Modules for Crosscutting Models

Mira Mezini and Klaus Ostermann

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

Abstract. Traditional programming languages assume that real-world systems have "intuitive", mind-independent, preexisting concept hierarchies. However, our perception of the world depends heavily on the context from which it is viewed: Every software system can be viewed from multiple different perspectives, and each of these perspectives may imply a different decomposition of the concerns. The hierarchy which we choose to decompose our software system into modules is to a large degree arbitrary, although it has a big influence on the software engineering properties of the software. We identify this *arbitrariness of the decomposition hierarchy* as the main cause of 'code tangling' and present a new model called CAESAR[1], within which it is possible to have multiple different decompositions *simultaneously* and to add new decompositions on-demand.

1 Introduction

The criteria which we choose to decompose software systems into modules has significant impact on the software engineering properties of the software. In [14] Parnas observed that a data-centric decomposition eases changes in the representation of data structures and algorithms operating on them. Following on Parnas work, Garlan et al. [2] argue that function-centric decomposition on the other side better supports adding new features to the system, a change which they show to be difficult with the data-centric decomposition.

Software decomposition techniques so far, including object-oriented decomposition, are weak at supporting multi-view decomposition, i.e., the ability to simultaneously breakdown the system into inter-related units, whereby each breakdown is guided by independent criteria. What current decomposition technology does well is to allow us to view the system at different abstraction levels, resulting in several *hierarchical* models of it, with each model be a refined version of its predecessor in the abstraction levels.

By multi-view decomposition, we mean support for simultaneous *crosscutting* rather than *hierarchical* models. The key point is that our perception of the world depends heavily on the perspective from which we look at it: Each perspective may imply a different decomposition of the concerns. In general, these view-specific decompositions are equally reasonable, none of them being a

[1] Project homepage at www.st.informatik.tu-darmstadt.de/pages/projects/caesar

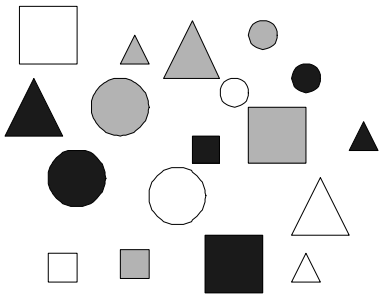

Fig. 1. Abstract concern space

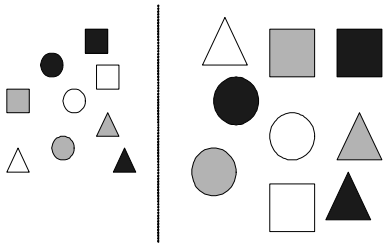

Fig. 2. Divide by size

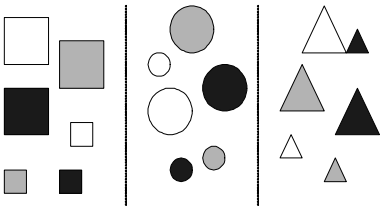

Fig. 3. Divide by shape

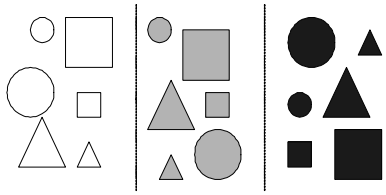

Fig. 4. Divide by color

sub-ordinate of the others, and the overall definition of the system results from a superimposition of them.

One of the key observations of the aspect-oriented software development is that a programming technique that does not support simultaneous decomposition of systems along different criteria suffers from what we call *arbitrariness of the decomposition hierarchy* problem, which manifests itself as tangling and scattering of code in the resulting software, with known impacts on maintainability and extendibility. In Fig. 1 to 5 we schematically give an idea of the problem at a non-technical level. Assuming the domain symbolically represented in Fig. 1 is to modeled, we can decompose according to three criteria, size, shape, and color, represented in Fig. 2, Fig. 3, and Fig. 4 respectively, whereby each of these decompositions is equally reasonable.

With a 'single-minded' decomposition technique that supports only hierarchical models, we have to choose one fixed classification sequence, as e.g., *color* → *shape* → *size* illustrated in Fig. 5. However, the problem is that with a fixed classification sequence, only the first element of the list is localized whereas all other concerns are tangled in the resulting hierarchical structure. Fig. 5 illustrates this by the example of the concern 'circle', whose definition is scattered around several places in the hierarchical model. Only the color concern is cleanly separated into white, grey and black, but even this decomposition is not satis-

factory because the color concern is still tangled with other concerns: "white" is e.g., defined in terms of white circles, white rectangles and white triangles.

