Integrating Independent Components with On-Demand Remodularization

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

Klaus Ostermann
Siemens AG, CT SE 2
D-81730 Munich, Germany
Klaus.Ostermann@mchp.siemens.de

ABSTRACT

This paper proposes language concepts that facilitate the separation of an application into independent reusable building blocks and the integration of pre-build generic software components into applications that have been developed by third party vendors. A key element of our approach are *on-demand remodularizations*, meaning that the abstractions and vocabulary of an existing code base are translated into the vocabulary understood by a set of components that are connected by a common *collaboration interface*. This general concept allows us to mix-and-match remodularizations and components on demand.

Keywords

On-Demand Remodularization, Fluid Aspect-Oriented Programming, Collaboration-Based Decomposition

1. INTRODUCTION

This paper proposes language concepts that facilitate the separation of an application into independent reusable building blocks and the integration of pre-build generic software components into applications that have been developed by third party vendors. By *generic component* we mean any piece of generic application logic - a generic software building block - that is implemented independently of any particular application it might be integrated into in the future.

An example is a component that implements a graph coloring[1] algorithm in terms of a generic graph structure, consisting of node and edge abstractions, i.e., independent of the structure of any particular application that might need graph coloring. It makes sense to capture graph algorithms in a generic way, since they are used in almost any application domain. For example, the university administration example in Fig. 1 can be viewed as a graph whose vertices are the courses and edges are gained by connecting any pair of courses that are taught by the same teacher or required by the same student year. This view on the module structure of the software would be necessary on the demand of applying the graph coloring component in order to compute a minimum coloring, which would represent an optimal assignment of time slots to courses.

Several other eventually completely different or overlapping remodularized views of the university administration example can be thought of that are needed on the demand of adding new features to the software by applying the coloring algorithm for other purposes than time slot assignment, or by applying other algorithms, e.g., graph matching[2] for supporting yet quite other features. Tab. 1 presents different sample representations of the university administration example as a graph.

One can easily generalize from graph algorithms to any kind of generic application logic in other domains such e.g., price calculation logic in a e-commerce order system, bonus program in an online travel agency software, etc. In the following, we call application code to which we want to apply a specific component (like the aforementioned university administration example) the *base code*, and the code that implements the desired functionality (like some graph algorithm) is the *component code*.

The message conveyed from the discussion so far is that appropriate language technology is needed to support a software development process in which generic functionalities such as graph coloring or price calculation (a) are provided as ‘off-the-shelf’ components whose implementation is decoupled from any particular application, and (b) can be integrated a-posteriori into a multitude of existing software, implying the needed remodularization of the existing software, however, without physically changing it. A physical change of the modular structure of the base application is not only undesirable; it is even impossible, since different generic components have in general quite different views of what the modular structure should be. In the absence of such a language technology, the implementation of different graph algorithms will be scattered around several classes, of-

[1] A graph coloring is an assignment of colors to the vertices of a graph such that no two vertices that are connected by an edge have the same color. A minimum coloring is a coloring with a minimum number of colors.

[2] A matching is a subset $M \subseteq E$ of the graph's edges such that every vertex is connected to at most one edge from M. A maximum matching is matching with a maximum number of edges in M.

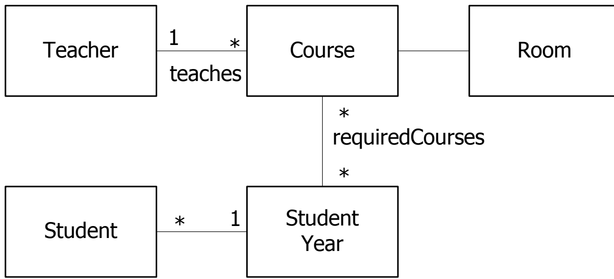

Figure 1: University Example

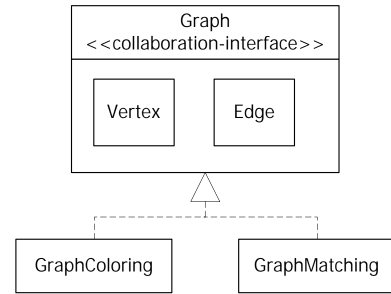

Figure 2: Graph collaboration

fen dublicated, rendering the resulting software a nightmare to maintain and evolve.

Unfortunately, as argued in [2, 8, 16], current object-oriented languages are not very well equipped to cope with the subtle problems that occur when integrating independently developed components. Industrial component models such as EJB and CCM do not tackle this problem, either. With beans in the EJB model one can indeed 'write application logic once and run it in (almost) any server platform'. However, late integration of generic independently developed application logic into existing EJB software is not supported.

The problem does not only apply to the integration of components from third party vendors but also to the integration of reusable modules in general: those that capture different separated concerns of a system in any software engineering effort. The principles of *separation of concerns* [5], which manifests itself today in the form of *aspect-oriented programming* [12, 26, 1, 18], tells us that we should try to divide our software in smaller pieces that are as independent from each other as possible, in order to facilitate maintenance, understandability and reusability. However, as indicated in [11], the *aspect-oriented programming* community recognizes that much has to be still done for supporting flexible integration of crosscutting concerns.

The work presented in this paper aims at improving the state-of-the-art technologies targeted at the problem domain outlined so far. Probably, the most important contribution of our approach is the introduction of *collaboration interfaces* for declaring generic component types. Collaboration interfaces are in general nested interfaces, allowing bundling of several abstractions that together build up the concept world of a component type into a family of virtual types [6]. For example, a component that provides algorithms on graphs articulates its world outlook - the structure and the requirements of a graph to which the algorithms could be applied - in the collaboration interface. Other components that also operate on graphs can refer to the same collaboration interface. This is schematically illustrated in Fig. 2.

