Implementing collaboration-based Designs using Aspect-Oriented Programming

E. Pulvermüller\textsuperscript{1} \quad A. Speck\textsuperscript{1} \quad A. Rashid\textsuperscript{2}
\textsuperscript{1}Wilhelm-Schickard-Institute for Computer Science, University of Tübingen
72076 Tübingen, Germany
{pulvermu|speck}@informatik.uni-tuebingen.de
\textsuperscript{2}Computing Department, Lancaster University
Lancaster LA1 4YR, UK
marash@comp.lancs.ac.uk

Abstract

The collaboration-based approach is a powerful means to develop modularised systems by stepwise refinement. A number of efficient implementation techniques for such collaboration-based systems have been proposed (e.g. mixin layers). In this paper we introduce an alternative, new approach to realise a collaboration-based design resulting in further improvements. Our approach is based on the well-known observation that the knowledge about inter-object collaborations cannot be localised within objects but cross-cuts many objects. Such cross-cutting concerns are effectively addressed by applying the separation of concerns principle. We have, therefore, employed Aspect-Oriented Programming (AOP) to build collaboration-based designs. In the paper we illustrate and discuss our Aspect-Oriented approach both for the horizontal (i.e. the collaborations) and vertical (i.e. the refinements) dimensions. Although the example implementations are based on AspectJ0.4beta7 from Xerox PARC, the approach is generic enough to be implemented using other AOP techniques.

1. Introduction

The system decomposition concept pioneered by Parnas [33] has resulted in the development of numerous methodologies to decompose design problems into intellectually manageable, encapsulated and reusable pieces; the motivation being the decomposition promise to counter the inherent complexity of software. The object-oriented paradigm, for example, is based on this decomposition principle. However, the encapsulation and reuse mechanisms offered by object orientation are effective for small systems only and do not scale up to development of large software systems. These systems require higher level abstractions and larger units of encapsulation. As a result researchers and practitioners have observed that not only the fine-grained classes and objects should be of interest but also higher level abstractions and their interactions are essential for designing reusable modules encapsulating a largely orthogonal aspect of the design. Consequently supplementary techniques such as design patterns [18], frameworks [16, 34] and layered designs e.g. [9, 46] have been adopted.

This paper focuses on implementation techniques for a form of layered designs known as collaboration-based (or role-based) designs [9, 19, 20, 37, 46]. In a collaboration-based design an application is decomposed into a set of classes and a set of collaborations. Encapsulated within each application class are several roles. Each role manifests a particular aspect of the class’ behaviour. A co-operating suite of roles constitutes a collaboration.
Applications are defined using layered compositions of these collaborations. Since collaborations represent a distinct and largely independent aspect of an application they are an effective means to develop highly modularised designs.

Work has been carried out to preserve the modularity of collaboration-based designs at the implementation level. An approach based on mixin classes [11] was proposed by VanHilst and Notkin [43, 44, 42]. The proposed approach preserved the design modularity at the implementation level (as opposed to framework based implementation techniques e.g. [36]). However, scalability was an issue due to complex parameterisations in the presence of multiple classes and inability to contain intra-collaboration changes. These problems were addressed by Smaragdakis and Batary [7, 38, 39] who introduced mixin layers to implement collaboration-based designs. A mixin layer is similar to a mixin class with the difference that the former is scaled to a multiple class granularity. Although existing implementations of mixin layers offer a scalable solution to directly map collaboration-based designs onto implementation, like their peers the code handling the collaboration is spread across the various roles resulting in code tangling [24] and hence maintenance problems. Another shortcoming of the existing approaches is their reliance on parameterised inheritance resulting in consistency issues in C++ and usage of non-standard language extensions in Java. Also, extensive reliance on inheritance renders them unsuitable to implement collaboration-based designs in a distributed environment.

In this paper we propose an alternative approach to implement collaboration-based designs and therefore achieve a functionality similar to mixin layers. Our work shows how to overcome the difficulties of existing approaches. The proposed approach is based on the observation that the knowledge about inter-object collaborations cannot be localised within objects but cross-cuts many objects. Such cross-cutting concerns are effectively addressed by Aspect-Oriented Programming (AOP). Usage of aspects avoids code tangling hence improving maintainability. We also employ aspects to avoid the use of parameterised inheritance (and hence templates and template-like mechanisms). We describe two refinement mechanisms: one based on inheritance and another on composition. The next section provides an overview of collaboration-based design and use of mixin layers to implement such a design. We discuss the shortcomings of existing strategies for implementing collaboration-based designs and the reasons why AOP would be a suitable approach to achieve such functionality. We then illustrate and discuss our aspect-oriented approach. This is followed by a concrete example and a discussion of the related work. The example implementations are based on AspectJ0.4beta7 [48] from Xerox PARC and Java from Sun Microsystems. However, the approach is generic enough to be implemented using other AOP techniques.

