\mathsf{p} \text{ if } \mathsf{p} \notin \overline{\mathsf{p}}' \\
&\mathcal{L}in2(\overline{\mathsf{p}}\,\mathsf{p}'\overline{\mathsf{p}}''\mathsf{p}, \overline{\mathsf{p}}'\mathsf{p}') &&= \mathcal{L}in2(\overline{\mathsf{p}}\,\overline{\mathsf{p}}''\mathsf{p}, \overline{\mathsf{p}}')\;\mathsf{p}' \\
& &&\text{(Note: use first case that matches)}
\end{aligned}$$

Fig. 10. Mixin computation for class C given mixin list $\overline{\mathsf{p}}$ of enclosing family class

A definition of how mixins are linearized and composed is given in Fig. 10[6]. Therein, p denotes a mixin by the static path $\mathsf{C}_1 ... \mathsf{C}_n$, C_i class names, that denotes the lexical position of the class body corresponding to `p`. For example, the mixin for the class definition in line 3 of Listing 1 is `HD.CompositeNode`. The

[6] These definitions are part of the aforementioned formalization of virtual classes [16].

notation $\overline{\mathsf{p}}$ denotes a *list* of mixins $\mathsf{p}_1, ..., \mathsf{p}_{|\overline{\mathsf{p}}|}$, e.g., $\overline{\mathsf{p}}$ = `[HD,AdjHD,AdjAngHD]`, and $\overline{\overline{\mathsf{p}}}$ denotes a list of mixin lists. The $[\,...\mid...\,]$ notation is used to denote *list comprehensions* as e.g., in Haskell or Python[7].

Given a class C and the mixin list of the enclosing family $\overline{\mathsf{p}}$, the $\mathcal{A}$*ssemble* function computes the mixin list that determines the definition of C relative to $\overline{\mathsf{p}}$. For illustration, we will simulate the evaluation of $\mathcal{A}$*ssemble*(`[HD,AdjHD], CompositeNode`), which calculates the mixin list of `CompositeNode` in the context of `AdjustedHierarchyDisplay`, resulting in `[HD.Node,AdjHD.Node,HD.CompositeNode]`. To do so, $\mathcal{A}$*ssemble* first calls $\mathcal{D}$*efs* to collect all the definitions of `CompositeNode` (our C) located in any of the class bodies specified by `[HD,AdjHD]` (our $\overline{\mathsf{p}}$) [8]. The result is HD.CompositeNode, because there is only one definition of `CompositeNode` and no further-binding.

The complete mixin list for a class C must also include the mixins of all its ancestors. For this purpose, $\mathcal{A}$*ssemble* applies $\mathcal{E}$*xpand* over the list of mixins returned by $\mathcal{D}$*efs* and linearizes the result. For each p in this list, $\mathcal{E}$*xpand* computes the mixin list for each superclass of p, again relative to the enclosing mixin list $\overline{\mathsf{p}}$. For this purpose, $\mathcal{E}$*xpand* recursively applies $\mathcal{A}$*ssemble* over the list of all superclasses, linearizes the result, and adds the mixin p at the end. The recursion is well-defined because it recurses only on the superclasses and the superclass relation has no cycles in a well-formed program (we stop when we reach a top-level object, which has a trivial mixin list).

For our setting of p = `HD.CompositeNode`, and $\overline{\mathsf{p}}$ = `[HD,AdjHD]`, $\mathcal{E}$*xpand* will be applied to `HD.Node` - the only superclass of `HD.CompositeNode`, which will cause $\mathcal{A}$*ssemble*(`[HD,AdjHD], Node`) to be recursively called, resulting in `[HD.Node,AdjHD.Node]`. Since `Node` does not have any further superclases, this is the end of the recursion - the mixin `HDComposite` is added to `[HD.Node,AdjHD.Node]` yielding the overall result: $\mathcal{A}$*ssemble*(`[HD,AdjHD], CompositeNode`) = `[HD.Node,AdjHD.Node,HD.CompositeNode]`.

Linearization is a technique for topological sorting of an inheritance graph, so that method calls can be dispatched along the calculated order. The function $\mathcal{L}$*inearize* in in the lower part of Fig. 10 linearizes a list of mixin lists, i.e., it produces a single mixin list which contains the same mixins as those in the operands, in an order which is controlled by the operands. $\mathcal{L}$*inearize* is defined in terms of a binary linearization function, $\mathcal{L}$*in2*. This function is an extension of the C3 linearization algorithm [3, 13]. The linearization algorithm has been designed so that the ordering of mixins in a virtual class can be controlled by the programmer of a subclass, in a similar spirit as when the programmer of a subclass can decide to override a method in any mainstream object-oriented programming language, see [3, 13].

---

[7] For example, $[\,2n \mid n \leftarrow 1...5, n > 3\,]$ is the list $[8, 10]$

[8] The function $\mathcal{C}$*lassDef*, which is not defined here, simply looks up a class in the program.

```
abstract public cclass IHierarchyDisplay {
    abstract public Node getRoot(); /* data model */
    abstract public void calculateLayout(); /* visualization */
    abstract public void draw(Graphics g); /* visualization */
    abstract public void refresh(); /* visualization   */
    ...
    abstract public cclass Node {
        abstract public String getText(); /* data model */
        abstract public boolean textFits(String text); /* visualization */
    }
    abstract public cclass CompositeNode extends Node {
        abstract public Node getChildAt(int i); /* data model */
        abstract public int getChildCount(); /* data model */
        abstract public void calculateLayout(); /* visualization */
    }
}
public cclass HierarchyDisplay extends IHierarchyDisplay { ... }
```

Listing 4. Collaboration interface of hierarchy display

```
final public IHierarchyDisplay hier = new HierarchyDisplay();
hier.CompositeNode node = hier.new CompositeNode(); /* error */
final public IHierarchyDisplay2 hier2 = new HierarchyDisplay2();
hier2.CompositeNode node = hier.new CompositeNode(); /* ok */
```

Listing 5. Polymorphic instantiation of classes of an abstract family class

3.3 Abstract Classes and Collaboration Interfaces

The benefits of polymorphism can be maximized by using abstract family classes. A separate interface concept is not necessary because we do not have the single inheritance bottleneck. For example, we can use an abstract family class `IHierarchyDisplay` to define the public interface of the `HierarchyDisplay` component, as shown in List. 4. The abstract family class exposes the public methods of the component as well as the classes that should be visible to the clients, e.g. the abstraction in List. 4 does not expose the `Connection` class.

Declaring a class as abstract means that it cannot be instantiated. According to this rule, we cannot create instances of the class `IHierarchyDisplay`. The virtual classes `Node` and `CompositeNode` are declared as abstract too. This means that they cannot be instantiated polymorphically through a family variable with type `IHierarchyDisplay`; so the line 2 in List. 5 will generate a compiler error.

In a similar way, we can allow polymorphic instantiation of virtual classes by declaring them as concrete. For example, List. 6 shows an alternative interface to the hierarchy display component, which declares its virtual classes `Node` and `CompositeNode` as concrete even though they contain abstract methods. The intent of such a design is to allow their polymorphic instantiation as shown in line 4 of List. 5. Here, the instantiation is requested through the abstract interface, but the class that is actually instantiated is `CompositeNode` of `HierarchyDisplay2`, which belongs to a concrete family class and must implement all the inherited abstract methods.

Abstract classes are also allowed within concrete classes. For example, in an alternative design of `HierarchyDisplay` component, there could be a new class

```
1  abstract public cclass IHierarchyDisplay2 {
2      ...
3      public cclass Node {
4          abstract public String getText();   ...
5      }
6      public cclass CompositeNode extends Node {
7          abstract public Node getChildAt(int i); ...
8      }
9  }
10 cclass HierarchyDisplay2 extends IHierarchyDisplay2 { }
```

Listing 6. Alternative interface to the hierarchy display

`LeafNode` to represent leaf nodes, while `Node` would serve only as an abstraction to define the common interface for all types of nodes. In such case, declaring `Node` as abstract would prevent its instantiation.

If we do not know whether a virtual class will be concrete in concrete subfamilies, it is better to declare it as abstract, because we can override an abstract class with a concrete one, but not the other way around. Overriding a concrete class with an abstract one is not allowed, because it would break the soundness of polymorphic instantiation.