Standard interfaces as we know them ,e.g., from Java, express what a client can expect from a server. In addition to this 'client-from-server contract', collaboration interfaces (both simple and nested) also capture what servers expect from potential client contexts in which they might be integrated. We say, they also make explicit the server-from-client contract. The implementations of these two contracts are completely decoupled from each other. We distinguish between *implementing a component type* and *binding a component type*. The general architecture for using our proposal is outlined in Fig. 3.

Implementing a component type means implementing the first contract of its collaboration interface. The implementation relies on the declarations in the second contract in order to remain oblivious of the potential contexts of use. This renders a component implementation independent of specific applications. However, by the second contract being integral part of its type, any component implementation carries around a port that makes it pluggable into unknown worlds.

Binding a component type to a concrete application means implementing the second contract of its collaboration interface with respect to the application at hand. We assume that the world of a particular application into which a component type should be integrated, is, in general, very different from the component's world. Therefore, we introduce means for *on-demand remodularization* [21] of the application during binding. These are used to express how the abstractions of a base application should be translated to the vocabulary of a particular collaboration interface. An important property of our approach is its ability to create multiple independent remodularizations of the same objects, indicated in Fig. 3 by two different remodularizations of the same structure. This was an important requirement on the integration of independent components (recall the different views of the university example in Tab. 1).

Analogous to component implementations, component bindings - also called remodularization modules - are oblivious of concrete implementations of their component type. However, by means of the second contract of their type, they can easily be plugged with arbitrary implementations. To the best knowledge of the authors, our approach is the first one that decouples the component implementations and remodularizations as indicated in Fig. 3 and thus allows us to combine arbitrary implementations of a collaboration interface with arbitrary bindings. In addition, due to the incorporation and generalization of common OO concepts (such as types, subtype polymorphism and late binding) reuse is very naturally supported in both dimensions: component

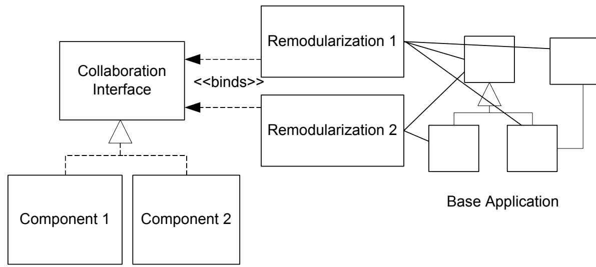

Figure 3: General architecture for collaboration interfaces and on-demand remodularization

implementation and remodularization.

An important insight that drove the development of our approach to on-demand remodularization is the fact that simple mappings from component abstractions to base classes ("In the collaboration C, class X plays the role R") are not sufficient: The base application does not necessarily have classes or objects that do directly correspond to a role in a particular collaboration. For example, there is no abstraction in the base code that corresponds to the edge role in the course collision graph (Tab.1). The edges are only implicitly represented as pairs of courses that need to be computed at runtime. Therefore, in general, the mapping of the abstractions needs full computational power. One of the contributions of this paper is the fact that we present an approach which allows such flexible mappings.

In addition, our on-demand remodularization is object-based rather than class-based. Class-based means that a remodularization that affects a class, applies to all instances of a class, while object-based means that the remodularization may be created for individual objects on-demand. The advantage of object-based remodularization is twofold. First, we have fine-grained control over the integration process because we can choose for each object whether it should be part of a collaboration or not. Second, the same object can participate in multiple component instances. For example, a particular course instance can be a vertex in the course collision graph and simultaneously an edge in the "Teacher uses Room" graph (see Tab.1). It can even be a vertex in another course collision graph which is independent from the first one.

The remainder of this paper is organized as follows. Sec. 2 presents shortcomings of current language technology with respect to supporting integration of generic components. Sec. 3 introduces our notions of collaboration interfaces and on-demand remodularization. Sec. 4 outlines future work, in particular our plans towards *fluid aspect-oriented programming*. Sec. 5 discusses related work, and Sec. 6 summarizes the paper.

2. PROBLEM STATEMENT

In this section, we set up the stage for the rest of the paper. The goal is to identify shortcomings of current object-oriented language technology with respect to supporting the development of generic components designed for late integration into various contexts of use. The following sections will present our proposal for coping with these shortcomings. The target of our criticism is the common concept of interfaces as we know it e.g., from Java. We argue that they lack two important features:

- **A**ppropriate support for declaring component types as a set of mutually recursive types. Defining generic components 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.
- **S**upport 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 *client-from-server contract*. In order to define generic components 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 *server-from-clients contract* to denote these expectations. What is needed is support for a loose coupling of these two contracts, that is (a) decoupling them to facilitate reuse, while (b) enabling them to tightly communicate with each other as part of a whole. Standard interfaces fail short - or at least render it very sumptuous - in this regard.

To illustrate these shortcomings, let us have a critical look at a simple example. Fig. 4 shows a simplified version of the `TreeModel` interface in Swing[3], Java's GUI framework [10]. This interface provides a generic description of the data model for a `JTree`, or other GUI tree controls. For illustration purposes, Fig. 4 also presents a pseudo interface for tree GUI controls in `TreeGUIControl`, as well as a pseudo implementation of this interface in `SimpleTreeDisplay` (the latter roughly corresponds to `JTree`).

In our terminology the code in Fig. 4 defines a generic component for displaying arbitrary data structures that can be viewed as trees in a GUI. When this component is used in a particular context, e.g., for object structures that represent arichmetic expressions, it provides to this context the `display` functionality. In turn, it expects from the context a concrete implementation of `getChildren` and `getStringValue`. These operations can only be implemented specifically for a concrete data type to be presented as a tree. That is, `TreeGUIControl` corresponds roughly to what we called the clients-from-server contract in the type of our component, while `TreeModel` correspond roughly to what we called the server-from-clients contract. The class `SimpleTreeDisplay` represents a sample implementation of the clients-from-server contract.