2. Background

2.1. Collaboration-based Design

Although the object-oriented paradigm allows good structuring of the design, the units of encapsulation offered are rarely sufficient especially in the development of large software systems. This is due to the fact that the units of encapsulation, the objects, need to interact with each other to carry out a specific activity. Consequently, there is a need for higher level abstractions and larger units of encapsulation in order to strengthen the expressiveness of the design and improve reuse. This need is effectively addressed by collaboration-based design.
A collaboration-based design decomposes an application into a set of collaborations and a set of objects. A collaboration specifies a set of allowed behaviours for a group of objects cooperating to perform a specific activity. The part of an object that fulfills its responsibilities within a collaboration is referred to as its role. A collaboration is a co-operating collection of roles. An object can participate in a collaboration if it supports the required role. Objects normally participate in more than one collaboration and hence comprise of several roles. It should be noted that an object does not need to play a role in each collaboration in the application.

Figure 1 shows an example of a collaboration-based design. The horizontal dimension depicts the collaborations while the vertical dimension represents the objects. An intersection of the two dimensions depicts a role: a unit of design common to both dimensions. Four collaborations and three objects are shown in figure 1. Collaboration 2 contains two different roles: Role 2.1 and Role 2.2 which are encapsulated by Object 1 and Object 2 respectively. Collaboration-based design is a form of layered design as layered compositions of collaborations constitute an application.

A good collaboration-based design is the one where collaborations are largely independent of each other. Such a design offers a number of advantages. It allows expressing collaborations as explicit units of design as opposed to the more traditional use of classes as the basic building blocks. Other approaches such as design patterns [18] offer similar solutions e.g. behavioural patterns. However, for every collaboration there does not exist a pattern to be used. Secondly, patterns are a guide to rebuild collaborations aiming at reducing the cross-cutting collaboration code and focusing on encapsulation of classes. In contrast collaboration-based design accepts collaborations as individual design units.

From a refinement perspective if an object within a collaboration is replaced the new object does not necessarily need to belong to the same class as long as it supports the required role. Conversely, if a collaboration is changed or replaced the roles played by its participating objects in other collaborations stay unaffected.

From an evolutionary standpoint, roles encapsulate fewer decisions as compared to classes and hence are more stable [44].

2.2. Mixin Layers

The concept of mixin classes was introduced by Bracha [11]. A mixin class is a class whose superclass is not specified at the time of its definition. It eventually inherits from a superclass. Mixin layers scale the concept of mixin classes to multiple class granularity. They are “mixins encapsulating other mixins” [38] as shown in figure 2. The encapsulated mixins are referred to as inner mixins while the mixin encapsulating them is called the
outer mixin. Smaragdakis and Batory define a mixin layer as “an outer mixin where the parameter (superclass) of the outer mixin encapsulates all parameters (superclasses) of inner mixins” [38]. It should be noted that mixin layers can be nested; inner mixins being mixin layers themselves.

Mixin layers offer an effective, scalable way to directly map collaboration-based designs onto implementation. As shown in figure 2 and figure 3 collaborations map onto outer mixins while roles map onto inner mixins. As a result the structure of the design is preserved in the implementation. Furthermore, the higher level abstractions (the collaborations) offered are the same as mixin classes.

Mixin layers are an effective means for stepwise refinement [47] as the layers are reusable and exchangeable. Mixin layers have been successfully employed to build systems in GenVoca [6, 7, 8, 39]. The existing implementations for mixin layers [7, 38] are scalable as compared to earlier approaches e.g. the VanHilst and Notkin approach and those based on object-oriented frameworks.

The mixin layer implementations proposed by [7, 38] are by far the most effective means of realising a collaboration-based design. However, they have a number of disadvantages:

- The code handling the collaboration is spread across the various roles resulting in code tangling [24]. This results in serious maintenance problems. As the size of the collaboration grows intra-collaboration changes become very costly. In [38] Smaragdakis and Batory attempted to reduce the extent of this problem which existed, to a larger extent, in the VanHilst and Notkin approach [43, 44, 42]. However, they did not address the maintenance issues arising from cross-cutting code (handling the collaboration) spread across the various roles in the collaboration.

- The proposed implementations [7, 38] are based on parameterised inheritance. This results in reliance on templates in C++ and a requirement to use a non-standard template extension for Java and other languages. Smaragdakis and Batory questioned the effectiveness of C++ templates for implementing mixin layers in [38]. They identified the various consistency issues arising due to the lack of type-checking for templates and the “interpretive” behaviour of template programming. Although their proposed
Java language extension [6, 7] to support mixins and mixin layers addresses the issues of mixin integration (arising in C++) in a type-safe framework, the programmer is bound to a non-standard language extension.

- Although it is possible to employ this approach in a distributed environment [5], it has disadvantages in this field. The problem is that the refinement mechanism is based on inheritance which is not suitable for distributed systems [45] (CORBA, for example, does not define implementation inheritance [32]). There is a tight dependency of the subclasses on their base class which is contradictory to their distribution. The contracts between the components in an implementation hierarchy are implicit and thus not clearly defined. This problem is known as fragile base class problem [41]. As outlined in [40] use of inheritance can lead to unimplemented, inconsistent methods and conflicts in method interfaces.