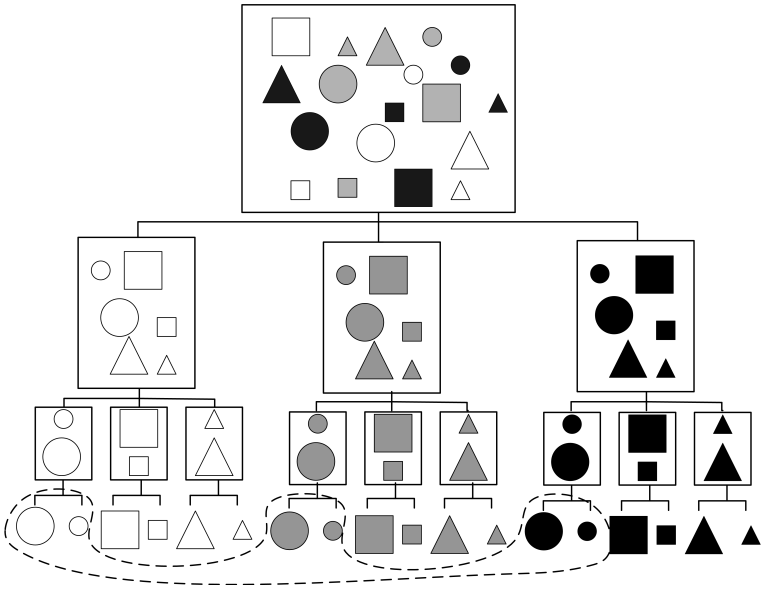

Fig. 5. Arbitrariness of the decomposition hierarchy

The problem is that models resulting from simultaneous decomposition of the system according to different criteria are in general "crosscutting" with respect to the execution of the system resulting from their composition. With the conceptual framework introduced so far, *crosscutting* can be defined as a relation between two models with respect to the abstract concern space. This relation if defined via *projections* of models (hierarchies). A projection of a model M is a partition of the concern space into subsets $o_1, \ldots, o_n$ such that each subset o_i corresponds to a leaf in the model. For illustration Fig. 6 shows a projection of the `color` model from Fig. 4 onto the abstract concern space of Fig. 1.

Now, two models, M and M', are said to be crosscutting, if there exist at least two sets o and o' from their respective projections, such that, $o \cap\ o'$, and neither $o \subseteq o'$, nor $o' \subseteq o1$. Fig. 7 illustrates how the `black` concern of the `color` model (Fig. 4) crosscuts the `big` concern of the `size` model (Fig. 2). These two concerns have in common the big, black shapes, but neither is a subset of the other: the black module contains also small black shapes, while the size model contains also non-black big shapes.

On the contrary, a model M is a *hierarchical refinement* of a model M' if their projections $o_1, \ldots, o_n$ and $o'_1, \ldots, o'_m$ are in a subset relation to each other as follows: there is a mapping $p : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$ such that $\forall i \in$

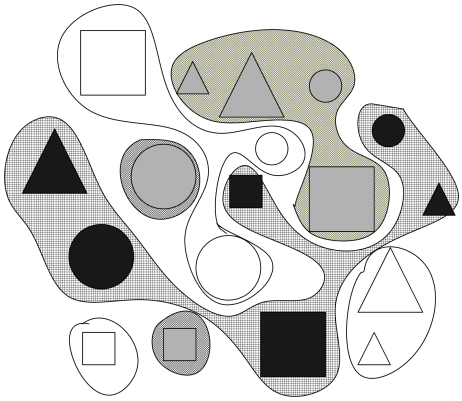

Fig. 6. Projection of the 'color' decomposition

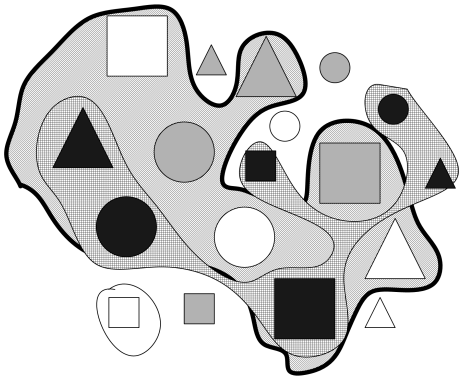

Fig. 7. Crosscutting Hierarchies

$\{1, \ldots, n\} : o_i \subseteq o'_{p(i)}$. Crosscutting models are themselves not the problem, since they are inherent in the domains we model. The problem is that our languages and decomposition techniques do not (properly) support crosscutting modularity (see the discussion on decomposition arbitrariness above).

Motivated by these observations, we are working on a language model, called CAESAR, with explicit module support for crosscutting models. This paper puts special emphasis on the conceptual rationale and motivation behind CAESAR. It leaves out many technical details and features that are not of interest for the first-time reader. For more technical details we refer to two recent conference papers [11, 12]. For instance, in [11] we describe how CIs, CI bindings, and CI implementations can be specified incrementally by an appropriate generalization of inheritance. In [12] we enrich CAESAR with means for describing *join points* (points in the execution of a base program to intercept) and *advices* (how to

react at these points) and describe a new idea called *aspectual polymorphism*, with which components/aspects can be deployed polymorphically.

The remainder of this paper is structured as follows: In Sec. 2 we will discuss a concrete example that illustrates shortcomings of current language technology with respect to crosscutting hierarchies. In Sec. 3 we will shortly present our solution to cope with the identified problems. Sec. 4 is a wrap-up of what we have gained with the new technology. Sec. 5 discusses related work. Sec. 6 summarizes the paper.

2 Problem Statement

In this section, we identify shortcomings of current language technology with respect to supporting crosscutting models. We will use the Java programming language to illustrate shortcomings of current language technology, but the results are applicable to other module-based and object-oriented languages as well.

We introduce a simple example, involving two view-specific models of the same system: the GUI perspective versus the data types perspective. Fig. 8 shows a simplified version of the `TreeModel` interface in Swing[2], Java's GUI framework [4]. This interface provides a generic description of data abstractions that can be viewed and, hence, displayed as trees. Fig. 8 also presents a interface for tree GUI controls in `TreeGUIControl`, as well as an implementation of this interface in `SimpleTreeDisplay` (the latter roughly corresponds to `JTree`).