The design in Fig. 4 does actually a good job in decoupling these two contracts via the interface `TreeModel`. Different implementation of GUI controls rely on this interface and can therefore be reused with a variaty of concrete implementations of it, i.e., with a variaty of data structures. The

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

Graph	Vertex	Edge (v1,v2)
Course Collision	Courses	Teacher teaches both v1 and v2 or both v1 and v2 required by same student year
Student Contacts	Students	v1 and v2 visit a common course
Student knows Teacher	Students, Teachers	Student v1 visits a course by teacher v2
Teacher uses Room	Teachers, Rooms	Course with Teacher v1 and assigned room v2

Table 1: Possible Mappings from University Example to Graph

```
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() ...
      ...
      // prepare parameters for getStringValue
      ... tm.getStringtValue(...);
      ...
   }
}
```

Figure 4: Simplified version of the Java Swing `TreeModel` interface

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

Figure 5: Expression Trees

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

  }
}
```

Figure 6: Using `TreeModel` to display expressions

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.

So, what is wrong with the approach to specifying generic components exemplified by the design in Fig. 4? 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 abstrations 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.

The disadvantages of using the most general type, `Object`, are twofold. First, it is conceptually questionable. If every abstraction that is involved in the component 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 `Object node`.

Second, we loose type safety. Let us have a look at Fig. 5 and Fig. 6. Fig. 5 shows a simple base application for expressions, and Fig. 6 demonstrates how the expression classes can be adapted ('remodularized') to fit in the concetual world of a `TreeModel`. In our terminology, `ExpressionDisplay` in Fig. 6 represents an implementation of the server-from-client contract. 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.

The question naturally raises here: Why didn't the Swing designers define an explicit interface for tree nodes as in Fig. 7 from the very beginning? Well, there are good reasons for this. With the explicit type `NodeTree` 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. 8. The problem with such kind of wrappers, which have been very nicely presented in [8], is that we create new wrappers every time we need a wrapper for an expression, thereby leading to the *identity hell*, which refers to the fact that we loose the state and identity of previously created wrappers for the same node. The

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

Figure 7: `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;
  }
}
```

Figure 8: Mapping `TreeNode` to `Expression`

questionable alternative would be to use hash tables which is not only laborious but does also involve the definition and use of additional classes, thereby rendering the code more complex and less readable[4].

So far, we discussed problems resulting from the lack of appropriate support for defining multiple related abstractions in one module. Let us now illustrate the problems resulting from the second shortcoming of standard interfaces. Consider for this purpose the `getStringValue()` method in Fig. 4 and Fig. 6. 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. 6 uses only the `selected` parameter and ignores the rest. 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, 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`. 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. [5]

The third point is the fact 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. 6 and also inside the tree gui control. For example, we might want to cache the computed string value or children in Fig. 6, 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 not only laborious but does also involve the definition and use of additional classes, thereby rendering the code more complex and less readable.

3. LATE INTEGRATION AND ON-DEMAND REMODULARIZATION

In this section, we will present the main features of our proposal. The first subsection gives a short overview of the concepts by means of the `TreeDisplay` example from the previous section. The next subsection goes into more detail by means of the `Graph` example from the introduction. The last subsection summarizes what we have gained so far.

3.1 Concepts in a Nutshell

The discussion in Sec. 2 suggests the need for a new notion of interfaces. In particular, we demanded means to express the interplay between multiple abstractions and for bidirectional communication. Our proposal for these extensions are *collaboration interfaces*.

Fig. 9 shows a collaboration interface for the tree example discussed in Sec. 2.

The first obvious conceptual extension in Fig. 9 is the introduction of `provided` and `required` modifier. We think that interaction is frequently inherently bi-directional. Hence, it is quite natural to allow interfaces to be bi-directional as well, making explicit both client-from-server and server-from-client contracts. In our model, operations belonging to the client-from-server contract are declared with the modifier *provided*, while those belonging to the server-from-client contract are declared with the modifier *expected*.

This categorisation of the operations comes with a new model of what it means to implement an interface. We explicitly distinguish between *implementing* an interface's client-from-server contract and *binding* the same interface's

[4]In fact, Swing offers a `TreeNode` interface similar to the one in Fig. 7. However, classes that define data structures to be displayed as tree nodes should anticipate this and explicitly implement the interface.

[5]Please note that the additional `TreeNode` interface would also be of no help concerning the bidirectional communication problem exemplified by the `getStringValue()` method.

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

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

Figure 9: Collaboration interface for TreeDisplay

```
class ExpressionDisplay binds TreeDisplay {
  ExpressionDisplay(Expression r) { root = r; }
  Expression root;
  TreeNode getRoot() { return ExprTreeNode(root); }

  class ExprTreeNode binds TreeNode {
    Expression e;
    ExprTreeNode(Expression e) { this.e=e;}
    TreeNode[] getChilds() {
      return ExprTreeNode[](e.getSubExpressions());
    }
    String getStringValue() {
      String s = e.description();
      if (hasFocus()) s ="<"+s+">";
      return s;
  }
}
```

Figure 10: Binding of `TreeDisplay` for expressions

server-from-client contract. Two different keywords are used for this purpose: `implements`, respectively `binds`.

The second important point is the introduction of interface nesting. The interface `TreeNode` is nested into the declaration of `TreeDisplay`. We will go into more detail about nested interfaces later, but please note now that nesting of bidirectional interfaces in our approach has a much deeper semantics than usual nested classes and interfaces in Java.

A class that binds such an interface is declared with a `binds` declaration. Fig. 10 shows the class `ExpressionDisplay`, which binds the `TreeDisplay` interface of Fig. 9.

Binding an interface means that all `expected` methods have to be implemented by that class. In addition, nested classes can bind nested interfaces of the outer interface, e.g., `ExprTreeNode` binds `TreeNode`. Again, `ExprTreeNode` implements all `expected` methods and is on the other hand allowed to call the `provided` methods.