2.3. Suitability of Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [24] offers a new programming paradigm aimed at achieving an improved separation of concerns. AOP aims at easing program development by providing further support for modularisation at both design and implementation levels. Objects are designed and coded separately from code that cross-cuts the objects. The cross-cutting code is encapsulated into separate constructs known as aspects. An aspect weaver is responsible for merging the objects and the aspects. This can be done statically as a phase at compile-time [22] or dynamically at run-time [22, 30].

AOP helps to avoid code tangling. Also, there is an improvement in reusability (of both object and aspect code). Predefined non-functional concerns can be flexibly exchanged and the application can be adopted to specific contexts, situations or cases [35].

Since collaborations are by nature cross-cutting aspects appear to be a natural choice for implementing collaboration-based designs. Code expressing the collaboration behaviour is tangled in all objects taking part in this collaboration. Usage of aspects enables to avoid code tangling hence improving maintainability. Aspects can also be employed to avoid using parameterised inheritance.

In the following section we propose our aspect-based approaches. Using our approaches the code handling the collaborations and the refinements shifts to a domain specific language (in our case an aspect-oriented language). This domain specific language is used to express the non-functional parts of the application e.g. how the various roles are linked. The functional code is expressed using an object-oriented language (or languages based on other paradigms) and the two are woven together to produce the application. This results in avoiding language specific mechanisms such as templates and non-standard language extensions.

3 The Aspect-Based Approaches

We first briefly revisit collaboration-based design before describing our approaches. All the roles in an application can be arranged as a matrix [12] as shown in figure 4. The horizontal dimension in the role matrix captures the collaborations between the objects while the vertical dimension captures the relationships between all the roles of one object. Existing implementation techniques for collaboration-based design provide a direct mapping from design onto implementation. Where collaborations are explicitly captured the vertical
interaction between the various roles in a collaboration is not explicitly identified. The code handling the interaction is spread across the various roles resulting in code tangling leading to lack of flexibility during intra-collaboration changes.

In this section we discuss our proposed implementations of collaboration-based designs. We first describe our approach to address these flexibility problems in the horizontal dimension. We discuss the separation of cross-cutting collaboration code from the roles participating in a collaboration into an aspect hence improving support for (horizontal) intra-collaboration changes. We then address the vertical dimension describing how aspects can be used to achieve refinements and separation of the interaction code in this dimension. We show two refinement mechanisms: one is based on inheritance as in [7, 38, 43, 44, 42] (but avoiding the use of parameterised inheritance and reliance on templates and template-like mechanisms) while the other is based on composition hence avoiding inheritance-based refinement (if desirable) and making the approach usable in distributed environments.

### 3.1. Horizontal Dimension (The Collaboration)

As shown in figure 5 the code for a role played by an object in a collaboration can be categorised as:

- code implementing the functional requirements
- code implementing the typical non-functional requirements (e.g., for synchronisation purposes)
- code implementing all cross-cutting requirements concerning the role interaction

Usually the code sections implementing the cross-cutting interaction behaviour form part of the code section implementing non-functional requirements. We, however, investigate this cross-cutting code category separately. From this point onwards this category will be the focus of our discussion. Instead of implementing the cross-cutting interaction code hard-wired within the roles participating in the interaction we propose implementing roles and
/** 
The Collaboration1 aspect links Object1_Role1_1, Object2_Role1_2 and Object3_Role1_3.
*/
aspect Collaboration1 {

// ----------------- section 1 -----------------
// introduction of variables necessary for a collaboration
introduction Object1_Role1_1 (Object2_Role1_2 object2_role1_2;)
introduction Object2_Role1_2 (Object3_Role1_3 object3_role1_3;)

// ----------------- section 2 -----------------
// introduction of collaboration methods:
introduction Object1_Role1_1 {
  public int collaborate1_method1();
}
introduction Object2_Role1_2 {
  public int collaborate1_method2();
}

// ----------------- section 3 -----------------
// advising on existing methods:

section 1: introduce new variables which are necessary to collaborate into existing classes

section 2: introduce new methods which contain collaboration code into existing classes

section 3: advise existing methods of existing classes (before or after)

Figure 6. Aspect Collaboration1

their main functionality (code handling functional and other non-functional requirements) separate from the interaction functionality. Since the main goal of AOP is to achieve better separation of concerns, all interaction related code can be eliminated from the roles and put into a separate aspect (as shown in figure 5). This serves a twofold purpose. First, the collaboration is not distributed across various roles but localised within an aspect. Therefore, it is easier to change or exchange the collaboration’s behaviour. Second, it allows dividing an object into roles which are primarily independent of the concrete collaboration they participate in. Therefore, these roles can be reused in other collaborations.