In Java, a class containing an abstract method, must be declared as abstract. In CaesarJ this rule is weakened: *a method can be abstract when at least one of its enclosing classes is abstract*. This rule is sufficient to ensure that abstract methods will never be called, because it excludes the possibility of direct instances of the class declaring the method. According to this rule, it is legitimate to have concrete classes with abstract methods inside an abstract family class, which is the case in List. 4. It is also possible to have abstract classes with abstract methods inside a concrete class.

Abstract classes used as interfaces enable a more fine-grained separation of the different concerns of our hierarchy display. One of these concerns that we might want to separate is how the data to be displayed is represented. Our previous implementation stored the data model directly in corresponding fields. The comments in Listing 4 now identify a set of methods which are the interface to the data model. With inheritance and class composition we can now separate the display logic from the data model.

Listing 7 shows a sample implementation of the data model. The corresponding family class is abstract because it is only an implementation of the data model; hence, the part of the `IHierarchyDisplay` interface responsible for the visualization is missing. On the other hand, Listing 8 shows a version of `HierarchyDisplay` that does not define the data model, hence, it is abstract as well. Note that the code in Listing 8 uses the methods responsible for the data model without defining them. In an appropriate composition, such as in Listing 9, both facets of a hierarchy display are composed. Since the composition is complete, it does not need to be abstract and can be used directly.

There are two important aspects in the design embodied in Listings 4, 8, and 9: (a) the partition of the interface methods into different facets, as indicated by

```
abstract public cclass MutableHierarchyModel extends IHierarchyDisplay {
    protected Node root;
    public void getRoot() { return root; }
    ...
    public cclass Node {
        protected String text;
        public String getText() {
            return fitsText(text) ? text : text.substring(0, 1);
        } ...
    }
    public cclass CompositeNode {
        protected List children = new LinkedList();
        public Node getChildAt(i) { return (Node)children.get(i); }
        public int getChildCount() { ... }
        ...
    }
}
```

Listing 7. Hierarchy display data model as a separate module

```
abstract public cclass HierarchyDisplay extends IHierarchyDisplay {
    protected Component view = null;
    public void calculateLayout()
        getRoot().calculateLayout();
        refresh ();
    }
    public void draw(Graphics g) { ... }
    public void refresh() { ... }
    ...
    abstract public cclass Node {
        protected TextShape shape = new Rectangle();
        public void draw(Graphics g) { shape.setText(getText()); shape.draw(g); }
        public boolean textFits(String text) { ... }
        ...
    }
    abstract public cclass CompositeNode {
        protected List connections = new LinkedList();
        public void draw(Graphics g) {
            super.draw(g);
            for (int i1 = 0; i1 < getChildCount(); i1++) getChildAt(i1).draw(g); ...
        }
        public void calculateLayout() { ... }
        ...
    }
    public cclass Connection { ... }
}
```

Listing 8. HierarchyDisplay without data model implementation

the comments in List. 4, and (b) the design rule that subclasses of the interface responsible for one facet implement only those methods belonging to this facet (whereby any method declared in the interface can be called). In such a design,

```
public cclass MutableHierarchyDisplay
    extends HierarchyDisplay & MutableHierarchyModel { }
```

Listing 9. Reconstructing the mutable hierarchy display component

the interface in List. 4 controls the collaboration between different facets of its implementation, hence, we call such interfaces *collaboration interfaces*.

It is tempting to turn this design pattern into a language feature, so that conformance to a particular interface facet is checked by the compiler. In previous publications [37, 38] we actually proposed to divide the methods of such a *collaboration interface* into two generic fixed facets: *expected* and *provided*. In the implementation of CaesarJ, we dropped this mechanism because it is not general enough. In general, there can be many different facets of an interface, not just two. We are currently working on a new interface concept that allows more freedom in this regard while still retaining static checking of classes with respect to the facets they are responsible for.

4 Crosscutting Integration

In this chapter, we will introduce CaesarJ features for supporting crosscutting composition.

4.1 Bindings

The implementations of hierarchy display facets that are presented in List. 7 and List. 8 are self-consistent and completely encapsulated behind the collaboration interface. Alternatively, facets can be implemented as adapters of already existing classes. In our example, we might want to display the company model from Fig. 6 with our hierarchy display. The company model can indeed be seen as a data model for our hierarchy display, except that it does not fit to its internal modular structure. Hence, in the following we will implement the data model facet of the hierarchy display as an adapter to the company model, which allows to view the company model as a data model for the hierarchy display.

A family class which implements a component facet by adapting external classes is called *binding*. Concrete family classes are produced by combining bindings with the families implementing the component functionality to be integrated. Fig. 11 shows how the family class `CompanyHierarchyDisplay` for visualizing the company organizational hierarchy is defined as a combination of the implementation of the visualization facet of the `IHierarchyDisplay` interface implemented in `HierarchyDisplay` and the data model facet of `IHierarchyDisplay` defined as its binding to the company model `CompanyHierarchyBinding`.

Bindings map between types from two domains by means of *wrapper* classes - dynamic extensions of other classes, called *wrappees*. A wrapper can introduce new state and operations, as well as adapt the wrappee to required interfaces. The wrapper-wrappee relationship is established by the keyword `wraps`. A wrapper can access its wrappee by means of the special identifier `wrappee`.

To map the display and the company domains of our example, we have to bind hierarchy display nodes to company model objects. One wrapper class is needed for each type of display node. Top nodes are bound to company objects, nodes at the second level of the display hierarchy to department objects, and

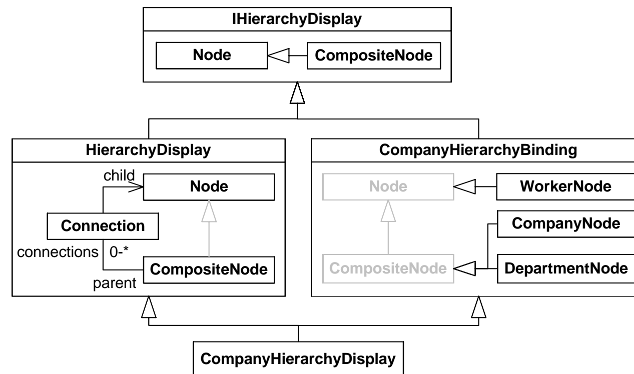

Fig. 11. Integration of hierarchy display into company model

bottom nodes to employees. In List. 10, `WorkerNode` is a wrapper for `Employee`. It adapts `Employee` to the data model facet of `Node` by implementing `getText`, the only method related to the data model in `Node`, using methods of `Employee`. Wrappers for `Company` and `Department` turn these classes into composite nodes in the display world: they inherit from `CompositeNode` and implement its data model related methods for retrieval of text and children.

Wrappers are created by *wrapper constructors*, which take as parameters the objects to be wrapped and return the corresponding wrapper objects. For example, a `DepartmentNode` should return the nodes representing the workers at that department as its children. Its method `getChildAt(i)` in List. 10 retrieves the `i`-th employee of the department and wraps it into a `WorkerNode` object. A wrapper constructor differs from a conventional instantiation: given a certain wrappee object, `o`, only the first call of a wrapper constructor with `o` as a parameter creates a new wrapper for `o`; consecutive calls will always return the same wrapper instance.

Such "wrapper recycling" ensures that there is only one wrapper for one wrappee per binding and, hence, enables *stateful wrappers*. Attaching additional state to wrapped objects is important: a component to be integrated has its own state which cannot be inferred from wrappee's state. For example, the position of a node and its graphical attributes cannot be inferred from the data model and must be stored in node objects.

However, an object can have multiple wrappers of the same type within different family instances. The wrapper constructor call `WorkerNode(..)` in List. 10 is in fact an abbreviation for `this.WorkerNode(..)`. Wrappers are also virtual classes - their meaning is relative to an enclosing family. Wrapper constructors are also available outside the family class by explicitly qualifying their calls with a reference to a family instance.