A special mechanism, called *wrapper recycling*, is employed in the methods `getRoot()` and `getChilds()`. We will elaborate on that later; for the moment this mechanism can be thought of as a special navigation mechanism between a base class like `Expression` and a corresponding wrapper class like `ExprTreeNode`.

```
class SimpleTreeDisplay implements TreeDisplay {
  void display() {
    TreeNode n = getRoot();
    ... TreeNode c = n.getChilds()[i];
    ... paint(position, c.getStringValue());
    ...
  }
  void onSelectionChange(TreeNode n, boolean selected) {
    n.setSelected(true);
  }
  class TreeNode {
    boolean selected;
    ...
    boolean isSelected() { return selected; }
    // other provided methods similar to selected
    void setSelected(boolean s) { selected =s;}
  }
}
```

Figure 11: A sample implementation of `TreeDisplay`

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

Figure 12: Demo code that uses the code in Fig. 10 and 11

Fig. 11 shows a sample tree GUI control that implements the inverse part of the interface, that is, it provides implementations for all `provided` methods and is free to use the `expected` methods.

`SimpleTreeDisplay` has to provide a nested class for all nested bidirectional classes of `TreeDisplay`. These classes are identified by name – `SimpleTreeDisplay` defines a class with name `TreeNode`. This nested class has to implement the inverse part of the `TreeNode` interface, namely all `provided` methods.

Implementation details are ignored for the moment, since we will look more carefully at the implementation and binding details in the following two sub-sections. What is important to note at this point is that there is only one implementation of the server-from-client contract for `Expression`. Furthermore, please note that the classes in Fig. 10 and 11 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 the 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, using the composition operator `+`, as illustrated by the class `SimpleExpressionDisplay` in Fig. 12, which shows some sample code that uses the compound class. Only such compound classes are allowed to be instantiated by the compiler.

Note that the overall definition of the type `TreeNode` de-

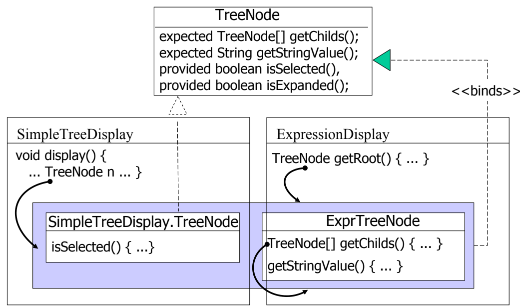

Figure 13: Integration Time Binding of Types

pends on the respective composition within which the type is used. This does not only affect the external clients, but also the internal references to `TreeNode` within the implementations of `ExpressionDisplay` and `SimpleDisplay`, as illustrated in Fig. 13. In other words, the nested types introduced by the collaboration interface are *virtual types* [14]. This integration time binding of types together with the separation of the two contracts and their independent implementation allows to freely reuse implementations of the two contracts in arbitrary compositions, as illustrated in Fig. 12: We could combine `SimpleTreeDisplay` as well with any other binding of `TreeDisplay`, and `ExpressionDisplay` could be combined with any other implementation of `TreeDisplay`.

As an interim result, let us compare this solution for the tree display to the conventional solution discussed in Sec. 2.

- In opposition to the Swing interface in Fig. 4, we do not need to use `Object`, every item is well-typed and we do not need type casts. The methods that are conceptually a part of tree nodes, are expressed as methods of a dedicated nested interface.
- Due to the use of bidirectional interfaces, we do not have the problem related to the `getStringValue()` parameters: The implementation of this method, as in Fig. 10, can call some of the `provided` methods if required, the other values are not even computed.
- It is easy to associate additional state with tree nodes. For example, the `TreeNode` implementation in Fig. 11 adds a `selected` field, and the `TreeNode` binding in Fig. 10 could as well have added extra state to `ExprTreeNode`.

3.2 Concepts in Detail

In this section we want to elaborate on the concepts introduced in 3.1 in more detail. For this purpose, we will introduce a more complex example that is better suited to demonstrate the advantages of our approach. This example is based on the idea of graph components already mentioned in the introduction.

Fig. 14 shows sample collaboration interfaces for graphs. The first interface `Graph` defines the general abstractions and

```
interface Graph {
  interface Vertex {
    expected Edge[] edges();
  }
  interface Edge {
    expected Vertex getV1();
    expected Vertex getV2();
  }
}

interface ColoredGraph extends Graph {
  provided computeMinimumColoring(Vertex v[]);
  override Vertex {
    expected void setColor(int c);
    expected int getColor();
  }
  override Edge {
    provided float getBadness();
}

interface MatchedGraph extends Graph {
  provided computeMaximumMatching(Vertex v[]);
  override Edge {
    expected void setMatched(boolean b);
    expected boolean isMatched();
  }
}
```

Figure 14: Graph collaboration interfaces

properties of these abstractions, and the `ColoredGraph` and `MatchedGraph` interfaces refine the `Graph` interface by adding methods or refining nested interfaces of `Graph`. Nested interfaces can be extended by means of an `override` declaration, as for example the `Vertex` refinement in `ColoredGraph`. The purpose of the `getBadness()` method is to have a measure how troublesome a particular edge is with respect to minimum coloring, meaning that if an edge with a high badness would be removed, it is likely that we can color the graph with less colors.

Fig. 15 shows two different components that implement the `ColoredGraph` interface of Fig. 14. They represent two different algorithms for graph coloring. Every component provides implementations for the `provided` methods that are specified in the implemented collaboration interface and is free to use the declared `expected` methods. In addition, it may or may not add additional methods and state to the collaboration abstractions. For example, `SuccessiveAugmentationColoring` adds a field `temp_color` to `Vertex`.