The code fragment in figure 6 depicts such a collaboration aspect (called Collaboration1). The aspect code has been written in AspectJ 0.4 beta 7 syntax [48] which is an aspect-oriented extension to Java. In AspectJ 0.4 beta 7, each aspect has (like classes) its name following the keyword aspect and contains the advised methods with their names and the class names referring to these methods. Besides method advising \(^1\), it is also possible to introduce (keyword introduction) whole methods or variables into existing classes.

\(^1\) By using the keyword before the aspect weaver injects this advised code at the beginning of the method. The keyword after augments the method at the end.
Braces “(...)” have been used to hide implementation details not relevant to the explanation of the concepts. In the implementation the name RoleX.Y is renamed to ObjectY.RoleX.Y \( \forall X, Y \in [N] \). The roles Object1.Role1.1, Object2.Role1.2 and Object3.Role1.3 are implemented as classes. The concrete implementation of these classes depends on the functionality required of each role in a certain problem domain. For the aspect implementation in figure 6 it has been assumed that for all ObjectN.Role1.N (where \( 1 \leq N \leq 3 \)) there exist constructors and that Object1.Role1.1 has a method called “start( )” (in our implementation this method is used to initiate the collaboration). This method has been advised in the Collaboration1 aspect.

To improve understandability the aspect itself has been structured and divided into three sections:

**Section 1** All helper variables are introduced into the roles. These are essential to store references to instances of other roles for collaboration purposes.

**Section 2** This section comprises of all the additional collaboration methods. These methods (their names, parameters and bodies) depend on the concrete collaboration. They call other methods and are themselves called to perform the interaction between the roles. These methods use the references to other roles introduced in section 1.

**Section 3** This code section addresses all points in the execution of the program where further behaviour has to be added (before or after) to existing methods as a consequence of the collaboration. The augmentation of the constructors (indicated with keyword new) is typical since each object needs to create or get references to the objects it is collaborating with. All other advising depends on the problem domain and the concrete collaboration behaviour.

As discussed earlier using aspects (for the horizontal dimension) to encapsulate all interaction related code provides more flexible collaborations overcoming the shortfalls of existing approaches.

### 3.1.1. Vertical Dimension (The Refinements)

#### 3.1.2. First Approach: Inheritance-based refinement

The motivation behind our first proposal is to provide an alternative implementation technique for inheritance-based refinement without using parameterised inheritance hence removing the need to rely on templates or template-like constructs. We employ aspects to realise both inner and outer mixins ensuring that we do not violate the actual definition of a mixin. We propose two techniques to implement mixins. Both of them allow to implement collaboration-based designs by providing an alternative implementation for mixin layers:

First, the inheritance relationship can be inserted using a weaver. This scenario is shown in figure 7. It should be noted that the notation used in figure 7 is UML. The parts which are realised using aspects are shown in brackets according to the notation introduced by [29]. The initial code for Role 3.1 will be as follows:

```java
class Role3_1 { ( ... ) }
```

This definition is augmented by an inheritance relationship to the superclass Role 2.1 by a weaver resulting in:

```java
class Role3_1 extends Role2_1 { ( ... ) }
```
Since the superclass of Role 3.1 is determined only at weave time (which can be a phase during compilation or run-time) it is a mixin class. Weaving of the extends is possible using a weaver based on a parser for Java allowing manipulation of Java parse trees. JTS [6], for example, offers such a parser.

Our second technique (for inheritance-based refinement) to implement mixins is based on the observation that in ordinary inheritance the knowledge about the inheritance relationship is fixed within the subclass code. Using templates to realise mixins does not remove this knowledge from within the subclass but parameterises it hence making the superclass exchangeable. We propose introducing an intermediate class which separates the knowledge about the connection between the subclass and its superclass. This is shown in figure 8. Since the intermediate class has the sole purpose of achieving separation of concerns we call it an aspect class. Aspect classes are different from ordinary classes. Although they can be implemented using the class construct 2, they behave differently when supported by a weaver and can be exchanged at weave time.

As shown in figure 8 Role 3.1 inherits from the aspect class Intermediate (2_3).1. This aspect class itself inherits from Role 2.1. At weave time this aspect class can be exchanged which results in an exchange of the superclass of Role 3.1. Role 3.1 is, therefore, a mixin class. As shown in figures 7 and 8 our proposed techniques can be scaled to implement mixin layers. In the aspect class based implementation we call the aspect separating the mixin layers an aspect layer. This is because such an aspect class comprises of other aspect classes (implemented as inner classes). Exchanging the aspect layer (using a weaver) results in an exchange of mixin layers. Some sample code showing two collaborations is shown below:

class Collaboration2 extends Intermediate1_2 {
    class Role2_1 extends Intermediate1_2.Role1 { ( ... ) }
    class Role2_2 extends Intermediate1_2.Role2 { ( ... ) }
    class Role2_3 extends Intermediate1_2.Role3 { ( ... ) }
}
class Intermediate2_3 extends Collaboration2 {

2In our implementation we actually used the class construct to realise aspect classes.
class No1 extends Collaboration2.Role2_1 { ( ... ) }
class No2 extends Collaboration2.Role2_2 { ( ... ) }
class No3 extends Collaboration2.Role2_3 { ( ... ) }
}
class Collaboration3 extends Intermediate2_3 {
class Role3_1 extends Intermediate2_3.No1 { ( ... ) }
class Role3_2 extends Intermediate2_3.No2 { ( ... ) }
class Role3_3 extends Intermediate2_3.No3 { ( ... ) }
}

It should be noted that in the implementation the terms are slightly renamed: Class Intermediate (2_3).1 in figure 8, for example, is represented by Intermediate2_3.No1 in the code above. Furthermore, it should be noted that in the above code classes Intermediate2_3.No1, Intermediate2_3.No2 and Intermediate2_3.No3 have been implemented as inner classes No1, No2 and No3 in the aspect class Intermediate2_3.