Mapping between the abstractions in their respective domains is not enough for full integration: the components often need to adapt their behavior within the

```
abstract public cclass CompanyHierarchyBinding extends IHierarchyDisplay {
   protected Company _company = null;
   public Node getRoot() { return CompanyNode(_company); }
    ...
   public cclass WorkerNode extends Node wraps Employee {
       public String getText() {
          return textFits(wrappee.getFullName()) ?
             wrappee.getFullName() : wrappee.getInitials();
       }
   }
   public cclass DepartmentNode extends CompositeNode wraps Department {
      public String getText() { ... }
      public Node getChildAt(int i) {
         return WorkerNode(wrappee.getWorkerAt(i));
      }
      public int getChildCount() { return wrappee.getWorkerCount(); }
   }
   public cclass CompanyNode extends CompositeNode wraps Company {
      public Node getChildAt(int i) {
         return DepartmentNode(wrappee.getWorkerAt(i));
      }
       ...
   }
   pointcut departmChildrenChange() : execution(* Department.addWorker(..)) ||
      execution(* Department.removeWorker(..));
   pointcut companyChildrenChange() : execution(* Company.addDepartment(..)) ||
      execution(* Company.removeDepartment(..));
   pointcut displayChange() : execution(* company.*.set*Name(..)) ||
      departmChildrenChange() || companyChildrenChange();
   after(Department d) : departmChildrenChange() && this(d) {
      DepartmentNode(d).calculateLayout();
   }
   after(Company c) : companyChildrenChange() && this(c) {
      CompanyNode(c).calculateLayout();
   }
   after() : displayChange() && !cflowbelow(displayChange()) { refresh(); }
}
```

Listing 10. Company hierarchy binding

composition. In our example, organizational changes, e.g., transfer of employees from one department to another, affects the layout of the hierarchy display and should cause the layout of certain branches in the hierarchy to be recalculated. In CaesarJ, such behavioral integrations are expressed within a binding by means of pointcuts and advice. CaesarJ supports AspectJ-like pointcuts and advice. In the following discussion, we assume that the reader is familiar with the crosscutting mechanisms in AspectJ [26] and the advantages of observing with pointcuts [20, 27].

In our example, the binding in List. 10 uses pointcuts to observe relevant changes. Pointcuts `departmChildrenChange` and `companyChildrenChange` observe `DepartmentNode`, respectively `CompanyNode` children changes. The pointcut `displayChange` observes any kind of change that affect the company hierarchy display[9]. Display update is done in advice (List. 10) using methods of the collaboration interface (List. 4). Top-level methods, such as `refresh`, can be called directly,

[9] It reuses the pointcuts for children changes and quantifies over all methods that affect names of the data model objects. We use the `cflowbelow` pointcut to ensure

```
public cclass CompanyHierarchyDisplay
   extends HierarchyDisplay & CompanyHierarchyBinding { }
```

Listing 11. Component for company hierarchy display

the methods of the nodes, e.g. `calculateLayout`, are called on the corresponding wrapper object.

As already mentioned, the company hierarchy display component is constructed by applying mixin composition to the implementation of its visualization facet and its company binding (List. 11). Fig. 12 depicts the implicit class diagram inside `CompanyHierarchyDisplay`, which is the result of merging the inherited classes and their relationships. It contains both the wrapper classes from the binding as well as the `Connection` class from the family `HierarchyDisplay`, which implements the visualization facet. The classes `Node` and `CompositeNode` of `HierarchyDisplay` become the superclasses of the wrapper classes in the context of `CompanyHierarchyDisplay`. Thus, the resulting wrapper classes inherit both the functionality related to the data model and to the visualization.

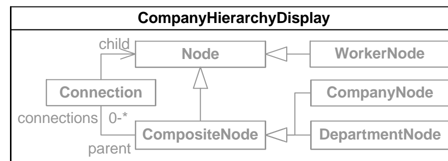

Fig. 12. Class diagram of the virtual classes within CompanyHierarchyDisplay

Of course, bindings do not necessarily need to be coded to collaboration interfaces. If the code of a binding is not reusable, it can be defined as a simple subclass of the family type that we want to adapt in a specific context. For example, `CompanyHierarchyDisplay` could also be implemented as subclass of `HierarchyDisplay`. Then the binding would be a concrete family class and could be instantiated directly.

4.2 Dynamic Wrapper Selection

This section considers the issue of defining wrappers in the presence of inheritance hierarchies. For example, the class `Employee` may have various subclasses, e.g., `InternalEmployee` and `ExternalEmployee` to distinguish between internal employees of the company and the employees subcontracted from other companies. We may want this difference to be reflected in the display of the organizational structure. The question is how to define wrappers for subclasses of already wrapped classes.

that the display is refreshed only once after a sequence of changes that constitutes a logical transaction.

```
abstract public cclass CompanyHierarchyBinding extends IHierarchyDisplay {
    ...
    public cclass DepartmentNode extends CompositeNode wraps Department {
        public Node getChildAt(int i) {
            return WorkerNode(wrappee.getWorkerAt(i));
        } ...
    }
    public cclass WorkerNode extends Node wraps Employee {
        public String getText() { ... }
    }
    public cclass WorkerNode wraps ExternalEmployee {
        public String getText() {
            return super.getText() + "(" + wrappee.getCompanyName() + ")";
        }
    }
}
```

Listing 12. Wrapper hierarchy for different types of employees

In CaesarJ, wrappers for classes in a hierarchy chain build a hierarchy of related wrappers which share the same name, but are distinguished by the type of the objects they wrap, i.e., they have different `wraps` clauses. All these wrappers share the same constructor which decides which specific wrapper type to create by the dynamic type of the wrappee object passed as a parameter. The general rule of wrapper selection is that the most specific wrapper is selected for the given object. In this way, a wrapper for an object can be retrieved polymorphically.

For illustration, consider the `WorkerNode` wrapper for `ExternalEmployee` in List. 12. It refines the implementation of `getText` so that the display text includes the name of the external company. For an instance of `ExternalEmployee`, the version of `WorkerNode` that wraps `ExternalEmployee` will be used, whereas for an instance of `InternalEmployee`, the `WorkerNode` wrapping `Employee` will be used, because there is no more specific wrapper declared for `InternalEmployee`. The wrapper is retrieved polymorphically in the `getChildAt` method of `DepartmentNode` (List. 12).

Polymorphic usage of wrappers imposes certain typing constraints. The `WorkerNode` wrapper constructor in the method `getChildAt` in List. 12 may return an instance of type `this.WorkerNode`, where the `WorkerNode` is the wrapper class for `Employee` or an instance of `WorkerNode` wrapper for `ExternalEmployee`. To allow the polymorphic usage of wrappers, `WorkerNode` for `ExternalEmployee` must be a subtype of `WorkerNode` for `Employee`. In CaesarJ, this is ensured by implicit inheritance between such wrapper classes, which is why we do not need to declare explicitly `WorkerNode` for `ExternalEmployee` as a subclass of `Node`. The general rule is that subtype relationship between wrappee classes implies inheritance between corresponding wrapper classes. As a consequence, wrappers with the same name build inheritance hierarchies, which reflect the inheritance hierarchies of their wrappees.

Dynamic wrapper selection can be ambiguous in case of multiple inheritance relationships between wrappees. This can occur when wrapping Java interfaces or CaesarJ classes. Consider the case of two wrappers for types `A` and `B`, where

both `A` and `B` are supertypes of the a given wrappee class `W`. If `B` is subtype of `A`, `B`'s wrapper will be selected for `W`. The ambiguous situation occurs when `A` and `B` are not comparable. The ambiguity can be resolved by declaring a wrapper for a supertype of `W` that is a subtype both of `A` and of `B`.

The problem is, however, that detection of such ambiguities cannot be done in a modular way and requires a global analysis of the type system of an application. An analogous problem has been raised in the context of the implementation of external methods in Multijava [10]; the problem is addressed there by disallowing dynamic dispatch on interface types. External methods on interfaces are allowed in Relaxed Multijava [40], but the ambiguities are detected only at class load time. In the current implementation of CaesarJ, ambiguities of dynamic wrapper selection are detected at runtime.

We plan to generalize the dynamic wrapper selection to multiple wrappees (such that there is, e.g., a unique wrapper for a *pair* of wrappees). At the time of writing this paper, the best strategy for dealing with ambiguities and with the inheritance relationships between wrappers (see next subsection) has not yet been fully worked out; hence, this is part of our future work. Note that multiple wrappees can still be used if the programmer defines a usual constructor and takes care of the wrapper management manually.

So far, we have discussed the case of defining wrappers for classes that have an inheritance relationship but are adapted to the same abstraction. In List. 12, both `Employee` and its subclass `ExternalEmployee` are adapted to `Node`. In the general case, given two abstractions `A1` and `A2` pertaining to one concern, where `A2` is a subtype of `A1`, it might be necessary to adopt them to two different abstractions pertaining to another concern B1 and B2, where B2 is a subtype of B1.