In our terminology the code in Fig. 8 defines a GUI-specific model of any system that supports the display of arbitrary data structures that are organized as trees. When this model is used in a concrete context, e.g., in the context of a system that manipulates arithmetic expression object structures, as schematically represented in Fig. 9, it provides to this context the `display()` functionality. In turn, it expects a concrete implementation of `getChildren(...)` and `getStringValue(...)`.

Please note how the GUI-specific model crosscuts a model resulting from a data-centric decomposition of the system that manipulates expressions, and/or other aggregate, tree-structured data types. This is exemplified in Fig. 9, which shows sample code for a data-centric decomposition of arithmetic expressions. In a running system, resulting from a composition of the GUI-specific and data-specific models, the tree module from the GUI-specific model and the expression module from the data type model intersect, since there will be display-enabled expression objects. However, none of them is a subset of the other: There might be non-displayable expression objects, as well as displayable tree-structured data type objects that are not expressions.

2.1 Lack of Appropriate Module Constructs

The need for crosscutting modularity calls for module constructs that capture individual crosscutting models of the system by encapsulating the definition

[2] Swing separates our interface into two interfaces, `TreeModel` and `TreeCellRenderer`. However, this is irrelevant for the reasoning in this model.

```
interface TreeModel {
  Object getRoot();
  Object[] getChildren(Object node);
  String getStringValue(Object node, boolean selected,
    boolean expanded, boolean leaf, int row, boolean focus);
  }
}

interface TreeGUIControl {
   display();
}

class SimpleTreeDisplay implements TreeGUIControl {
   TreeModel tm;
   display() {
      Object root = tm.getRoot();
      ... tm.getChildren(root) ...
      ...
      // prepare parameters for getStringValue
      ... tm.getStringtValue(...);
      ...
   }
}
```

Fig. 8. Simplified version of the Java Swing `TreeModel` interface

of multiple inter-dependent abstractions. In addition, mechanisms are needed to allow such modules to be self-contained (closed), while at the same time be opened to be composed with other crosscutting modules, while remaining decoupled from them. We argue that common module constructs in object-oriented languages and especially the common concept of interfaces as we know it, e.g., from Java, lack these two features.

Lack of support for declaring a set of mutually recursive types.

Defining generic models involves in general several related abstractions. We claim that current technology falls short in providing appropriate means to express the different abstractions and their respective features and requirements that are involved in a particular collaboration. Let us analyze the model in Fig. 8 from this point of view. The first "bad smell" is the frequent occurrence of the type `Object`. We know that a tree abstraction is defined in terms of smaller tree node abstractions. However, this collaboration of the tree and tree node abstractions is not made explicit in the interface. Since the interface does not state anything about the definition of tree nodes, it has to use the type `Object` for nodes.

```
class Expression {
  Expression[] subExpressions;
  String description() { ... }
  Expression[] getSubExpressions() { ... }
}
class Plus extends Expression { ... }
```

Fig. 9. Expression Trees

The disadvantages of using the most general type, `Object`, are twofold. First, it is conceptually questionable. If every abstraction that is involved in the GUI model definition is only known as `Object`, no messages, beside those defined in `Object`, can be directly called on those abstractions. Instead, a respective top-level interface method has to be defined, whose first parameter is the receiver in question. For example, the methods `getChildren(...)` and `getStringValue(...)` conceptually belong to the interface of a tree node, rather than of a tree. Since the tree definition above does not include the declaration of a tree node, they are defined as top-level methods of the tree abstraction whose first argument is `node: Object`.

Second, we loose type safety, as illustrated in Fig. 10, where we 'compose' the expression and GUI views of the system by adapting expressions to fit in the conceptual world of a `TreeModel`. Since we use `Object` all the time, we cannot rely on the type checker to prove our code statically safe because type-casts are ubiquitous.

```
class ExpressionDisplay implements TreeModel {
  ExpressionDisplay(Expression r) { root = r; }
  Expression root;
  Object getRoot() { return root; }
  Object[] getChildren(Object node) {
      return ((Expression) node).getSubExpressions();
  }
  String getStringValue(Object node, boolean selected,
    boolean expanded, boolean leaf, int row, boolean focus){
    String s = ((Expression) node).description();
    if (focus) s ="<"+s+">";
      return s;

  }
}
```

Fig. 10. Using `TreeModel` to display expressions

The question naturally raises here: Why didn't the Swing designers define an explicit interface for tree nodes as in Fig. 11? The problem is that than it becomes more difficult to decouple the two contracts, i.e., the data structures to be displayed from the display algorithm. The idea is that the wrapper classes around e.g., `Expression` would look like in Fig. 12. The problem with such kind of wrappers, as also indicated in [3], is that we create new wrappers every time we need a wrapper for an expression. This leads to the *identity hell*: we loose the state and identity of previously created wrappers for the same node. The questionable alternative would be to use hash tables which is not only laborious but does also involve the definition and use of additional classes for maintaining these hashtables, thereby rendering the code more complex and less readable[3].

```
interface TreeDisplay {
  TreeNode getRoot();
}
interface TreeNode {
  TreeNode[] getChildren();
  String getStringValue(boolean selected,
   boolean expanded, boolean leaf, int row, boolean focus);
}
```

Fig. 11. `TreeDisplay` interface with explicitly reified `TreeNode` interface

```
class ExprAsTreeNode
implements TreeNode {
  Expression expr;
  void getStringValue(...) {  /*as before*/  }
  TreeNode[] getChildren() {
    Expressions[] subExpr = expr.getSubExpressions();
    TreeNode[] children =
          new TreeNode[subExpr.length];
    for (i = 0; i<subExpr.length; i++) {
       children[i] = new ExprAsTreeNode(subExpr[i]));
    }
    return children;
  }
}
```

Fig. 12. Mapping `TreeNode` to `Expression`

[3] Actually, Swing offers a `TreeNode` interface similar to the one in Fig. 11. However, classes that define data structures to be displayed as tree nodes should anticipate this and explicitly implement the interface.