The two proposed techniques to realise refinements are very close to the implementation from [7, 38]. They only differ in the respect that the mechanism to implement mixins is not parameterised inheritance but aspects. This has a number of advantages:

- The template constructs are replaced by AOP mechanisms. As a result the proposed implementation is not bound to programming languages supporting templates or template-like constructs. This helps avoid the consistency checking problems when using templates in C++. It also avoids using non-standard template extensions to Java and other languages.

- AOP allows limited capabilities for transforming\(^3\) (and, therefore, extending and refining) existing applications. It, however, does offer meta-programming capabilities i.e. it provides facilities to write programs that manipulate other programs. Therefore, although not very powerful as a transformation system, AOP is more powerful than templates (whose primary use is to parameterise code fragments). As shown in the above discussion AOP is suitable and sufficient for realising mixins and mixin layers. Therefore, the complexity of using a transformation system is avoided\(^4\).

- AOP allows explicitly expressing the connection between subclasses and superclasses (and sublayers and superlayers) in a separate language. Although this seems to be a disadvantage at the first glance, this is not the case. The application classes and components are implemented in a programming language suitable for the specific problem domain. This language, however, is not necessarily suitable for expressing the connections between the various parts of the application. An aspect-oriented language is employed to implement these connections and relationships. In this context the AOP language serves as a domain specific language suitable for formulating the knowledge about the connections between various parts of the application.

Although our proposed approach overcomes the shortcomings of using parameterised inheritance to realise mixins and mixin layers (and hence refinements), inheritance being the underlying refinement mechanism, the approach is not suitable for implementing collaboration-based designs in a distributed environment. This shortcoming formed the motivation behind our second proposal.

---

\(^3\)There are rules as to where and what can be transformed.

\(^4\)Although AOP is sufficient we do not claim that the usage of a transformation system will not be useful in this context.
3.1.3. Second Approach: Composition-based Refinement

Our second approach is based on the observation that refinements (the vertical dimension) are subject to a number of constraints. In existing realizations of collaboration-based design (including our first approach) these constraints are met implicitly using inheritance. Consider for example Role 1.1 and Role 2.1 in figure 1. Role 2.1 can use Role 1.1 but the converse is not true. Role 1.1 cannot access the methods and data of Role 2.1. The more abstract role provides functionality to the more concrete roles but not vice versa. In our proposal these constraints are explicitly captured in a refinement aspect. Instead of using inheritance with its implicit contract between subclass and superclass, an explicit contract (a refinement aspect) is defined for each pair of related roles. The same is done for the inheritance relationships between the collaboration layers. These refinements have been shown in figure 9. The refinement aspects are shown in the figure as compositions (UML notation) encircled by brackets (keeping with the notation from [29]).

The explicit contracts conform to the properties of inheritance relationships at the same time avoiding the disadvantages of inheritance [40, 41] (e.g. usability in a distributed environment [45]). Furthermore, the approach is more flexible since the constraints can be changed (relaxed or strengthened) if necessary.

It should be noted that all refinement aspects follow a common implementation pattern since they are all built using a common algorithm derived from the defined constraints. We now describe the algorithm we have employed. This algorithm is similar to the ones employed by some programming languages to implement inheritance. The algorithm is as follows:

1. Introduce into this role (the sub-role) a reference to the neighboring role in the layer above (the super-role). For example, from figure 9, introduce a reference to Role 1.1 in Role 2.1.

2. Introduce into the super-role public accessor methods for all protected members i.e. introduce into Role 1.1 (figure 9) accessor methods for all protected members. This is required to provide access to protected members from the role in the layer below (to capture the inheritance constraints). Security is an issue in this case. However, this is not a major problem as only the sub-role is aware of these public accessor methods.

---

*This is a conceptual decision made at the beginning depending on the problem domain and kept afterwards. In this paper we define constraints according to the properties of an inheritance relationship.*
Additional security can be provided using an additional parameter or other suitable techniques.

3. Augment the constructor of the sub-role with code to instantiate the super-role. For example (figure 9) advise the constructor of Role 2.1 with code to create an instance of Role 1.1.

4. Augment the sub-role with methods to access the public and protected members of the super-role. For example (figure 9) advise all those accessor methods within Role 2.1 which intend to access or refine members of Role 1.1.