So far we have presented how to specify the collaboration interface of a generic component and how to implement the client-from-server contract of it. In the following, we elaborate on (a) the issues related to binding a collaboration interface to a concrete application, i.e., on the implementation of the server-from-client contract, and (b) on the integration of the two contract implementations by means of the `+` operator. As already mentioned, binding a collaboration interface to a concrete application implies an *on-demand remodularization* of the application. Remodularization means specifying how the model structure of the base application should be transformed to match the model structure expected by the collaboration interface. The term 'on-demand' is used to indicate two important characteristics of our remodular-

```
class SuccessiveAugmentationColoring
implements ColoredGraph {
  // successive augmentation coloring algorithm
  void computeMinimumColoring(Vertex v[]) {
    // successive augmentation coloring algorithm
    ... Edge e[] = v[i].getEdges(); ...
    ... Vertex w = e[j].getV2();
    ... if (w.isLegalColor(color)) w.temp_color = color;
    ... e[n].setBadness(badness); ...
    // commit final coloring
    for (int k=0; k<v.length;k++)
      v[k].setColor(v[k].temp_color);
  }
  class Vertex {
    int temp_color;
    boolean isLegalColor(int color) {
      Vertex neighbor[] = ...;
      for (int i=0;i<neighbor.length;i++)
        if (neighbor[i].getColor() == color) return false;
      return true;
    }
  }
  class Edge {
    float badness;
    float getBadness() { return badness; }
    void setBadness(float b) { badness = b; }
  }
}
class SimulatedAnnealingColoring
implements ColoredGraph {
  // Simulated Annealing coloring algorithm
  void computeMinimumColoring(Vertex v[]) {
    ...
  }
  ...
}
```

Figure 15: Different Coloring Algorithms

ization concept. First, remodularizations in our model are virtual, meaning that the base module structure is never changed physically; a remodularization rather defines a virtual view on top of the physical structure. Second, the remodularization specified for binding a collaboration interface `C` is effective only on the demand of executing functionality in `C`.

In Sec. 1 we gave an example that illustrated why the declarative mapping constructs as in [26, 18, 19] are not sufficient to express arbitrary on-demand remodularizations. In general, the full computational power of an object-oriented language is needed for this purpose. For this reason, our approach to specifying remodularizations is rather "manual". Our approach actually barely relies on other features than those available in the class construct for this purpose. In our model, remodularizations are implemented in 'almost standard' nested classes, which we call connector classes. Connectors wrap one or multiple abstractions of the base world and map them to an abstraction from the component world[6]. There are two special features of our connectors, each addressing one of the problems discussed in Sec. 2. Hence our description of them as 'almost standard nested classes'.

[6]Once we have covered the more difficult cases, convenient more declarative syntactic sugar mapping constructs can be introduced to the model for the simpler cases.

First, they have a built-in *wrapper recycling* feature, in order to escape the wrapper identity hell mentioned in Sec. 2. Wrapper recycling is a concept on how to create and maintain wrapper instances, and an intuitive way to navigate between abstractions of the component world and abstractions of the base world. It ensures that the same (identical) wrapper instance will always be retrieved for a set of constructor arguments. That is, the state and the identity of the wrappers is preserved.

For illustration, consider the code in Fig. 16, which shows a connector that binds the `ColoredGraph` interface and uses wrapper recycling. The nested classes `CourseCollision` and `CourseVertex` are remodularization wrappers around base objects. `CourseCollision` implements the expected interface of `ColoredGraph.Vertex` by wrapping an object `c` of type `Course`, while `CourseVertex` implements the expected interface of `ColoredGraph.Edge` by wrapping two objects of type `Course`, `c1` and `c2`. An example for using the wrapper recycling feature are the calls `CourseVertex(c1)` and `CourseVertex(c2)` in `getV1()`, respectively `getV2()`.

Syntactically, wrapper recycling refers to the fact that, instead of creating an instance of a wrapper `W` with a standard `new W(constructorargs)` constructor call, a wrapper is retrieved with the construct `outerClassInstance.W(constructorargs)`. The semantics of such an expression 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 key does not exist, create an instance of `outerClassInstance.W` with the annotated constructor arguments, store it in the hash table, and return the new instance. Otherwise return the object already stored in `mapW`.

For example, the wrapper recycling call `CourseVertex(c1)` in `getV1()` in Fig. 16 ensures that there is one unique `CourseVertex` wrapper for each course. That is, two subsequent wrapper retrievals for a course `c` yield the same wrapper instance - the identity and state of the wrapper are preserved. Similarly, there is only one unique instance of `CourseCollision` for every pair `(c1,c2)` of courses[7]

Please also note the argument `gc.CourseVertex[](courses)` in the `computeMinimumColoring` call. This is syntactic sugar for wrapper recycling of arrays, namely an automatic retrieval of an array of wrappers for an array of base objects.

The second important feature of our remodularization wrappers is that they are always created relative to

[7]In this example, an undirected edge in the course collision graph is represented by two directed edges. A direct representation of undirected edges would also be possible if we would pass a *set* with the two courses as elements in the `CourseCollision` constructor calls instead of the ordered pair of courses.

```
class SchedulingGraph binds ColoredGraph {
  class CourseVertex binds Vertex {
    Course c;
    Edge[] cachedEdges;
    CourseVertex(Course c) { this.c = c; }
    Edge[] edges() {
      if (cachedEdges == null) {
        Vector tc = c.getTeacher().getCourses();
        tc.append(c.getStudentYear().getCourses());
        cachedEdges = new Edge[tc.length];
        for (int i=0;i<tc.length;i++)
          cachedEdges[i] = CourseCollision(c,x);
        }
      }
      return cachedEdges;
    }
    void setColor(int color) {
      c.timeSlot = TimeSlots.getSlot(color);
    }
  }
  class CourseCollision binds Edge {
    Course c1,c2;
    CourseCollision(Course c1, Course c2) {
      this.c1=c1; this.c2 = c2;
    }
    Vertex getV1() { return CourseVertex(c1); }
    Vertex getV2() { return CourseVertex(c2); }
  }
}
```

Figure 16: Connector for scheduling graph

their enclosing component instance. Please note that all calls for retrieving a wrapper are always of the form `outerClassInstance.W(constructorargs)` and not simply `W(constructorargs)`. We apply the scoping rules common for Java nested classes also to wrapper recycling, meaning that the call `outerClassInstance.W(constructorargs)` can be shortened to `W(constructorargs)` if the scoping is clear. Wrapper creation relative to an enclosing instance refers to the fact that `W` is a virtual type whose meaning can only be determined in the context of an instance of the enclosing class. This is in the vein of family polymorphism [6]. Please note that the family polymorphism interpretation of virtual types does also remove the covariance problems that are usually associated with virtual types, see also [6] and [22].