A final important point to make before leaving this branch of the discussion is that it is difficult and awkward to associate state with abstractions like our tree nodes. We might want to associate state with tree nodes in both the `ExpressionDisplay` class in Fig. 10 and also inside the tree GUI control. For example, we might want to cache the computed string value or children in Fig. 10, because the re-computation might be expensive. In the GUI control itself, we might want to associate state like whether a tree node is selected or not or its position on the screen with the respective tree node. The only means to associate state with tree nodes is to make extensive use of hash tables, which is laborious and awkward.

Lack of support for bidirectional communication.

Interfaces provide clients with a contract as what to expect from a server object that implements the interface. We say, they express the *provided* contract. In order to define generic partial, view-specific models which are decoupled from their potential contexts of use, expressing expectations that a server might have on potential contexts of use is as important. We use the term *expected* contract to denote these expectations. What is needed is support for a loose coupling of client and server, that is (a) decoupling them to facilitate reuse, while (b) enabling them to tightly communicate with each other as part of a whole.

In our example, `TreeGUIControl` corresponds roughly to what we called the provided contract, while `TreeModel` corresponds roughly to what we called the expected contract. The class `SimpleTreeDisplay` represents a sample implementation of the provided contract. It expects from the context a concrete implementation of `getChildren(...)` and `getStringValue(...)`. In our terminology, `ExpressionDisplay` in Fig. 10 represents an implementation of the *expected* contract.

The design in Fig. 8 does actually a good job in decoupling the expected and provided contracts. Different implementation of GUI controls can be written to the `TreeModel` interface and can therefore be reused with a variety of concrete implementations of it, i.e., with a variety of data structures. The other way around, any data structure to be displayed is decoupled from a specific tree GUI control (e.g., `JTree`), such that the data structure can be displayed with different GUI tree controls.

Now, consider the `getStringValue(...)` method in Fig. 8 and Fig. 10. This method has noticeable many parameters that might be of interest when computing a string representation of the node. *Might be.* The sample implementation in Fig. 10 uses only the `selected` parameter and ignores the others. That means, the tree GUI control, which calls this method on the `TreeModel` interface, has to perform expensive computations to obtain the parameter values for this method (see implementation of `SimpleTreeDisplay::display(...)` in Fig. 8), although they might be rarely all used.

This is a typical case where we would like to establish a bidirectional communication between the two contracts of the tree displaying component. Here

we would like `ExpressionDisplay.getStringValue` to explicitly ask the tree GUI control to compute only relevant values for it, like `selected` or `hasFocus`, implying the GUI control interface provides respective operations. Recall that the GUI control interface corresponds to the provided interface of our generic component for displaying arbitrary data structures that can be viewed as trees in a GUI. As for now, the interfaces are completely separated (into `TreeModel` and `TreeGUIControl`), and there is nothing in the design that would suggest their tight relation as two faces of the same abstraction. As such, there is no build-in support for bidirectional communication between their respective implementations. Build-in means by the virtue of implementing two faces of the same abstraction, which serves as the implicit communication channel.

One can certainly achieve the desired communication by additional infrastructure (e.g., via cross-references) which has to be communicated to the respective programmers. However, we think that bidirectional communication is such a natural and frequent concept that the overhead that is necessary to enable bidirectional communication with conventional interfaces is too high. Please note that the additional `TreeNode` interface would also be of no help concerning the bidirectional communication problem exemplified by the `getStringValue()` method.

3 Caesar in a Nutshell

In this section, we will give an overview of the concepts that comprise our model by means of the `TreeDisplay` example from the previous section.

3.1 Collaboration Interfaces, their Implementations and Bindings

In order to cope with the problems discussed in Sec. 2 we propose the notion of *collaboration interfaces* (*CI* for short), which differ from standard interfaces in two ways. First, CIs introduce the `provided` and `required` modifiers to annotate operations belonging to the provided and the expected contracts, respectively, hence supporting bidirectional interaction between clients and servers. Second, CIs exploit interface nesting in order to express the interplay between multiple abstractions participating in the definition of a generic component.

For illustration, the CI `TreeDisplay` that bundles the definition of the generic tree displaying functionality from Sec. 2 is shown in Fig. 13. As an example for the "reification" of provided and expected contract, consider the methods `TreeDisplay.display()` and `TreeDisplay.getRoot()` in Fig. 13. Any tree display object is able to display itself on the request of a client - hence the `provided` modifier for `TreeDisplay.display`. However, in order to do so, it expects a client specific way of how to access the root tree node. What the root of a displayable tree will be depends on (a) which modules in a concrete deployment context of `TreeDisplay` will be seen as tree nodes and, (b) which one of them will play the role of the root node. Hence, the declaration of `getRoot` with the expected modifier. `TreeDisplay` comes with its own definition of a tree node:

The CI `TreeNode` is nested into the declaration of `TreeDisplay`. Please note that nesting of bidirectional interfaces in our approach has a much deeper semantics than usual nested classes and interfaces in Java: the nested interfaces are namely *virtual types* as in [1]. We will elaborate on that in Sec. 3.4.

```
interface TreeDisplay {
  provided void display();
  expected TreeNode getRoot();

  interface TreeNode {
    expected TreeNode[] getChildren();
    expected String getStringValue();
    provided display();
    provided boolean isSelected(),
    provided boolean isExpanded();
    provided boolean isLeaf();
    provided int row();
    provided boolean hasFocus();
  }
}
```