The algorithm is also depicted by the refinement aspect listed in figure 10. It should be noted that the above algorithm can be replaced with another one to enforce different constraints or even the same constraints. From the above discussion it is obvious that there are various explicit contracts in the proposed approach. Where the number of contracts in the horizontal dimension is not larger than the ones in earlier approaches, the number of contracts in the vertical dimension is substantially larger. This increase is due to the previously implicit contracts (due to inheritance relationships) being made explicit. The total number of pair-wise contracts in the vertical dimension can be calculated as:

\[ n \times (m - 1) \]

where \( n \) is the total number of objects in the application and \( m \) is the number of roles encapsulated by an object.

Explicit contracts have the advantage of improved flexibility and independence and the disadvantage that all contracts have to be managed explicitly. Our approach addresses the contract management problem by offering a possibility to automatically generate the refinement aspects based on an algorithm (as discussed above) describing how such an aspect is derived from an existing hierarchy of roles.

Making contracts explicit provides more independence and flexibility to the roles in the vertical dimension (as is the case with our approach in the horizontal dimension) resulting in improved reusability and a better separation of concerns.

Due to the close connection in an inheritance relationship distributed environments often do not support implementation inheritance between classes residing on different locations [45] (CORBA [32], for example, defines only interface inheritance and no implementation inheritance). Since this approach to refinement is not based on inheritance it can be employed to realise collaboration-based designs in an environment where roles are distributed.

4. Concrete Example

In this section we describe a collaboration-based design for a simple calculator. This design has been realised using our aspect-based approach.

The role matrix for the calculator application is shown in figure 11. Only the most relevant methods and attributes for the roles have been shown. The application consists of three collaborations. Collaboration C1 is the most abstract layer and maps an architectural pattern (the Model-View-Controller pattern [13, 25]) to roles. Although an architectural pattern does not deal with implementation, in our case it swiftly maps onto the implementation (hence reflecting the structure of the pattern within the implementation).

In figure 11 the Model handles all the data. If new data is stored the model updates the View and the Controller. The View collaborates with the Model to obtain new data for

---

6The Model-View-Controller has been simplified for this example.
```java
/**
 * The Object1_Role1_Role2 aspect combines Object1_Role1_1 with Object1_Role2_1
 * in one "object" - defines a contract and replaces inheritance.
 */

aspect Object1_Role1_Role2 {
    // ----------------- section 1 --------------------------------------------------
    // introduction of variables necessary for an interaction:
    introduction Object1_Role2_1 { Object1_Role1_1 super1_role1_1; }
    introduction Object1_Role1_1 { static Object1_Role1_1 thisInstance; }

    // ----------------- section 2 --------------------------------------------------
    // introduction of methods to access the members (methods and data) of the
    // "super-role" Object1_Role1_1.
    // Here, it is assumed that class Object1_Role1_1 has a protected attribute
    // of type integer called attribute1 and no protected methods.
    introduction Object1_Role1_1 {
        public void set_object1_role1_1_attribute1 (int _int_value) {
            attribute1 = _int_value;
        }
    }

    // ----------------- section 3 --------------------------------------------------
    // 1. advising on constructors of role 2.1:
    static advice Object1_Role2_1 & new(..) {
    }
    // 2. advising all other methods of the role 2.1 to enable access
    // to methods of role 1.1:
    static advice Object1_Role1_1 & int role2_1_method1() {
        before { (...) }
    }

    // end of aspect Object1_Role1_Role2
}
```

**Figure 10. Refinement aspect for contract between Object1_Role1_1 and Object1_Role2_1**
displaying purpose. The Controller handles new events and collaborates with the View e.g. to display messages and with the Model to store new data.

The second layer (Collaboration C2) refines the first one by realising an abstract calculator. The Calculation Model role in this layer processes all calculations. The View role in C1 is refined to a Numeric Display in this layer while the Controller is refined to Calculator Input Controller which handles specific events e.g. number input, sign input, calculation request, etc.

In the bottom layer the Calculation Model is refined to Basic Arithmetic (which determines, for instance, how to implement the algorithm for the add method). The Calculator Input Controller is refined to Mathematical Statement Parser which reads all input from the device and parses it.

By means of this example we demonstrate our approach based on refinement by composition. The code for the collaboration aspect for collaboration C1 is shown in figure 12. The various refinement aspects of object O1 have been combined into one aspect (due to space limitations) in figure 12. Where this is a valid option we suggest using one aspect per contract in order to keep the various roles in an object loosely coupled.