As introduced in Sec. 3.1, implementations of the two contracts (components and connectors) of a collaboration interface can be freely combined with the "+" operator. This is illustrated in Fig. 17, where two different complete realizations of the collaboration interface `GraphColoring` are defined. Both combinations, `Scheduling1` and `Scheduling2`, are subtypes of `SchedulingGraph` and can therefore uniformly be used. This allows writing the operation `computeScheduling` polymorphically with respect to the coloring algorithm used. The code in Fig. 17 illustrates this kind of polymorphism. `Scheduling1` and `Scheduling2` differ from each other on the coloring algorithm. One could also think of combining the same coloring algorithm with two different remodularizations of the university administration example. An example for this kind of polymorphism can be found in Fig. 18 and Fig. 19.

Fig. 18 shows three other remodularizations of the university

```
class Scheduling1 =
   SuccessiveAugmentationColoring + SchedulingGraph;
class Scheduling2 =
   SimulatedAnnealingColoring + SchedulingGraph;
...
ColoredGraph cg =
    wantScheduling1 ? new Scheduling1() : new Scheduling2();
computeScheduling(cg, someCourses);
...
void computeScheduling(SchedulingGraph sg, Course courses[]) {
  sg.computeMinimumColoring( sg.CourseVertex[](courses));
}
```

Figure 17: Demo code

```
class StudentContactsGraph binds Graph {
  class StudentVertex binds Vertex {
    final Student s;
    StudentVertex(Student s) {this.s=s;}
    Edge[] edges() {
      ... Student t = ... ;
      ... e[i] = StudentContact(s,t);
      return e;
    }
  }
  class StudentContact binds Edge {
    Student s,t;
    StudentContact(Student s, Student t) {
      this.s = s; this.t = t;
    }
  }
}
class StudentContactsColoring extends StudentContactsGraph
                              binds ColoredGraph {
  override StudentVertex {
    void setColor(int c) {
      Exam.joinGroup(s,c);
    }
  }
}
class StudentContactsMatching extends StudentContactsGraph
                              binds MatchedGraph {
  override StudentContact {
    void setMatched(boolean b) {
      Rooms.getFreeApartment().assignStudents(s,t);
    }
  }
}
```

Figure 18: Alternative bindings of `ColoredGraph` and `MatchedGraph`

```
class A =
   SuccessiveAugmentationColoring + StudentContactsColoring;
class B =
   SimulatedAnnealingColoring + StudentContactsColoring;
class C =
   MatchingAlgorithm1 + StudentContactsMatching;
class D =
   MatchingAlgorithm2 + StudentContactsMatching;
```

Figure 19: Free combination of components and connectors

application and illustrates another nice feature of our model that is due to the seamless integration of our new concepts into the standard object-oriented concepts of classes, inheritance and subtype polymorphism. The classes defined in Fig. 18 remodularize the university application to present the student contacts graph: The students are the vertices and two vertices are connected if the students visit a joint course. The class `StudentContactsGraph` represents the general remodularization to the `Graph` collaboration, while `StudentContactsColoring` and `StudentContactsMatching` refine this class in order to specialize the collaboration to `ColoredGraph` and `MatchedGraph`[8], respectively. Inheritance allows us to reuse the `StudentContactsGraph` connector in the definition of both `StudentContactsMatching` and `StudentContactsMatching`. Similar to the extension of nested interfaces in Fig. 14, the `StudentVertex` and `StudentContact` bindings of `StudentContactsGraph` can be refined with an `override` declaration, as with `StudentVertex` in `StudentContactsColoring`.

A minimum coloring in the student contacts graph would represent maximum groups of students that do not know each other and would therefore be good candidates for joint exams with little cheating opportunities. A maximum matching, on the other hand, would be helpful to assign the students to two person apartments, such that most students are pooled together with a person they know. The `StudentContactsColoring` connector can be directly combined with any of the coloring components introduced in Fig. 15. Similarly, `StudentContactsMatching` can be combined with an arbitrary matching algorithm that implements `MatchedGraph` (adumbrated in Fig. 19).

3.3 Summary of the Features

To summarize the chapter, we want to recall the different dimensions of polymorphism and reuse that are possible in our approach:

- **Collaboration interface dimension**: A hierarchy of collaboration interfaces can be defined, such as the `Graph` interface which is refined by `ColoredGraph` and `MatchedGraph` (see Fig. 14).
- **Component dimension**: Multiple independent implementations of a collaboration interface are possible, such as the different coloring implementations in Fig. 15. Component implementations can reuse other component implementations to implement more specialized collaboration interfaces via inheritance. For example, the communalities of the two coloring algorithms in Fig. 15 could have been outsourced into a common superclass.
- **Connector dimension**: Multiple independent bindings of a collaboration interface to the same or different applications can co-exist, such as the course collision and the student contacts remodularizations in Fig. 16 and 18, respectively. Inheritance among connectors, such as in Fig. 18, allows to reuse existing remodularization specifications when binding more specialized collaboration interfaces.
- **Bound component dimension**: The bound component is a subtype of both the component and the connector type. Therefore, client code, such as the `computeScheduling()` method in Fig. 17, can be reused with any implementation.