Fig. 13. Collaboration interface for TreeDisplay

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

An implementation class of a CI must (a) implement all `provided` methods of the CI and (b) provide an implementation class for each of the CI's nested interfaces. In doing so, it is free to use respective `expected` methods. In addition, an implementation class may or may not add additional methods and state to the CI's abstractions it implements. Fig. 14 shows a sample tree GUI control that implements `TreeDisplay`. The class `SimpleTreeDisplay` implements the only provided operation of `TreeDisplay`, `display()`, by forwarding to the result of calling the expected operation `getRoot()`. In addition to implementing `display()`, `SimpleTreeDisplay` must also provide a nested class that implements `TreeNode` - the only nested interface of `TreeDisplay`. The correspondence between a nested implementation class and its corresponding nested interface is based on name identity – `SimpleTreeDisplay` e.g., defines a class named `TreeNode` which is the implementation of the nested interface with the same name in `TreeDisplay`. This nested class has to implement all `provided` methods of the `TreeNode` interface, e.g., `display()`. The declaration of the instance variable

`boolean selected` and the corresponding query operation `isSelected` in `SimpleDisplay.TreeNode` are examples of new declarations added by an implementation class. Please note that just as nested interfaces, all nested implementation classes are virtual types (see Sec. 3.4).

```
class SimpleTreeDisplay implements TreeDisplay {
  void onSelectionChange(TreeNode n, boolean selected) {
    n.setSelected(true);
  }
  void display() { getRoot().display(); }

  class TreeNode {
    boolean selected;
    ...
    boolean isSelected() { return selected; }
    // other provided methods similar to selected
    void setSelected(boolean s) { selected =s;}
    void display() {
      ... TreeNode c = getChildren()[i];
      ... paint(position, c.getStringValue());
      ...
    }
  }
}
```

Fig. 14. A sample implementation of `TreeDisplay`

A binding class of a CI must (a) implement all `expected` methods of the CI, and (b) provide zero or more binding classes for each of the CI's nested interfaces (we may have multiple bindings of the same interface, see subsequent discussion). Just as implementation classes can use their respective expected facets, the implementation of the expected methods of a CI and its nested interfaces can also call methods declared in the respective provided facets. The process of binding a CI instantiates its nested types for a concrete usage scenario of the generic functionality defined by the CI. Hence, it is natural that in addition to their provided facets, binding classes also use the interface of abstractions from that concrete usage scenario. We say that bindings wrap abstractions from the world of the concrete usage scenario and map them to abstractions from the generic component world.

For illustration, the class `ExpressionDisplay` in Fig. 15 shows an example of binding the generic `TreeDisplay` CI from Fig. 13 for the concrete usage scenario, in which `Expression` structures are to be viewed as the trees to display. First, `ExpressionDisplay` binds the nested type `TreeNode` as shown in the nested class `ExprTreeNode`. The latter implements all expected methods of `TreeNode` by using (a) the provided facet of `TreeNode`, and (b) the interface

of the class `Expression` (via the instance variable `e`). Consider e.g., the implementation of the method `ExprTreeNode.getStringValue()`, which calls both `TreeNode.hasFocus()` as well as `Expression.getDescription()`.

In addition to binding `TreeNode`, `ExpressionDisplay` also implements the method `getRoot()` - the only method declared in the *expected* facet of `TreeDisplay`. Here is where the reference `root` to the `Expression` object to be seen as the root of the expression structure to display is transformed into a `TreeNode` by being wrapped into an `ExprTreeNode` object. Please note that this wrapping does not happen via an ordinary constructor call - `new ExprTreeNode(root)` in this case -, but rather by means of the *wrapper recycling* call `ExprTreeNode(root)`. We will elaborate on the concept of *wrapper recycling* in a moment.

```
class ExpressionDisplay binds TreeDisplay {
  Expression root;
  public ExpressionDisplay(Expression rootExpr) {
     root = rootExpr;
  }
  TreeNode getRoot() { return ExprTreeNode(root); }
  class ExprTreeNode binds TreeNode {
    Expression e;
    ExprTreeNode(Expression e) { this.e=e;}
    TreeNode[] getChildren() {
      return ExprTreeNode[](e.getSubExpressions());
    }
    String getStringValue() {
      String s = e.description();
      if (hasFocus()) s ="<"+s+">";
      return s;
    }
  }
}
```

Fig. 15. Binding of `TreeDisplay` for expressions

Except of binding the interface `TreeNode`, `ExprTreeNode` is basically a usual class that, in this case, wraps an instance of `Expression`. Since wrapping of objects in these classes is a very common task, we add some syntactic sugar for the most common case, namely by a `wraps` clause.