Consider for illustration the binding of `IHierarchyDisplay` to the containment hierarchy of GUI elements in a typical Java application in List. 13. In the standard Java library, `Component` is the supertype of all GUI elements, and `Container` is the supertype of GUI elements containing other elements. In order to display a GUI hierarchy, we have to bind `Node` to `Component` and `CompositeNode` to `Container`. Further, since `Container` is subtype of `Component`, their wrappers must belong to the same hierarchy.

So we have a situation, where wrappers of the same hierarchy must implement different interfaces of the collaboration interface. This is possible in CaesarJ, because each wrapper in the hierarchy can introduce new inheritance relationships. For example, in List. 13 the `ComponentNode` wrapper for `Container` inherits from `CompositeNode` besides its implicit inheritance from the wrapper `ComponentNode` for `Component`.

4.3 CaesarJ Bindings versus AspectJ Intertype Declarations

Bindings share with AspectJ's aspects pointcut and advice declarations, but they are based on a different static crosscutting model. CaesarJ aspects use wrappers instead of intertype declarations to add new functionality to existing objects. Bindings support reuse by the techniques of coding to interfaces, virtual

```
abstract public cclass GUIHierarchyBinding extends IHierarchyDisplay {
    protected Component rootComp = null;
    public Node getRoot() { return ComponentNode(rootComp); }

    public cclass ComponentNode extends Node wraps Component {
        public String getText() {
            String name = wrappee.getClass().getName();
            return name.substring(name.lastIndexOf('.') + 1);
        }
    }
    public cclass ComponentNode extends CompositeNode wraps Container {
        public Node getChildAt(int i1) {
            return ComponentNode(wrappee.getComponent(i1));
        }
        public int getChildCount() { return wrappee.getComponentCount(); }
    }
}
```

Listing 13. Binding for containment hierarchy of GUI elements

types, and mixin composition. AspectJ's reuse mechanisms, on the other hand, are limited to abstract aspects and aspect inheritance. In this section, we will discuss the implications of these differences.

Polymorphic aspectual extensions. Wrappers and dynamic wrapper selection allow to define functionality outside a base class (a.k.a open classes [10]), while retaining subtype polymorphism. Wrapper classes define functionality that is polymorphic with respect to both base object types (wrappees) and aspect types.

Consider for illustration List. 10. The method `calculateLayout` belongs to aspect functionality. It is polymorphic with respect to hierarchy node types (each node type has its own `draw` method, which is called by `calculateLayout`). In its control flow, `calculateLayout` eventually calls methods pertaining to data management, such as e.g., `getChildAt` or `getText`. The latter are defined relative to different base types in List. 10, i.e., they are polymorph w.r.t. base types. Each company object has its specific way to access children or to display a text label. For what is more, this specific way is also relative to a particular aspect. That is: It is not only possible to define different ways of accessing children for different company objects within the same aspect; the latter can also be different from aspect to aspect.

As discussed in [38], polymorphic behavior in the extent described above is not possible with intertype declarations. One can only achieve polymorphism w.r.t base types by invasively adding state and methods to base classes directly, however, at the cost of loosing independent extensibility and polymorphism with respect to aspect types [49, 38].

Late-bound operations outside the base model is also the motivation of the visitor design pattern [17]. Both late-bound wrappers and visitors activate their functionality dynamically. Nevertheless, differently from wrappers, visitors require preplanned preparations in the data model and manual implementation of the dispatch code. Furthermore, wrappers give more flexibility for default handling of certain variants of the data model. In List. 12, e.g., we specify default

display behavior for all types of `Employee` and handle only `ExternalEmployee` in a specific way. Finally, a visitor implements the late-binding of a single operation and, therefore, does not support interactions between multiple operations.

Wrappers also allow to associate arbitrary aspect-specific state with base objects; the state can either be defined in the wrapper or inherited from the component class the wrapper binds. As shown in List. 10, wrapper constructors provide a convenient way to navigate from a base (application) object to the corresponding aspect (component) wrapper. An alternative to wrapper constructors in AspectJ would be manual implementation of similar mechanisms for each specific case in the aspect code. Other solutions require modification of application classes to contain links to the corresponding wrapper objects. A more detailed discussion of the problems with managing aspect specific state extensions can also be found in [38].

Abstraction and Reuse. Collaboration interfaces support reuse of the same functionality in different contexts. In Sec. 4.1 and Sec. 4.2 we discussed two reuse scenarios for `HierarchyDisplay`: one with the organizational hierarchy of a company and another with the containment hierarchy of GUI elements. Bindings are also reusable, as long as they are defined to the collaboration interface. We can provide other implementations of hierarchy display for the same collaboration interface. In Sec. 3, we have defined different extensions to `HierarchyDisplay`, which are also alternative implementations of the visualization facet of the same collaboration interface. They can all be reused with our bindings to both the company and GUI structure.

By mixin composition, any display implementation is composable with any binding, as shown in Fig. 13. This is not an accident, but a consequence of proper abstraction. The collaboration interface sets an explicit contract for the composition. It ensures that methods used by the display implementation have the same signature as those implemented by the binding and the other way around. The collaboration interface also unifies the names of the shared classes, making them automatically composable. Any deviation from this contract would be detected by the compiler. The rules for abstract classes help to check correctness of the composition, too. The compiler can ensure that only concrete classes are instantiated and that concrete classes in concrete collaborations are fully implemented.

Once the variability basis (meaningful variations of the display functionality) is set up, CaesarJ developers can easily support different strategies for displaying the company structure, as illustratively shown in Fig. 14, with no additional overhead. The decision as which strategy to use in a concrete situation can be made statically or even dynamically. Here the CaesarJ's deployment mechanism, which will be discussed in the next section, comes into play.

As also discussed in [38], the same degree of reuse and variability is not possible with abstract aspects and intertype declarations of AspectJ. First, the linear inheritance hierarchy is not sufficient for multidimensional reuse, i.e., either the bindings would inherit from concrete display implementations or the other way around. Second, the intertype declarations of an abstract aspect cannot be

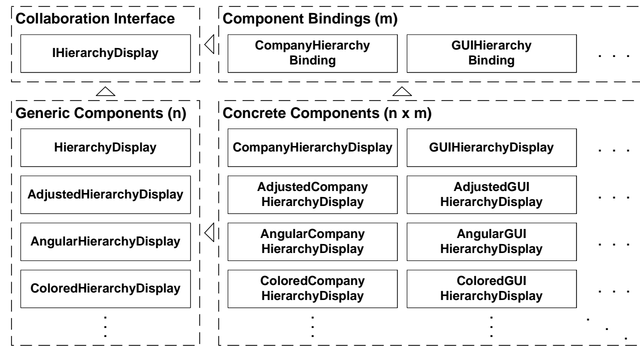

Fig. 13. Composing variations of display implementations and bindings

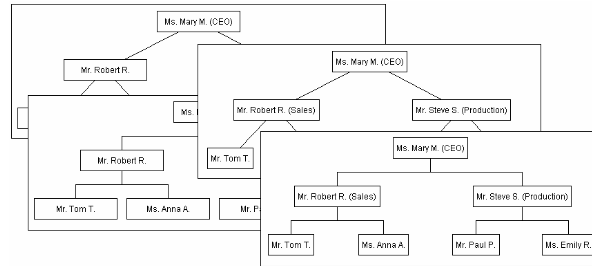

Fig. 14. Company hierarchy display variations

reused in multiple concrete aspects that provide alternative implementations of the introduced operations.

5 Dynamic Aspect Control

We can increase applicability of aspects by enabling flexible control over their activation time and scope. One way to achieve this is to encode the activation logic directly in the aspect. But, this tightly couples the aspect to specific parts of the application and limits its reusability in other contexts. A better solution is is to provide control mechanisms over aspects from outside. In this chapter, we review features of CaesarJ that enable dynamic control over aspects and their scope of activity.

5.1 Explicit Instantiation, Local and Thread-Based Deployment

In CaesarJ an aspect is simply a class containing pointcuts and advice. Like conventional objects aspects in CaesarJ can be instantiated at any point of the program execution using the keyword `new`. There can be multiple instances of an aspect type with independent state, life-cycle, and scope of deployment.

```
public class ShowCompanyHierarchyAction implements ActionListener {
    List companyList; Frame mainWindow;
    ...
    public void actionPerformed(ActionEvent e) {
        CompanyHierarchyDisplay hier = new CompanyHierarchyDisplay();
        hier.setCompany(selectCompanyFromList(companyList));
        HierarchyView view = createNewView(mainWindow);
        view.setHierarchy(hier);
        deploy hier;
    }
}
public class HierarchyView extends JComponent {
    IHierarchyDisplay hierarchy;
    ...
    public void close() { undeploy hierarchy; }
}
```

Listing 14. Lifecycle of company hierarchy display component

Like any other objects, aspects can be referenced, passed as parameters and used polymorphically.