[8]The code for the `MatchedGraph` collaboration interface and its implementation are not shown but are analogous to the coloring example

4. FUTURE WORK: TOWARDS FLUID AOP

This paper represents a stable snapshot of an ongoing work. In this section, we want to outline the next steps we tackle.

Our future work on this model heads for *fluid aspect-oriented programming*, a term recently coined by Gregor Kiczales. In [11], he writes:

> "But we can also see signs of a next generation of AOP technology, that we call fluid AOP. Fluid AOP involves the ability to temporarily shift a program (or other software model) to a different structure to do some piece of work with it, and then shift it back."

We think that due to its expressiveness concerning on-demand remodularizations, our approach is a good starting point for fluid AOP. Our idea of enabling fluid AOP in our approach is based on two related concepts: *callback methods* and *callback activation*.

Fig. 20 illustrates our ideas. It shows code for online scheduling, that is, a new schedule is automatically computed every time an assignment of teachers or student years to courses changes.

The method `courseAssignmentChanged()` in `SchedulingGraph` is a sample callback method. A callback method describes events (or *joinpoints*) that should lead to the execution of that method. For example, the definition of `courseAssignmentChanged()` states that this method should be called after the methods `assignCourse()` or `addRequiredCourse()` have been called on objects of type `Teacher` and `StudentYear`, respectively. However, the pure declaration of such a callback method does not have any computational effects. The callback methods have to be *activated* by means of an `apply` block. An example can be found in the lower part of Fig. 20. An `apply` block is configured with an object whose definition contains callback methods, such as the `sg` object in Fig. 20.

```
class SchedulingGraph binds ColoredGraph {
  // Remainder as in Fig. 15
  Course[] courses;
  SchedulingGraph(Courses c[]) { courses = c;}
  void courseAssignmentChanged(Course c)
    after calls(Teacher.assignCourse(c)) ||
         calls(StudentYear.addRequiredCourse(c)) {
      computeMinimumColoring(courses);
  }
}
...
// Scheduling1 and Scheduling2 as in Fig. 16
if (wantOnlineScheduling) {
  SchedulingGraph sg = wantScheduling1 ?
    new Scheduling1(courses) : new Scheduling2(courses);
  apply (sg) in {
    startCourseAssignment(courses, teachers, studentyears);
  }
} else {
  startCourseAssignment(courses, teachers, studentyears);
}
```

Figure 20: Fluid AOP with On-Demand Remodularization, Callbacks, and Callback Activation

The semantics of the `apply` block is that the callbacks defined in `sg` are active during the execution of everything that is inside the `apply` block - in this case, the call to `startCourseAssignment()` and everything that is transitively called in that method.

With respect to the idea of fluid AOP, the application is transformed to perform online scheduling during the execution of a method call, and after the execution, everything is shifted back.

Please note the dynamics that is involved in this process. First, we can choose at runtime whether we want to perform online scheduling or not (depending on the runtime value of the `wantOnlineScheduling` variable. Second, if we decided to use online scheduling, we can still choose from different scheduling (respectively coloring) algorithms which we want to use (exemplified by the decision between `Scheduling1` and `Scheduling2`).

The full details of this approach - for example, the exact syntax for specifying events and options for on efficient implementation - have not yet been worked out. Yet we are confident that we will be able to answer all open questions in the next time and that the prospects of this approach are worth the trouble.

5. RELATED WORK

Pluggable Composite Adapters (PCAs) [19] and their predecessor, *Adaptive Plug and Play Components (APPCs)* [18], 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.

Due to the lack of decoupling of the component implementations from their bindings, the connectors and adapters in APPC and PCA models are bound to a fixed component. Furthermore, the lack of the notion of virtual types is another drawback of these approaches as compared to the work presented here. In addition, both approaches rely on a dedicated mapping sublanguage that is less powerful than our notion of object-oriented wrappers with wrapper recycling. Among these approaches, the APPC model of remodularization is a class-based one, and only PCAs share the object-based on-demand remodularization with our approach.

In [7], a variant of the PCA construct, called *dynamic view connectors (DVCs)* is used in the architecture of an integrated software engineering environment to support the late integration of independently developed software engineering tools. This work demonstrates the power of on-demand remodularization in a real-world, fairly large system. By being basically a realization of the PCA concept, DVCs also share their shortcomings mentioned above.

The Hyperspaces model and its instantiation in Hyper/J [20] also support on-demand remodularization - this notion was actually first introduced by the Hyperspaces model. In Hyper/J, one can define an independent component in n hyperslice. Hyperslices are independent of their context of use by the feature of being declaratively complete, i.e., they declare as abstract methods everything that they need, but cannot implement themselves. A hyperslice is integrated into an existing application by means of composition rules. As the result, new code is generated by mixing the hyperslice code into the existing code. Hyperslices can also be generated by selecting code units from an existing application. This is how on-demand remodularization is supported in Hyper/J. Similar to PCAs, Hyper/J also lacks the notion of collaboration interfaces and the late binding related to it. Hyper/J's remodularization approach is class-based, and can therefore not be applied to individual objects. Furthermore, Hyper/J's sublanguage for remodularization and mapping specifications is fairly complex and not well integrated into the common OO framework.

At this point, the question rises of how to position the work presented here with respect to previously published works on *collaboration-based decomposition* (CBD). CBD approaches aim at providing modules that encapsulate a whole collaboration of classes. With CBD classes are decomposed into the roles they play in the different collaborations. The idea is nicely visualized by a two dimensional matrix with the classes as the column indexes and collaborations in which these classes are involved as the row indexes.

Mixin Layers [25] and delegation layers [22] are two representatives of approaches to CBD. Both approaches provide concepts for composing and decomposing a collaboration into *layers*, such that a particular collaboration variant can be obtained by composing the required layers. Mixin layers use a nested variant of mixin-inheritance [4], whereas delegation layers combine delegation and virtual classes in

order to defer the layer combination until runtime. None of these approaches support on-demand remodularization. The definition of a collaboration layer in these approaches also encodes how the collaboration will be integrated. The vocabulary of abstractions that are involved in an application is defined a-priori to the definition of any collaboration layer and is consequently shared by all layer definitions.