The code fragments in figures 12 and 13 also identify a potential problem. Since roles are instantiated in both the collaboration aspect and the refinement aspect different instances of the roles would exist in the two dimensions. We have countered this problem by allowing the refinement aspect to instantiate the roles. As detailed in figures 6 and 10 (and indicated in figures 12 and 13) we have applied the Singleton design pattern [18] to check if a role already exists when a collaboration aspect attempts to instantiate it. If the role has already been instantiated a reference to the already existing instance is returned (instead of a creating a new instance). The Calculator application is built by choosing the parts (which are reusable), i.e. the appropriate objects or roles, and the appropriate contracts (expressed in aspects) and weaving them. The developer has the flexibility to exchange

---

**Figure 11. Collaboration-based design for a calculator application**

<table>
<thead>
<tr>
<th>Role Matrix</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C1</strong> Model</td>
<td>Object[] data</td>
<td>Object[] view_data</td>
<td>Object[] event</td>
</tr>
<tr>
<td></td>
<td>store_data(Object[] o) update_view()</td>
<td>display()</td>
<td>event handle_event()</td>
</tr>
<tr>
<td><strong>C2</strong> Calculation Model</td>
<td>Calculation Method</td>
<td>Numeric Display</td>
<td>Calculator Input Controller</td>
</tr>
<tr>
<td></td>
<td>int result calculation_method() add(Object[]) sub(Object[]) mult(Object[]) div(Object[])</td>
<td>display_number()</td>
<td>event_number_input() event_sign_input() event_do_calculate()</td>
</tr>
<tr>
<td><strong>C3</strong> Basic_Arithmetic</td>
<td>add(int x, int y) sub(int x, int y) mult(int x, int y) div(int x, int y)</td>
<td>Mathematical_Statement Parser</td>
<td>char raw_input read(char x) parse()</td>
</tr>
</tbody>
</table>

Horizontal Dimension
/**
 * The MVC aspect links Model, View and Controller.
 */

aspect MVC {
    // ---------------- section 2 -----------------------------------------
    introduction Controller { Model model; View view; }
    introduction Model { View view; }
    introduction View { Model model; }
    // ---------------- section 3 -----------------------------------------
    introduction View { new(Model m) { model = m; }; }
    introduction Model {
        public void set_view(View v) { view = v; }
        public void update_view() { view.display(); }
    }
    static advice Controller & new(..) {
        before {
            Model model = new Model();
            model = model.makeInstance(); // avoid double instantiation
            View view = new View(model);
            view = view.makeInstance();
            model.set_view(view);
        }
    }
    static advice Model & public void store_data(Object o) {
        after { thisObject.update_view(); }
    }
    static advice View & public void display() {
        before { view_data = model.read_data(); }
    }
    static advice Controller & public void Handle_event() {
        after { model.store_data(event); }
    }
}

Figure 12. A collaboration aspect for the calculator application (Collaboration C1)
/**
 * The Calculation aspect combines Model, Calculation_Model and Basic_Arithmetic.
 */

aspect Calculation {
    // ----------------- section 1 -----------------------------------------------
    introduction Basic_Arithmetic { Calculation_Model calculation_model; }
    introduction Calculation_Model { Model model; }
    // static variables to avoid double instantiation (see fig. 10)
    introduction Model { static Model thisInstance; }
    introduction Calculation_Model { static Calculation_Model thisInstance; }
    // ----------------- section 2 -----------------------------------------------
    // there is no protected data or method in one of the roles;
    // methods to avoid double instantiation (Singleton Pattern, see fig. 10)
    introduction Model {
        public static Model makeInstance() {
            if(thisInstance == null) {
                thisInstance = new Model();
            }
            return thisInstance;
        }
    }
    introduction Calculation_Model {
        public static Calculation_Model makeInstance() {
            if(thisInstance == null) {
                thisInstance = new Calculation_Model();
            }
            return thisInstance;
        }
    }
    // ----------------- section 3 -----------------------------------------------
    static advice Calculation_Model & new(..) {
        before {
            model = new Model();
            model = model.makeInstance();
        }
    }
    static advice Basic_Arithmetic & new(..) {
        before {
            calculation_model = new Calculation_Model();
            calculation_model = calculation_model.makeInstance();
        }
    }
    static advice Calculation_Model & void calculator_method() {
        after { model.store_data(thisObject.result); }
    }
    static advice Basic_Arithmetic & int add(int x, int y) {
        after { calculation_model.add(thisJoinPoint.parameters); }
    }
    // the same for all other arithmetic methods
}

Figure 13. A refinement aspect for the calculator application (Object O1)
objects, collaborations or individual roles. To exchange these parts it is only necessary to exchange the aspects (which capture the knowledge about the connection between these parts) and weave the application. Using the type equations of the GenVoca model [4] the Calculator application can be defined as:

```plaintext
Calculator = C1 [ C2 [ C3 ] ]
```

This differs only syntactically compared to the way the application can be woven using the AspectJ0.4beta7 weaver [48]:

```plaintext
ajc @C1 @C2 @C3
```

Since there are no close connections but explicit contracts between the parts of the Calculator, these parts can also be distributed in a network.