The semantics of `wraps` is that

```
class ExprTreeNode binds TreeNode wraps Expression {...}
```

is equivalent to

```
class ExprTreeNode binds TreeNode {
  Expression wrappee;
```

```
  ExprTreeNode(Expression e) { wrappee = e;}
  ... }
```

Using `wraps`, the code in Fig. 15 can be rewritten as in Fig. 16. In the following code we will make frequent use of `wraps` but it is important to understand that it is just syntactic sugar and does not prevent us to create arbitrarily complex initialization procedures by using ordinary constructors.

```
class ExpressionDisplay binds TreeDisplay {
  ...
  class ExprTreeNode binds TreeNode wraps Expression{
    TreeNode[] getChildren() {
      return ExprTreeNode[](wrappee.getSubExpressions());
    }
    String getStringValue() {
      String s = wrappee.description();
      if (hasFocus()) s ="<"+s+">";
      return s;
  }
}
```

Fig. 16. Alternative encoding of `ExprTreeNode` using the `wraps` clause

Binding classes and their nested classes are "almost standard" classes – we do not use more declarative mapping constructs because the full computational power of a general-purpose programming language is needed to express arbitrarily complex mappings, and this is very hard to achieve with declarative means. "Almost standard", however, stands for two differences. First, nested binding classes are also virtual types (see Sec. 3.4). Second, they make use of the notion of *wrapper recycling*, which we discuss next.

3.2 Wrapper Recycling

Wrapper recycling is our mechanism to escape the wrapper identity hell mentioned in Sec. 2. It is a concept on how to create and maintain wrapper instances, and a way to navigate between abstractions of the component world and abstractions of the base world - the concrete usage scenario world -, ensuring that the same (identical) wrapper instance will always be retrieved for a set of constructor arguments. This way the state and the identity of the wrappers is preserved.

Syntactically, wrapper recycling refers to the fact that, instead of creating an instance of a wrapper `W` with a standard `new W(args)` constructor call, a wrapper is retrieved with the construct `outerClassInstance.W(args)`. For illustration consider once again the expression `return ExprTreeNode(root)` in the method `ExpressionDisplay.getRoot()` in Fig. 15. We already mentioned

in the previous section that the expression in the return statement is not a standard constructor call, but rather a wrapper recycling operator. We use the usual Java scoping rules, i.e., `return ExprTreeNode(root)` is just an abbreviation for `return this.ExprTreeNode(root)`.

The idea is that we want to avoid creating a new `ExprTreeNode` wrapper each time the method `getRoot()` is called on an `ExpressionDisplay`. The call to the wrapper recycling operation `ExprTreeNode(root)` is equivalent to the corresponding constructor call, only if a wrapper for `root` does not already exist, ensuring that there is a unique `ExprTreeNode` wrapper for each expression within the context of the enclosing `ExpressionDisplay` instance. That is, two subsequent wrapper retrievals for an expression `e` yield the same wrapper instance - the identity and state of the wrapper are preserved.

This is due to the semantics of a wrapper recycling call, which is as follows: The outer class instance maintains a map `mapW` for each nested wrapper class `W`. An expression `outerClassInstance.W(wrapperargs)` corresponds to the following sequence of actions:

1. Create a compound key for the constructor arguments, lookup this key in `mapW`.
2. If the lookup for the key fails, create an instance of `outerClassInstance.W` with the annotated constructor arguments, store it in the hash table `mapW`, and return the new instance. Otherwise return the object already stored in `mapW` for the key.

The wrapper recycling call `ExprTreeNode[](...)` in the method `ExprTreeNode.getChildren` in Fig. 15 is an example for the syntactic sugar we use to express wrapper recycling of arrays, namely an automatic retrieval of an array of wrappers for an array of base objects.

3.3 Composing Bindings and Implementations

Both classes defined in Fig. 14 and 15 are not operational, i.e., cannot be instantiated, even if they are not annotated as abstract. These classes are indeed not abstract, since they are complete implementations of their respective contracts. The point is that the respective contracts are parts of a whole and make sense only within a whole. Operational classes that completely implement an interface are created by composing an implementation and a binding class, syntactically denoted as *aCollabIfc*`<`*aBinding*`,`*anImpl*`>`. This is illustrated by the class `SimpleExpressionDisplay` in Fig. 17, which declares itself as an extension of the composed class `TreeDisplay <SimpleTreeDisplay,ExpressionDisplay>`. Only such compound classes are allowed to be instantiated by the compiler. For instance, Fig. 17 also shows sample code that instantiates and uses the compound class `SimpleExpressionDisplay`.

Combining two classes as in Fig. 17 means that we create a new compound class within which the respective implementations of the `expected` and `provided` methods are combined. The same combination also takes place recursively for

```
class SimpleExpressionDisplay extends
   TreeDisplay<SimpleTreeDisplay,ExpressionDisplay> {}
...

Expression test = new Plus(new Times(5, 3), 9);
TreeDisplay t = new SimpleExpressionDisplay(test);
t.display();
```

Fig. 17. Creating and using compound classes

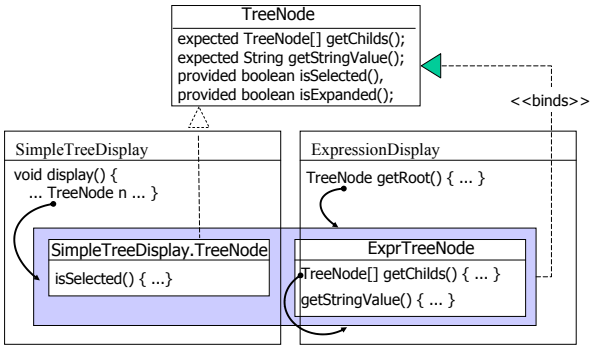

Fig. 18. Type rebinding in compound classes

the nested classes: All nested classes with a `binds` declaration are combined with the corresponding implementation from the component class. The separation of the two contracts, their independent implementation, and the dedicated late composition, allows us to freely reuse implementations of the two contracts in arbitrary compositions. We could combine `SimpleTreeDisplay` with any other binding of `TreeDisplay`. Similarly, `ExpressionDisplay` could be combined with any other implementation of `TreeDisplay`.