Collaboration layers are especially useful in case we have many different variants of a particular collaboration (for example, `Graph`, `ColoredGraph`, and `WeightedGraph`) and want to mix-and-match these variants at runtime (for example, create a `ColoredWeightedGraph` by composing the color and the weight layer). Collaboration layers complement the concepts proposed in this paper very nicely, because they would allow us to decompose both components and connectors into layers that could be combined on-demand. In the future, we plan to combine the dynamic composition features of the delegation layers approach with the concepts of the work presented here.

VanHilst and Notkin propose an approach for modelling collaborations based on templates and mixins as an alternative to using frameworks [27]. However, this approach may result in complex parameterizations and scalability problems. A *contract* [9] allows multiple potentially conflicting component customizations to exist in a single application. However, contracts do not allow conflicting customizations to be simultaneously active. Thus it is not possible to allow different instances of a class to follow different collaboration schemes.

Seiter et al [23] proposed a *context relation* to link the static and dynamic aspects of a class. While supporting multiple dynamic collaboration schemes, the approach is based on dynamically altering a class definition for the duration of a method invocation, thus affecting all class instances, whereas we propose a model for scoping the different collaboration schemes, thus we can be selective as to which objects are affected.

Hölzle [8] analyses some problems that occur when combining independent components. Our proposal can be seen as an answer to the problems and challenges discussed in [8]. Mattson et al [16] also indicate the problems with framework composition, analyze reasons for these problems and investigate the state of the art of available solutions. In [3], Bosch proposes a language construct for specifying a class as the adapter of another class, that is, for explicit expression of the adapter pattern. The adapter construct as proposed in [3] has two main restrictions: First, it does not support adaptation of entire collaborative functionality. Second, as indicated in [3], it does not allow interface incompatibility.

Our work is also related to *architecture description languages* (ADL) [24], for example Rapide [13], Darwin [15] and C2 [17]. 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 ADL, components also describe their functionality and dependencies in the form of required and provided interfaces. 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.

6. SUMMARY

This paper proposed language concepts that facilitate the separation of an application into independent reusable building blocks and the integration of pre-build generic software components into applications that have been developed by third party vendors.

A key element of our approach is the notion of *collaboration interfaces*, used to declare the type of generic components. Collaboration interfaces are nested interfaces, bundling several abstractions that together build up the concept world of a component type into a family of virtual types [6]. In addition to the 'client-from-server contract', expressed by standard interfaces, collaboration interfaces also capture what servers expect from potential client contexts in which they might be integrated, i.e., the server-from-client contract. The implementations of these two contracts are completely decoupled from each other.

The implementation of the second contract translates the abstractions and vocabulary of an existing code base into the vocabulary understood by a set of components that are connected by a common collaboration interface. This translation is called *on-demand remodularization*, since the translation is virtual and effective only during the execution of functionality in the collaboration interface, whose server-from-client contract is implemented by the remodularization. Our approach to remodularization is object-based and uses the full computational power of an object-oriented language. The concept of *wrapper recycling* was additionally introduced to support the specification of the remodularization.

The decoupling of component implementation from bindings via remodularizations, allows to mix-and-match remodularizations and components on demand. The *+* operator was introduced for this purpose. This decoupling combined with the lean integration of collaboration interfaces with generalized notions of inheritance and subtype polymorphism, provide for a high degree of reuse in our model.

There are several areas of future work which were briefly outlined in the paper. First, we plan to combine our approach with concepts from the delegation layers approach [22]. This will allow us to decompose both components and connectors into layers that could be flexibly combined on-demand. Another very important area of future work is to extend our remodularization language with constructs for better supporting the integration of reusable crosscutting concerns, as outlined in Sec. 4. This is a promising step towards fluid AOP.

7. REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using

composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Programming*. Springer, 1993.

[2] L. M. Berlin. When objects collide: Experiences with reusing multiple class hierarchies. In *Proceedings of OOPSLA '90*, pages 181–193, 1990.

[3] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices 25(10)*, pages 303–311, 1990.

[5] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

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

[7] S. Herrmann and M. Mezini. PIROL: A case study for multidimensional separation of concerns in software engineering environments. In *Proceedings of OOPSLA 2000*. ACM SIGPLAN Notices, 2000.

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

[9] I. M. Holland. Specifying reusable components using contracts. In *Proceedings ECOOP '93, LNCS 615*, pages 287–308, 1992.

[10] Java Foundation Classes. http://java.sun.com/products/jfc/.

[11] G. Kiczales. Aspect-oriented programming - the fun has just begun. In *Workshop on New Visions for Software Design and Productivity: Research and Applications*, Vanderbilt University, Nashville, Tennessee, December 13-14, 2001.

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

[13] 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.

[14] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89*. ACM SIGPLAN, 1989.

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

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

[17] 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.

[18] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, 1998.

[19] 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.

[20] H. Ossher and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717), IBM Thomas J. Watson Research Center, 1999.

[21] H. Ossher and P. Tarr. On the need for on-demand remodularization. In *ECOOP'2000 workshop on Aspects and Separation of Concerns*, 2000.

[22] K. Ostermann. Dynamically composable collaborations with delegation layers. In *To appear in proceedings of ECOOP '02*, 2002.

[23] L. Seiter, J. Palsberg, and K. Lieberherr. Evolution of object behavior using context relations. In D. Garlan, editor, *Proceedings of the 4th ACM SIFSOFT Symposium on Foundations of Software Engineering*, pages 46–56. ACM Press, 1996. Software Engineering Notes 21(6).

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

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

[26] 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.

[27] M. VanHilst and D. Notkin. Using role components to implement collaboration-based design. In *Proceedings OOPSLA 96*, 1996.