Instantiation does not automatically activate an aspect; the latter must be *deployed* in order to activate its pointcuts and advice. Aspects can be deployed on different dynamic scopes. For simplicity, we will first consider the simplest deployment method called *local deployment*. By means of `deploy`, respectively `undeploy`, statements aspects are deployed, respectively undeployed, on all joinpoints occurring in the local virtual machine process, as illustrated in List. 14 (lines 9 and 15).

Let us illustrate the usefulness of explicit aspect instantiation for the design of the company hierarchy display. The component `CompanyHierarchyDisplay` (List. 11) is an aspect, because it inherits pointcuts and advice from its binding (List. 10). On the other hand, `CompanyHierarchyDisplay` has all the properties of a conventional class. It can be instantiated whenever the application needs to display the company hierarchy. It can be instantiated once more, if the user opens one more view. Furthermore, the decision which concrete variation of the aspect to instantiate may depend on runtime conditions, e.g., user preferences can specify, which of the hierarchy display variations should be used.

In List. 14, `CompanyHierarchyDisplay` is created in a class that handles the menu action to open a hierarchical view of a company. After instantiation we can initialize it with additional data and pass it as a parameter to a view object, which uses its visualization functionality. Once initialized, the aspect is deployed and starts observing changes in the company model[10].

An aspect is garbage collected when it is not referenced anymore and is not deployed. It would be incorrect to garbage collect deployed aspects, because even if they are not explicitly referenced, they are still reachable through joinpoint interception and provide a meaningful functionality just by reacting to certain

[10] We could also deploy the aspect in its constructor, i.e., automatically at creation time; the solution in List. 14 is however safer, because the observation begins only after completing initialization, which establishes necessary application invariants.

```
1 deployed public cclass CompanyDisplayLogging {
2     void around() : execution(* draw(..)) && this(CompanyHierarchyDisplay) {
3         CompanyLogger logger = new CompanyLogger();
4         deploy (logger) { proceed(); }
5     }
6 }
```

Listing 15. Thread-based deployment

```
1 deployed public cclass CompanyLogger {
2     pointcut logMethods() : execution(* company.*.*(..)) || execution(company.*.new(..));
3     before() : traceMethods() {
4         System.out.println(thisJoinPointStaticPart.toString());
5     }
6 }
```

Listing 16. Singleton aspect to trace company model

application events. Aspects can be undeployed explicitly from outside or implicitly as a reaction to some joinpoint. For example, `CompanyHierarchyDisplay` can be undeployed by its client view when the view is closed (List. 14).

By deploying and undeploying aspects at certain points of program execution we control their scope of application. In List. 14, we deploy `CompanyHierarchyDisplay` when its owner view is created and undeploy it, when the view is destroyed. In this way we limit the scope of the aspect application to the lifetime of the corresponding view. We can also restrict the scope of the applicability of the aspect to individual control flows. A similar level of flexibility of aspect control is hard to achieve with static aspect activation as in AspectJ.

The scoping enabled by local deployment is limited only by time of activation. This may not be sufficient in a multi-threaded environment, where we might want to limit the scope of aspects to a single thread. For this purpose, CaesarJ provides *thread-based deployment*, expressed by the `deploy` block. For illustration consider how the aspect in List. 15 is deployed on the scope of the control flow inside the block and does not have any influence on concurrent executions.

Thread based deployment works well for crosscutting of inherently synchronous processes, such as calculations or workflows. However, in event-driven, datacentric environments, we may want to observe updates and events independently of the thread that causes them; in this case, local deployment is more suitable.

5.2 Static Deployment

When an aspect needs to be active all the time, it can be deployed statically. There are two ways to express static deployment. One can declare an aspect class as statically deployed by adding the `deployed` modifier to its declaration. This means that a single instance of the class must be created and deployed at load time. This is useful for implementing singleton aspects. List. 16 demonstrates a singleton aspect `CompanyLogger`, which traces all operations on the company model to the console window.

```
public cclass CompanyLogger extends AbstractLogger { ... }
public cclass Application {
    deployed final private static CompanyLogger compLogger = new CompanyLogger();
    ...
}
```

Listing 17. Static deployment outside the aspect

By declaring the aspect class as statically deployed, we couple its definition with the decision that it will not be used dynamically. The decision can be postponed by expressing it outside the aspect: the `deployed` keyword can also be applied to static fields, declared as `final`. This causes the instance referenced by the field to be automatically deployed at the load time of the enclosing class. List. 17 shows the alternative way to create a statically deployed instance of `CompanyLogger`.

Static deployment is mainly a matter of convenience, it allows to express a special case of dynamic deployment in a more compact way.

5.3 Remote Deployment

Distributed applications open one more dimension for scoping. Process boundaries should not be an obstacle for aspect-oriented interaction techniques. Just as distributed object-oriented applications need to call methods on remote objects, distributed aspect-oriented applications need to intercept remote joinpoints.

In a distributed environment, the company model - instances of `Company`, `Department`, `Employee` - would most probably reside on a server; the display functionality - the `CompanyHierarchyDisplay` together with dependent instances of `Node` and `CompositeNode` - would be on the client. Observation with pointcuts would have to cross the process boundaries: advice would be executed in the context of the hierarchy display component on the client as a reaction to joinpoints of the company model on the server.

Interception of remote joinpoints is enabled by *remote deployment*, which allows to deploy aspects on the scope of remote processes. Remote aspect deployment must be enabled on the server process by calling a special API method `activateAspectDeployment` on an instance of `CaesarHost` initialized with the RMI address identifying the server (List. 18). This creates and publishes an object that accepts aspect deployment requests on the process where `activateAspect-Deployment` is called.

On the client side, aspects can be deployed on the remote process by constructing an instance of `CaesarHost` with the same RMI address and using its `deployAspect` and `undeployAspect` methods. List. 19 shows a modified version of List. 14, which initializes the hierarchy display of a remote company object. We just have to change the way the `Company` object is retrieved and replace local deployment of display component with remote deployment.

Remote aspect deployment is built on top of the Java RMI infrastructure that deals with such issues as remote calls, marshaling of method arguments and

```
public class CompanyServer {
    public static void main(String[] args) {
        ...
        CaesarHost host = new CaesarHost("rmi://myserver.net/MyServer/");
        host.activateAspectDeployment();
        ...
    }
}
```

Listing 18. Server process hosting company model

```
public class ShowCompanyHierarchyAction implements ActionListener {
    CaesarHost host = new CaesarHost("rmi://myserver.net/MyServer/");
    ...
    public void actionPerformed(ActionEvent e) {
        try {
            CompanyHierarchyDisplay hier = new CompanyHierarchyDisplay();
            hier.setCompany((Company)host.resolve("Company"));
            HierarchyView view = createNewView(mainWindow);
            view.setHierarchy(hier);
            host.deployAspect(hier);
        }
        catch (CaesarRemoteException e) { System.out.println(e.getMessage()); }
    }
}
```

Listing 19. Initializing display of remote company model

management of remote references. Additionally, we provide a tool that generates stubs for CaesarJ classes that must be executed for each CaesarJ class that is used or deployed remotely.

5.4 Deployment on a Distributed Control Flow

In Sec. 5.1, we argued that the aspects observing processes need to be deployed on single threads. In List. 15, the aspect `CompanyLogger` was deployed inside the execution of the method `draw` of the company display component to monitor how the display uses the data model. Such a solution fails in a distributed environment, where the display component is working on the client side, but the data model is located on the server.

On the other hand, the remote deployment, described in Sec. 5.3 enables observation of the data model activity on the server process, but it does not distinguish between requests from different clients. So, if we deploy `Company-Logger` using remote deployment method, it will monitor all the activity on the data model during its deployment period. However, we need to intercept only the joinpoints on the server side that are in the control flow of the `draw` method including the synchronous remote calls.