Note that the overall definition of the nested type, e.g., `TreeNode`, depends on the concrete composition of implementation and binding types within which the type is used. This does not only affect the external clients, but also the internal references. For instance, references to `TreeNode` within `ExpressionDisplay` and `SimpleDisplay` are rebound to the composed definition of `TreeNode` in `SimpleExpressionDisplay`, as illustrated in Fig. 18. Their meaning would be different in another compound class, e.g., resulting from composing `SimpleDisplay` with another binding class, or `ExpressionDisplay` with another implementation class. This is a natural consequence of the fact that nested types introduced by the collaboration interfaces are virtual types, on which we will elaborate in the following.

3.4 Virtual Types

In CAESAR approach, all types that are declared as nested interfaces of a CI and all classes that implement or bind such interfaces (including classes that extend the latter) are *virtual types* and *virtual classes*, respectively [6]. In the context of this paper, we use the notion of virtual types of the family polymorphism approach [1]. This means: (a) similar to fields and methods, types also become properties of objects of the class in which they are defined, and consequently (b) their denotation can only be determined in the context of an instance of the enclosing class. Hence, the meaning of a virtual type is late bound depending on the receiver object that executes when the virtual type at hand is referenced.

Consequently, all type declarations, constructor calls, and wrapper recycling calls for virtual types/classes within a CI are actually always annotated with an instance of the enclosing class. That is, type declarations and constructor invocations are always of the form `enclInst.MyVirtual x`, respectively `enclInst.MyConstructor()`. Similarly, wrapper recycling calls are also always of the form `outerClassInstance.W(args)` and not simply `W(args)`. For the sake of simplification, we apply the scoping rules common for Java nested classes also to type declarations and constructor or wrapper recycling calls: A call `OuterClass.this.W(args)` can be shortened to `W(args)`, and the type declaration `OuterClass.this.W` can be shorted to `W` as long as there are no ambiguities. This scoping rule applies to all type declarations and wrapper recycling calls that have appeared so far in this chapter.

For instance, all references to `ExprTreeNode` in Fig. 15 should be read as `ExpressionDisplay.this.ExprTreeNode`. The implication is that the meaning of any reference to the type name `ExprTreeNode` within the code of `ExpressionDisplay` will be bound to the compound class that combines `ExpressionDisplay.ExprTreeNode` with the implementation class of `TreeNode` that is appropriate in the respective execution context. For example, in the context of a `SimpleExpressionDisplay` as in Fig. 17, `ExprTreeNode` will be bound to the respective definition in the compound class `TreeDisplay<ExpressionDisplay,TreeNode>`. The same references will be bound differently if they occur in the execution context of an object of some subclass of `ExpressionDisplay` or in the context of a different implementation class. The same also applies to nested implementation and compound classes.

The rationale behind using virtual types lies in their power with respect to supporting reuse and polymorphism, as argued in [1]. We will rather shortly discuss how our specific use of virtual types (borrowed from [1]) does *not* suffer from covariance problems usually associated with virtual types, as for example the virtual type proposal in [18], which requires runtime type checks. If we have a virtual type in a contravariant position, as for example the argument type of `setRoot` in Fig. 19, type safety is still preserved, because subsumption is disallowed if the enclosing instances are not identical. In order to make the approach sound, all variables that are used as part of type declarations have to be declared as `final` because otherwise the meaning of a type declaration might change due to a field update. For illustration consider the declaration of the

variable `ed` in the sample code in Fig. 19. It is used as part of a type declaration for the variable `t` and is therefore declared as `final`. For more details on typing issues we refer to [1].

```
Expression e = ...;
final ExpressionDisplay ed =
                    new SimpleExpressionDisplay(e);
...
// let FileSystemDisplay be a binding of
// TreeDisplay to the file system structure
class SimpleFileSystemDisplay extends
 TreeDisplay<SimpleTreeDisplay,FileSystemDisplay> {};

FileSystem fs = ... ;
final FileSystemDisplay fsd =
       new SimpleFileSystemDisplay(fs);
...
ed.TreeNode t = ed.getRoot();
fsd.setRoot(t); // Type error detected by typechecker!
        // sd.TreeNode is not subtype of ed.TreeNode
```

Fig. 19. Type safety due to family polymorphism

4 Evaluation

Before discussing the implications of our new model, let us at first compare the new solution to the conventional solution discussed in Sec. 2.

- Other than the Swing interface in Fig. 8, we do not need to use `Object`; every item is well-typed and we do not need type casts. The methods that are conceptually part of the interface of tree nodes, are expressed as methods of a dedicated nested interface.
- Due to bidirectional interfaces, we do not have the problem related to the `getStringValue()` parameters: The implementation of this method, as in Fig. 15, causes the computation of only those values about the state of displaying that are really needed by means of calling appropriate methods in the `provided` interface.
- It is easy to associate additional state with tree nodes. For example, the `TreeNode` implementation in Fig. 14 adds a `selected` field, and the `TreeNode` binding in Fig. 15 could as well have added extra state to `ExprTreeNode`.

Abstracting from the concrete example, what have we gained with the new language means proposed in this paper? The important point is the idea of encoding modules in terms of their own world-view, encoded in a collaboration

interface, together with means to translate this world view into the world view of a particular application by means of CI bindings. Due to the independence of a binding from a particular CI implementation, CI bindings are universal, reusable "world-view translators".