5. Discussion and Related Work

Pioneering work to directly map collaboration-based designs onto implementation was carried out by VanHilst and Notkin [43, 44, 42]. Where the approach (based on mixin classes) offered significant improvements over existing framework-based implementation techniques, not only each role was parameterised by a super-role but also by the roles it needed to collaborate with. This resulted in scalability problems due to complex parameterisations and deep nesting of templates and an inability to contain intra-collaboration changes. Smaragdakis and Batory [7, 38] addressed these problems by introducing mixin layers (scaling mixin classes to a multiple class granularity). Where the approach addressed some of the scalability problems of VanHilst and Notkin’s approach intra-collaboration changes were still expensive due to the code handling the collaboration being spread across the various collaborating roles. Being based on parameterised inheritance (similar to VanHilst and Notkin’s approach) the work relied on templates or template-like mechanisms resulting in consistency issues in C++ and usage of non-standard extensions to Java. Work similar to mixin layers was carried out by Findler and Flatt [17] who employed mixins and units in MzScheme to directly map collaboration-based designs onto implementation. The work allowed merging and, therefore, extending the separate units in the same fashion as inheritance but with further advantages. Where connections between modules or classes were specified separately from their definitions, module or object interactions were not addressed. The approach only addressed the evolution of modules, units and classes and did not consider the interaction between the parts and the impact of evolution on this interaction. Reliance on inheritance as the only refinement mechanism also led the above approaches to be unusable in distributed environments. Our approach in contrast to the above-mentioned approaches provides an effective separation of concerns, both for collaborations and refinements. The code handling a collaboration is extracted into an aspect and woven together with the roles during a compile-time phase (or even at run-time if dynamic aspects are used). This makes intra-collaboration changes easy and less expensive. Our approach also removes the reliance on parameterised inheritance and hence templates and template-like mechanisms. We propose two refinement mechanisms, one based on mixin inheritance (using aspects) and another based on composition. The later can be employed to realise collaboration-based designs in distributed environments.

Although the proposed concepts have been demonstrated using Java and AspectJ0.4beta7 [48] the approach is generic. The only invariant is the use of AOP. If an appropriate as-
pect language and a weaver are available the proposed concepts can be realised in other programming languages. Such tools, for example, exist for Smalltalk [10]. Similarly, the proposed concepts can be realised using any AOP technique allowing introduction of new functionality or changing existing functionality. Composition filters [1, 2], for example, can be employed to realise the same functionality. In this approach an object is extended with input and output filters. The approach can, therefore, be used to define interactions in both horizontal and vertical dimension.

The proposed approach has a strong basis in refinements [47, 3] and aspect-oriented programming [24]. We employ aspects to provide an approach for system development by stepwise refinement (in contrast to the more conventional use e.g. to inject cross-cutting tracing functionality). For our design with aspects we oriented by the design guidelines for aspect-oriented programming from [23]. All relationships between classes or roles and interaction aspects are class-directional. Our work also bears complementary links to work on synthesising objects supported by generators [4, 14]. Exploring these links is a fruitful area of research. Using explicit contracts for our composition-based refinement bears strong relations to the work on contracts described in [19, 20, 40]. Some related work on the design and implementation of role models using aspect-oriented programming can be found in [21]. Our approach captures knowledge about the connection between objects, roles, collaborations (i.e. between all parts of an application) in aspects in order to improve maintenance and reuse. [15] propose describing domain knowledge in aspects in order to separate this knowledge from the algorithms. Adaptive Programming [27, 28] supports software development, reuse and maintenance by means of traversal strategies. Instead of hard-wired structural knowledge paths within the classes, this knowledge is separated. Therefore, it can be useful in capturing the knowledge of the horizontal and vertical interaction paths in our approach. Some related work about the composition of applications consisting of both classes and aspects using adaptive plug and play components and aspectual components can be found in [26, 31].

6. Conclusions and Future Work

We have proposed an implementation technique for directly mapping collaboration-based designs onto implementation using aspect-oriented programming. The approach provides functionality similar to that achieved by using mixin layers. We have shown that in contrast to earlier work in the area our approach achieves a better separation of concerns and, therefore, better reuse. The code handling the collaborations and the refinements shifts to constructs expressed in a domain specific language (in our case an aspect-oriented language). Intra-collaboration changes are less expensive and reliance on templates and template-like mechanisms is removed. Besides, using composition-based refinement, the approach can be employed to implement collaboration-based designs in a distributed environment. We have presented realisation of a calculator based on our approach.

An area of future interest is the implementation of our approach using other programming languages and AOP mechanisms. Our approach is a contribution to the area of system development using stepwise refinement and explicit contracts. The theoretical foundations of contracts should be revisited in the light of expressing these contracts in a separate language construct supported by a weaver transforming these contracts into executable programs. While the generation of collaboration aspects is difficult to realise, the generation of the refinement aspects can be derived from the constraints and the role hierarchy. In
our future work we will investigate this generation of aspects (both for collaborations and refinements) and based on this we will explore the generation of product-line applications in a domain. Closely related is the area of composition validation in the context of our approach.

Acknowledgments. We would like to thank Don Batory, Ulrich Eisenecker and Ruzanna Chitchyan for inspiring discussions and helpful remarks on previous drafts of this paper. We also want to thank the AspectJ team for their support.

References