The necessary filtering is provided by another deployment method supported in CaesarJ - *deployment on a distributed control flow*. This deployment method is expressed by API calls, as shown in List. 20. The aspect affects the synchronous control flow, which the current thread is part of. The synchronous control flow may involve multiple threads from different processes, which interact through

```
1 deployed public cclass CompanyDisplayLogging {
2     void around() : execution(* draw(..)) && this(CompanyHierarchyDisplay) {
3         CompanyLogger logger = new CompanyLogger();
4         RemoteDeployment.deployOnControlFlow(logger);
5         proceed();
6         RemoteDeployment.undeployFromControlFlow(logger);
7     }
8 }
```

Listing 20. Deployment on distributed control flow

synchronous calls. In this way, we can filter the joinpoints of company model methods, which were requested by `draw` functionality.

6 Implementation

In this section, we discuss the implementation of CaesarJ on top of the Java Virtual Machine by presenting the steps performed by the CaesarJ compiler.

6.1 Implementation of Virtual Classes

By explicitly redefining virtual classes in a sub-family, we potentially also introduce implicit (inherited) types and relations. Let us consider List. 1 for illustration. Although the virtual type `CompositeNode` is not explicitly declared in `AdjustedHierarchyDisplay`, there is an *implicit* virtual type `AdjustedHierarchyDisplay.CompositeNode` (dashed rectangles in Fig. 15). Furthermore, there are a number of implicit relations, namely the relations inherited from the super-family and the subtype relations between different refinements of a virtual class (dashed inheritance arrows in Fig. 15).

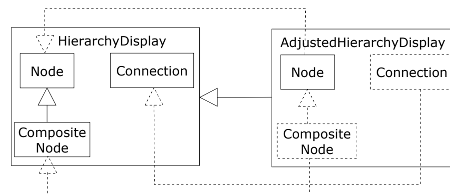

Fig. 15. Implicit types and relations

In CaesarJ, all available virtual types are generated at compile-time as Java classes. Hence, the first step in the compilation process is the calculation of the *type graph* by extending the explicit (in source code) declared structure with the information about the implicit types and relations. For example, all boxes and relations in Fig. 15 (explicit and implicit ones) constitute the type graph of List. 1. Note that the `cclass` interface hierarchy directly reflects the structure of the type graph.

Next, for each CaesarJ type contained in the type graph, the compiler generates a corresponding mixin list with the algorithm already discussed in Sec. 3.2.

Finally, the generated mixin lists are transformed to Java language constructs. This is achieved as follows. For each `cclass` declaration, the CaesarJ compiler generates a Java interface, called the *cclass interface* in the following discussion, and a Java class, called the *cclass implementation* below. This results in an *interface* and an *implementation hierarchy.* For example, Fig. 16 shows separated implementation and inheritance hierarchies for the set of classes defined in the framed listing. The interfaces have the same name as the `cclass` declarations in the source code. A `_Impl`-suffix is appended to the names of implementation classes.

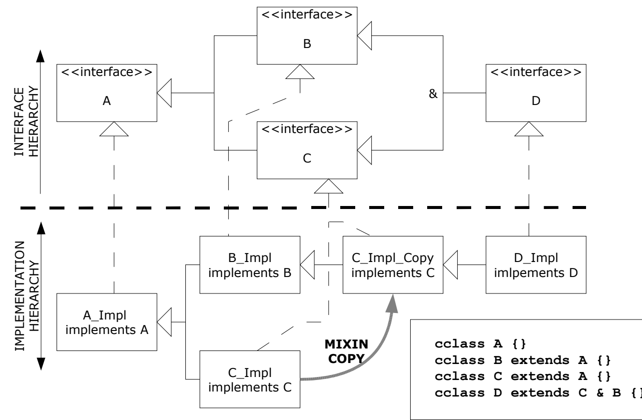

Fig. 16. Separated interface and implementation hierarchies

To construct the implementation hierarchy, every `cclass` is conceptually viewed as a mixin whose super-class parameter is restricted to the declared parent or a subclass of it. The `cclass` implementations are gained from a composition of an ordered mixin list. For example, the mixin list of `B_Impl` is `[A, B]`.

Given the set of mixin lists for a set of CaesarJ classes, the compiler constructs the single inheritance implementation hierarchy using plain Java classes. Nodes in the resulting inheritance hierarchy are `cclass` implementations; each path in it, starting with the root node, corresponds to a mixin list. Sometimes, mixins need to be duplicated and the declared parent of a mixin has to be changed. For example, the mixin list of `D_Impl` is `[A, B, C, D]`. That is, the super of `C`, which is `A` by default, needs to be replaced with `B`.

To generate the corresponding path in the implementation hierarchy, the compiler generates a new Java class by cloning the bytecode of the original mixin, then it replaces in the cloned bytecode all occurrences of the old super-class references with the new one. In our example, `C_Impl_Copy` is the clone of `C_Impl`, having the references to the old super-class replaced by `B_Impl`.

```
1  class N extends M { }
2  class O extends N { }
3  class P extends O { }
4
5  cclass CollabA {
6      cclass A wraps M { ... }
7      cclass A extends B wraps O { ... }
8      cclass B { ... }
9  }
10 cclass CollabB extends CollabA {
11     cclass A wraps M { ... }
12     cclass A wraps P { ... }
13 }
```

Fig. 17. Code with wrapper classes

```
1  cclass CollabA {
2      cclass A_M { ... }
3      cclass A_O extends B & A_M { ... }
4      cclass B { ... }
5      public A_M newAforM(M x) {
6          if (x instanceof O) {
7              return newAforO((O)x);
8          }
9          return new A_M(x);
10     }
11     public A_O newAforO(O x) {
12         return new A_O(x);
13     }
14 }
15 cclass CollabB extends CollabA {
16     cclass A_M { ... }
17     cclass A_P extends A_O { ... }
18     public A_O newAforO(O x) {
19         if (x instanceof P) {
20             return newAforP((P)x);
21         }
22         return new A_O(x);
23     }
24     public A_P newAforP(P x) {
25         return new A_P(x);
26     }
27 }
```

Fig. 18. Generated code

The interface hierarchy hides the implementation hierarchy. It contains the public methods of the `cclass` implementations and represents their subtype relations. E.g., the interface `D` is a subtype of `B` and `C`. Since the implementation class always implements the corresponding `cclass` interface, we can preserve type compatibility by working with `cclass` interfaces, e.g., we can assign `D_Impl` to `B` and `C`.

6.2 Implementation of Wrappers

Wrapper recycling is managed by family objects via a map from wrappees to wrappers for each hierarchy of wrappers with the same name. When a wrapper constructor is called, a wrapper is retrieved from a hash table using the wrappee as a key. If the wrapper for given wrappee is not available, a new most-specific wrapper for given wrappee is created and registered in the hash table.

Wrapper classes are translated to virtual classes that are identified by the pairs of wrapper and wrappee names as shown in List. 17 and List. 18. The example shows that wrappers can be overridden in the sub-family by declaring a new wrapper with the same name and for the same wrappee class. The inheritance relationships between the translated wrappers are generated according the subtype relationships of their wrappee classes. In the next compilation steps wrappers are treated in the same way as described in Sec. 6.1 for the simple virtual classes.

List. 18 also shows methods generated for the creation of the most specific wrapper for the given wrappee. Such methods are generated for each wrapper class. Each method checks if the direct subclasses of the wrapper class can be applied for the given wrappee: if yes, the selection is delegated to the more specific method; if not, the current wrapper class is instantiated. Note, that the instantiation is polymorphic, which ensures that the most specific versions of the wrappers will be instantiated. The selection methods must be overridden when the set of the direct subclasses of the wrapper is changed, e.g., `newAforO` is overridden in List. 18, because a new subclass of `A_O` was defined.

Another important implementation issue is the life-cycle management of wrappers. The life-cycle of a wrapper is coupled to the life-cycle of two objects, the family and the wrappee. In the current implementation, families use a standard Java hash map to implement the mapping from wrappees to wrappers. This means that entries in the map are garbage-collected only when the corresponding family object is garbage-collected. A better solution would be to release a mapping as soon as neither the wrappee nor the wrapper are reachable. Weak references sound like a solution at first glance but it turned out that the support for weak references as currently implemented in the JVM is not sufficient to implement this strategy.

6.3 Implementation of Dynamic and Remote Deployment

In this section, we sketch the implementation of aspect deployment in CaesarJ. The aspect deployment framework builds upon static aspects. Each crosscutting class is split into two classes: an *implementation class* and a *registry class*. The implementation class encapsulates the behavior of the aspect objects, while the registry class manages their deployment. Pieces of advice, declared in the aspect, are translated to methods in the implementation class, while the pointcuts are copied to the registry class.