5 Related Work

Pluggable Composite Adapters (PCAs) [13] and their predecessor, *Adaptive Plug and Play Components (APPCs)* [10], have been important starting points for our work. Both approaches offer different means for on-demand remodularization. The APPC model had a vague definition of required and provided interfaces. However, this feature was rather ad-hoc and not well integrated with the type system. Recognizing that the specification of the required and expected interfaces of components was rather ad-hoc in APPCs, PCAs even dropped this notion and reduced the declaration of the expected interface to a set of standard abstract methods. With the notion of collaboration interfaces, the approach presented here represents a qualitative improvement over PCA and APPC.

The Hyperspace model and its instantiation in Hyper/J [17] also target multiple co-existing hierarchies. However, despite the common goal, there are some important differences between these two approaches. In a nutshell, the functionality offered by Hyper/J can be summarized as *extracting concerns* and *composing concerns*.

Extracting concerns means that one can take a piece of existing software and tag parts of the software, e.g., method `a()` in class `A` and method `b()` in class `B`, by means of a so-called *concern mapping*. Later, this mapping can be used to extract a particular concern from this software and reuse it in a different context. This is similar to the old idea of retroactive generalization in inheritance hierarchies [15]. An important concept for extracting concerns is the notion of *declarative completeness*. Basically, this means that all methods that are used inside the tagged methods but are not tagged themselves are declared as *abstract* in the context of the extracted concern. Our model does not have any dedicated means for feature extraction.

However, we think that with respect to *composing concerns* our approach is in some important ways superior to Hyper/J. Composition in Hyper/J happens by means of a so-called *hypermodule specification*, which describes in a declarative sublanguage, how different concerns should be composed. In terms of our model, a hypermodule performs both the functionality of our binding classes and the actual composition with the `+` operator. Due to this mixing and due to the absense of an interface concept similar to our collaboration interface, Hyper/J has no polymorphism and reuse as in our approach, e.g., one cannot switch between different implementations and bindings, and one cannot use them polymorphically. Since the mapping sublanguage is declarative, it relies on similar signatures that can be mapped to each other, and transformations other than name transformations (e.g., type transformations), are very difficult. In addition,

Hyper/J's sublanguage for mapping specifications from different hyperslices is fairly complex and not well integrated into the common OO framework.

The last important difference is that Hyper/J's approach is class-based: it is not possible to add the functionality defined in a hyperslice to individual objects, instead the objects have to be created as objects of the compound hypermodule from the very beginning. Therefore, multiple independent bindings that are added to individual objects at runtime are not possible.

Hölzle [3] analyses some problems that occur when combining independent components. Our proposal can be seen as an answer to the problems and challenges discussed in [3].

Our work is also related to *architecture description languages* (ADL) [16], for example Rapide [5], Darwin [7], C2 [9], and Jiazzi [8]. The building blocks of an architectural description are components, connectors, and architectural configurations. A component is a unit of computation or data store, a connector is an architectural building block used to model interactions among components and rules that govern those interactions, and an architectural configuration is a connected graph of components and connectors that describe architectural structure. In comparison with our approach, ADLs are less integrated into the common OO framework, and do not have a dedicated notion of on-demand remodularization in order to provide a new virtual interface to a system.

We think that collaboration interfaces might also prove very useful in the context of ADL. In ADL, components also describe their functionality and dependencies in the form of required and provided methods (so-called *ports*). The goal of these ports is to render the components reusable and independent from other components. However, although the components are syntactically independent, there is a very subtle semantic coupling between the components, because a component `A` that is to be connected with a component `B` has to provide the exact counterpart interface of `B`. The situation becomes even worse if we consider multiple components that refer to the same protocol. The problem is that there is no central specification of the communication protocol to which all components that use this protocol can refer to – in other words: we have no notion of a collaboration interface.

6 Summary

Traditional programming languages assume that real-world systems have "intuitive", mind-independent, preexisting concept hierarchies. We argued that this is in contrast to our perception of the world, which depends heavily on the context from which it is viewed. We identified the *arbitrariness of the decomposition hierarchy* as the main cause of 'code tangling' and presented a new model called Caesar, within which it is possible to have multiple different decompositions *simultaneously* and to add new decompositions on-demand.

References

1. E. Ernst. Family polymorphism. In *Proceedings of ECOOP '01*, LNCS 2072, pages 303–326. Springer, 2001.
2. D. Garlan, G. E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *Computer*, 25(6):30–38, 1992.
3. U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings ECOOP '93, LNCS*, 1993.
4. Java Foundation Classes. http://java.sun.com/products/jfc/.
5. D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
6. O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89*. ACM SIGPLAN, 1989.
7. J. Magee and J. Kramer. Dynamic structure in software architecture. In *Proceedings of the ACM SIGSOFT'96 Symposium on Foundations of Software Engineering*, 1996.
8. S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *Proceedings of OOPSLA '01*, 2001.
9. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in C2-style architectures. In *Proceedings of the 1997 international conference on Software engineering*, pages 692–700, 1997.
10. M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, 1998.
11. M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of OOPSLA '02, Seattle, USA*, 2002.
12. M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proc. International Conference on Aspect-Oriented Software Development (AOSD '03), Boston, USA*, 2003.
13. M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer, 2001. University of Twente, The Netherlands.
14. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
15. C. H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *OOPSLA '89 Proceedings*, 1989.
16. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. PrenticeHall, 1996.
17. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. International Conference on Software Engineering (ICSE 99)*, 1999.
18. K. K. Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP '97*, 1997.