The registry is a singleton aspect, which is statically woven using the AspectJ weaver [26]. The methods of the registry can be seen as statically woven hooks, which are responsible to dispatch calls to the corresponding methods of the deployed instances of the implementation class. The singleton registry instance maintains a container of the deployed instances of the implementation class, as shown in Fig. 19.

Aspect containers in Fig. 19 decide on the deployed objects that must be called at a certain joinpoint. Different deployment methods use different types of containers. The container for local deployment notifies all objects it manages. The thread-based deployment strategy, on the other hand, uses a map-based container that notifies only the objects that are deployed on the current thread. Simultaneous use of multiple deployment strategies is supported by using a composite container, which aggregates the aspect containers of multiple deployment methods. The relationship between the implementation and the registry classes is not one to one. The compiler analyzes inheritance relationships between aspect classes, which may involve mixin composition, and generates shared registry classes for aspects with identical crosscutting behavior.

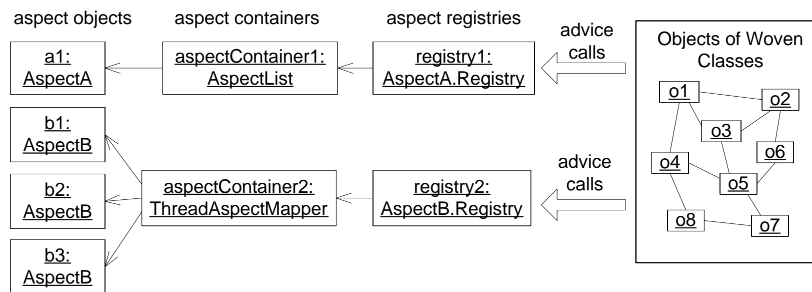

Fig. 19. Sample runtime configuration with dynamically deployed aspects

The implementation of dynamic deployment has been carried out with care about performance. The weaver inserts advice calls only at joinpoints which are referenced by the aspects in the application. If no aspect is deployed at a joinpoint, the dispatch logic causes one redundant static method call and one field check for the null value. When aspects are deployed, there is only one additional virtual method call as compared to AspectJ. The reader interested in a more details on aspect registries is referred to [18].

Remote Deployment. Remote aspect deployment in CaesarJ uses Java RMI, which generates stub classes for transparent communication with remote objects. Stubs must also be generated for aspects that are remotely deployed. When an aspect object is deployed remotely, a stub is created for this object on the remote process. The stub intercepts joinpoints on the remote process and marshals the advice calls to the real aspect object. The stubs are generated by a specialized RMI compiler for CaesarJ classes. Classes, which are used or deployed remotely, must be prepared by this tool. Differently from standard Java RMI, the CaesarJ RMI compiler does not require specially prepared remote interfaces. The stub can be generated for any CaesarJ class.

Deployment on Distributed Control Flow. Each synchronous control flow is represented by a single thread in each involved process; therefore, the aspects deployed on the control flow are actually deployed on these threads. During remote call the client process must send the aspects deployed within the current thread, and the remote process must again deploy the received aspects on the tread that serves the client request.

To send aspects to another process, the marshaling of remote method calls has been modified. Normally the stub of a remote object marshals the reference to the object, the name of the called method and the arguments. In the modified version, the stub additionally sends the references to the aspects, which are deployed on the current control flow. The remote process unpacks the received references to the aspects and deploys them on the corresponding thread.

7 Related Work

We have divided work similar to the paper itself into three different groups: Hierarchical composition, crosscutting composition, and dynamic aspect control.

7.1 Hierarchical Composition

Virtual classes were originally introduced in *BETA* [34] and were further developed in gbeta [12], which supplemented them with mixin composition and family polymorphism. CaesarJ provides a solid implementation of these concepts on the JVM and combines them with language features for crosscutting composition.

Jx [42] supports a kind of nested inheritance. A major difference is that Jx considers inner classes not as properties of the enclosing object, but as properties of the surrounding class. Applicability of Jx is limited to linear refinements, because it does not provide any composition mechanisms for family classes. A similar linear refinement of classes is also supported in Keris [54], but as extension technique for static modules rather than for instantiable family classes.

AHEAD [4] is the newest technology based on ideas of GenVoca [5] and Mixin Layers [48]. AHEAD supports modularization of application features in large-scale units called layers, which are sets of files describing fragments of different artifacts of the application including fragments of Java classes. The layers are composed using a mixin composition technique that is similar to the one of CaesarJ. The provided implementation is based on source-to-source transformation. Layers lack subtyping and abstraction capability. In CaesarJ, abstractions play an important role for ensuring validity of individual family classes and their composition. In AHEAD reliable composition of layers is assured by additional specifications. Differently from CaesarJ family classes, layers cannot be instantiated and used polymorphically.

Virtual classes are composed along two dimensions: at first the mixins of the same virtual class are composed, and then such mixin list is composed with analogous mixin lists of its superclasses. Traits [6] also support the composition of the classes along two dimensions: composing the traits inside a class and then along the inheritance hierarchy. Composition of traits inside a class is an orthogonal dimension w.r.t. the both composition dimensions of virtual classes. Traits in combination with virtual classes would mean that virtual classes could be extended with new traits in their further bindings.

7.2 Crosscutting Composition

Hölzle analyzed the problem of integrating independent components in object-oriented languages [24]. Our work addresses many problems identified by Hölzle. CaesarJ is also related to Hyper/J [51] and the notion of multi-dimensional separation of concerns (MDSOC) [52]. In order to avoid the "tyranny of the dominant decomposition" CaesarJ is not limited to hierarchical refinement

and composition techniques, but supports development of multiple independent hierarchies and their crosscutting composition by means of bindings.

However, on the technical level CaesarJ is very different from Hyper/J. In Hyper/J, one can define an independent component in a hyperslice. A hyperslice is integrated into an existing application by means of composition rules, specified in a hypermodule. Hyperslices are independent of their context of use, because they are declaratively complete, i.e. they declare as abstract methods everything that they need, but cannot implement themselves. This is different from the CaesarJ approach of shared abstractions in form of a collaboration interface, which facilitates reliable composition and makes the composition code itself reusable. The composition mechanisms in Hyper/J are class based and cannot be applied in a dynamic way like CaesarJ bindings. Furthermore, Hyper/J's sublanguage for mapping specifications from different hyperslices is fairly complex and not well integrated into the common OO framework.

Integration of multi-abstraction of components was addressed by the predecessor technologies of CaesarJ: Adaptive Plug and Play Components (APPCs) [36], Aspectual Components (AC) [31] and Pluggable Composite Adapters (PCA) [39]. Due to lack of necessary abstraction capabilities, connectors and adapters in APPC, AC, and PCA models are bound to a fixed implementation of an aspect and cannot be reused. CaesarJ also extends these technologies with mechanisms for layered refinement and composition.

The idea of collaboration based design and composition with bindings is also implemented in ObjectTeams [22]. The notion of a team is analogous to our family classes, and the roles inside teams have similar semantics as virtual classes. This enables linear refinement of teams and separation of generic team implementation from its concrete binding to application classes. However, similarly to APPC, AC and PCA, ObjectTeams does not support collaboration interfaces and reuse of bindings. Besides, the crosscutting capabilities supported in the form of call-ins are significantly less expressive than the pointcuts supported by CaesarJ.

Framed Aspects [33], Sally [19], and LogicAJ [28] provide a form of genericity for AspectJ-like aspects. In this way an aspect is more reusable, because it can be bound to various application classes by specifying different generic parameters. In CaesarJ, we assume that bindings to different classes are different and may require totally different adaptation code. Nevertheless, CaesarJ can benefit from generic pointcuts for better reuse of similar bindings.

7.3 Dynamic Aspect Control

Method Call Interception (MCI)[29] offers dynamically deployed joinpoint interception on the basis of source code instrumentation. Their idea to use central registry to control execution of explicitly instantiated and deployed advice objects is similar to our local dynamic deployment mechanism outlined in Listing 14. In comparison to the implementation of MCI [30], CaesarJ provides additional optimizations by creating a specialized registry for each type of aspect and weaves it only at the joinpoints, referenced by the pointcuts of the aspect.

CaesarJ and most of other dynamic aspect activation approaches, such as EAOP[11], JAC[45], PROSE[46], JBoss AOP[25] and AspectWerkz[8], require one or another form of pre-runtime class preparation for weaving. The classes are either prepared at compile time, at class load time or during just-in-time compilation. There are two possibilities for pre-runtime class preparation: either to insert hooks at all joinpoints of a loaded class or to limit to a fixed set of known join points. While the first option causes significant performance overhead, the second option (also used in CaesarJ) assumes initial knowledge about aspects, which will be activated.

Dynamic aspect deployment can be more efficiently implemented on the systems supporting real run-time weaving, such as Steamloom [7] and AspectS [23]. Steamloom is particularly well-suited for the needs of aspect deployment in CaesarJ, because it supports thread local aspects as well as aspect deployment on individual objects.

Remote pointcuts introduced in DJcutter [41] allow definition of aspects which refer to joinpoints on remote processes. All aspects run on a special aspect server and intercept joinpoints in all or some of the registered hosts. CaesarJ extends applicability of remote joinpoints, by combining them with dynamic aspect deployment. This enables dynamic selection of servers as well as connecting and disconnecting from the server at any point of program execution. Besides, dynamic deployment postpones the decision about local or remote usage of an aspect until runtime.

The idea of unifying aspects and classes, i.e. support for stateful aspects and their explicit instantiation, has also recently been implemented in the Eos-U language [47]. In Eos-U, the advice overriding problem is solved by completely replacing advice with methods. A similar effect can be achieved in CaesarJ by inserting method calls on self in the advice bodies.

8 Summary and Future Work

In this paper we gave an overview of the CaesarJ programming language. We demonstrated that advanced object-oriented techniques for multi-class components and interaction based on joinpoint-interception are complementary technologies, which can be used together to solve important software design problems.

In Sec. 3, we showed that by treating collaborations as classes, we can apply object-oriented techniques on a larger scale. Virtual classes and propagating mixin composition provide a means for abstraction, refinement and polymorphism of multi-class components, but they are not sufficient for integration of independently developed components with different modular structure. The problem of crosscutting integration of structure and behavior can be solved by the mechanisms for join-point interception and dynamic object extensions in form of wrappers. The unification of aspects and collaborations facilitates development of reusable well-modularized aspects, as was explained in Sec. 4. Finally,

Sec. 5 demonstrated that treating an aspect as a class enables its free instantiation and flexible control over its scope of application.

The current implementation of the CaesarJ compiler covers all the features presented in this paper except the dynamic wrapper selection, which is part of our on-going work. The existing implementation is stable and is already used in case studies of our industrial partners. We also provide an IDE for the language in the form of an Eclipse plug-in, which, among other features, includes views for visualization of CaesarJ virtual classes and crosscutting structure. The compiler, the Eclipse plug-in, the language reference as well as other documentation are available from `caesarj.org`.

There are several areas of ongoing and future work. We are investigating ways to provide a better support for CaesarJ language features on the virtual machine level. So far, we have been building support for shadow search, weaving, and dynamic aspect deployment into the aspect-oriented virtual machine Steamloom [21]. However, CaesarJ is not built on top of Steamloom so far. We believe that implementing CaesarJ compiler on top of Steamloom could significantly improve the implementation, but JVM compatibility would have to be sacrificed.

Also, we plan to add support for the virtual classes directly at the virtual machine level. This will significantly facilitate the implementation of the compiler and will avoid a lot of code duplication generated right now by the compiler to simulate the virtual class semantic on top of standard Java classes. Another path in our future research will be concerned with bringing into CaesarJ a more powerful pointcut language such as the one supported by the prototype language presented in [44].

Yet other threads of future work will be concerned with the module system of CaesarJ. One interesting issue to consider is to support a more flexible bundling of classes into families. Right now, related virtual classes have to be defined within a module. This might be too restrictive. We would like to be able to bundle classes that are defined independently into families. This imposes hard challenges on static typing, which need to be considered carefully. Another issue to investigate is the relation between CaesarJ modules, generic components and crosscutting bindings, to genericity. We believe that CaesarJ modules, equipped with some extensions such as so-called *final bindings* are able to simulate generics properly, but this needs to be investigated more in the future.

Acknowledgments

This work is partly supported by TOPPrax Project sponsored by the German Ministry of Education and Science (BMBF) and the European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe) sponsored by the EU FP6.

References

[1] J. Aldrich. Open modules: Modular reasoning in aspect-oriented programming. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL) at AOSD'04*, 2004.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[3] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A monotonic superclass linearization for Dylan. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 69–82. ACM Press, 1996.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[5] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genvoca model of software-system generators. *IEEE Softw.*, 11(5):89–94, 1994.

[6] A. P. Black and N. Scharli. Traits: Tools and methodology. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 676–686, Washington, DC, USA, 2004. IEEE Computer Society.

[7] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[8] J. Boner. Aspectwerkz. http://aspectwerkz.codehaus.org/index.html.

[9] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[10] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. *SIGPLAN Not.*, 35(10):130–145, 2000.

[11] R. Douence and M. Südholt. A model and a tool for event-based aspect-oriented programming. Technical Report Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

[12] E. Ernst. *gbeta - a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[13] E. Ernst. Propagating class and method combination. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 67–91, London, UK, 1999. Springer-Verlag.

[14] E. Ernst. Family polymorphism. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 303–326, London, UK, 2001. Springer-Verlag.

[15] E. Ernst. Higher-order hierarchies. In L. Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

[16] E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. In *33rd ACM Symposium on Principles of Programming Languages (POPL'06)*. ACM SIGPLAN-SIGACT, to appear, 2006.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[18] J. Hallpap. Towards Caesar: Dynamic deployment and aspectual polymorphism. Master's thesis, Department of Computer Science, Darmstadt University of Technology, 2003. http://www.st.informatik.tu-darmstadt.de/database/theses/thesis/DiplomaThesis.pdf?id=15.

[19] S. Hanenberg and R. Unland. Parametric introductions. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 80–89, New York, NY, USA, 2003. ACM Press.

[20] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[21] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An execution layer for aspect-oriented programming languages. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 142–152, New York, NY, USA, 2005. ACM Press.

[22] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 248–264, London, UK, 2003. Springer-Verlag.

[23] R. Hirschfeld. AspectS - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[24] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 36–56, London, UK, 1993. Springer-Verlag.

[25] JBoss Inc. JBoss aop beta3. http://www.jboss.org, 2004.

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[27] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[28] G. Kniesel, T. Rho, and S. Hanenberg. Evolvable pattern implementations need generic aspects. In *RAM-SE*, pages 111–126, 2004.

[29] R. Lämmel. A semantical approach to method-call interception. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 41–55, New York, NY, USA, 2002. ACM Press.

[30] R. Lämmel and C. Stenzel. Semantics-Directed Implementation of Method-Call Interception. 46 pages; Accepted for publication in IEE Proceedings Software; Special Issue on Unanticipated Software Evolution, Nov. 2003.

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

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

[33] N. Loughran and A. Rashid. Framed aspects: Supporting variability and configurability for aop. In J. Bosch and C. Krueger, editors, *International Conference*

on Software Reuse, Madrid, Spain, volume LNCS 3107, pages 127–140. Springer, 2004.

[34] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM Press.

[35] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[36] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 97–116, New York, NY, USA, 1998. ACM Press.

[37] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 52–67, New York, NY, USA, 2002. ACM Press.

[38] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[39] 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, 2000. University of Twente, The Netherlands.

[40] T. Millstein, M. Reay, and C. Chambers. Relaxed multijava: balancing extensibility and modular typechecking. *SIGPLAN Not.*, 38(11):224–240, 2003.

[41] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed aop. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15, New York, NY, USA, 2004. ACM Press.

[42] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. *SIGPLAN Not.*, 39(10):99–115, 2004.

[43] K. Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 89–110, London, UK, 2002. Springer-Verlag.

[44] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP'05: European Conference on Object-Oriented Programming*. Springer LNCS, 2005.

[45] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings REFLECTION '01, LNCS 2192*, pages 1–24, 2001.

[46] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[47] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[48] Y. Smaragdakis and D. S. Batory. Implementing layered designs with mixin layers. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570, London, UK, 1998. Springer-Verlag.

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

[50] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[51] P. Tarr and H. Ossher. Hyper/J user and installation manual, 1999. http://www.research.ibm.com/hyperspace.

[52] 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.

[53] M. VanHilst and D. Notkin. Using role components in implement collaboration-based designs. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 359–369, New York, NY, USA, 1996. ACM Press.

[54] M. Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.