# Component Frameworks for Software Generators

A. Speck[†] and E. Pulvermüller[◇]

[†]Wilhelm-Schickard-Institut für Informatik
University of Tübingen
D-72076 Tübingen
speck@informatik.uni-tuebingen.de

[◇]Universität Karlsruhe
Institut für Informatik
D-72076 Karlsruhe
pulvermueller@acm.org

**Abstract.** Software generators support the automatic or semi-automatic development of software. In this paper we focus on the basic model of generation and how such a basic architecture called component framework may be evolved. This architectural model defines the relations and interactions of software components on an abstract level. The specialization of this model may be done according to usual generation techniques like the GenVoca architecture.
Since the original mixin layer based implementation of GenVoca lacks of some drawbacks caused by the (mixin) inheritance we replace the inheritance by aggregation. This concept provides in general the same functionality like mixin inheritance but provides stable contracts between the components. Our mixin mechanism is realized with aspects according to the Aspect-Oriented Programming (AOP) principles.

## 1 Introduction

The idea to build software from components has been proposed by D. McIlroy on the NATO conference at Garmisch-Partenkirchen in 1968 [8]. Software should be manufactured like machines, i.e. it should be built from separate components which are provided by third party vendors. These components may be also reused in a large number of systems [23].

In this paper we introduce an abstract component architecture model (Component Framework according to [23]) which supports component generators. Based on this model custom specific systems may be generated from standard components. Our software generator refers to the GenVoca architecture [17] and the collaboration-based design [24]. The abstract component framework has been developed on base of the experiences gained from different software systems [12].

In the following we introduce the basic concepts. Firstly, the abstract component framework and the general approach of the collaboration-based design according to GenVoca. Secondly, based on this, we demonstrate our approach of a software generator applying Aspect-Oriented Programming (AOP).

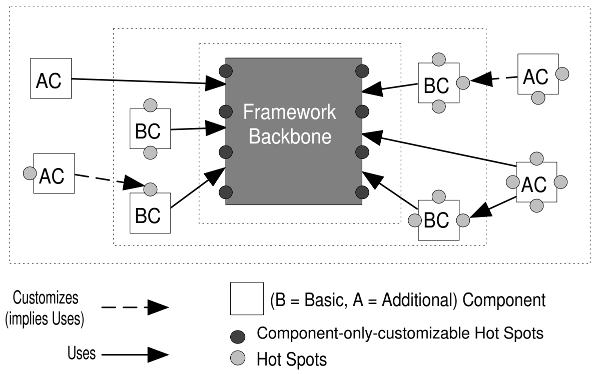


**Fig. 1.** Architectural Model of the Component Framework

## 2 Abstract Component Architecture

The ideas towards an abstract component architecture arose when comparing three different software systems (a generic simulation and visualization system,a tool for automated design of VHDL systems and an industrial robot control [10], [19]). We identified many commonalities which enabled to elaborate a basic framework for the arrangement of components within systems.

In a component-based system different types of components appear (cf. figure 1). Following [12] the *Backbone Components* together with the *Basic Components* provide the central functionality of the system. *Additional Components* refine the system. The components are arranged in shells (or layers). The innermost components provide the central services of the system while the peripheral components contain the refinements.

The *Backbone Components* manage the behavior of the system and the interactions between the components. (In other frameworks we have observed a similar mechanism: one subsystem or component is responsible for the framework's interactions and behavior [6], [7], [12].) Together with the *Basic Components* they realize the basic functionality of the system. The *Additional Components* supplement the first two types of components.

We have detected two kinds of relationships among the components. First they are just plugged-in which means that one component offers services which are requested by another. (Certainly both components may use services from each other.) The arrow indicates that a more outward component is connected to a more basic one. The second type of relationship is well-known from the framework approach. One component customizes the other (cf. customization in [13], [16]). Of course customization implies the usage of services provided by the other components.

All these component types (or components) may be developed by different persons playing different roles in the development process. This means that they can be implemented by different vendors [12]. It is only necessary that the interfaces fit together. Since the assembly of different hardware components is successfully practiced for a very long period the composition of software components should be possible too.

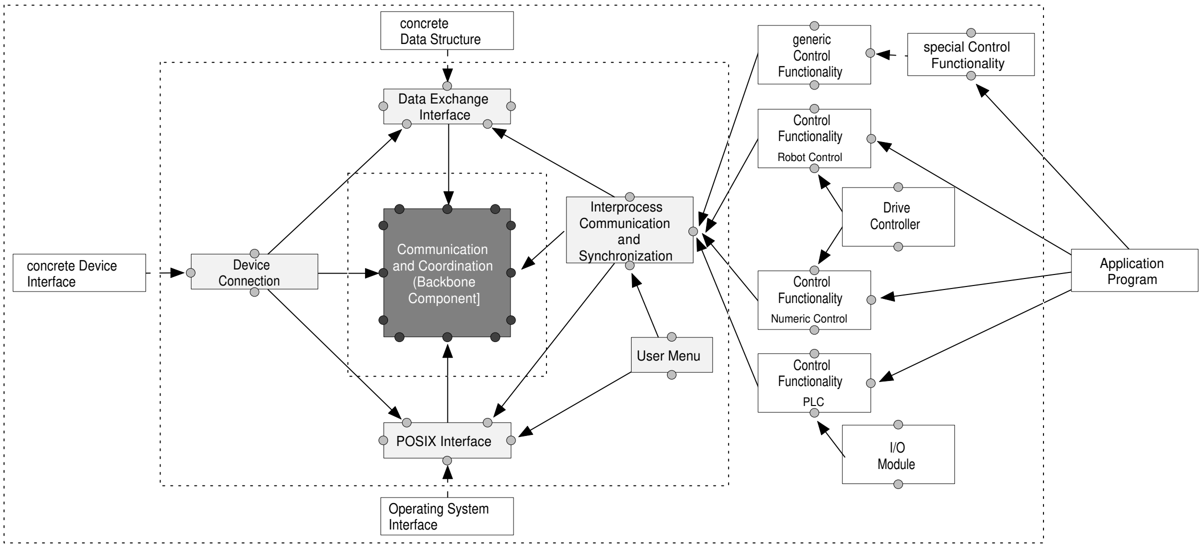


**Fig. 2.** Concrete extended Component Diagram [20]: an Instance of the Architectural Model in the Domain of Industrial Control Systems

## 3 Collaboration-based Software Generators

A collaboration-based design decomposes an application into a set of collaborations and a set of objects. A collaboration specifies a set of allowed behaviors 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 3 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. Three collaborations and three objects are shown in figure 3. Collaboration 2 contains three different roles: Role 2.1, Role 2.2 and Role 2.3 which are encapsulated by Object 1, Object 2 and Object 3 respectively. Collaboration-based design is a form of layered design as layered compositions of collaborations constitute an application.

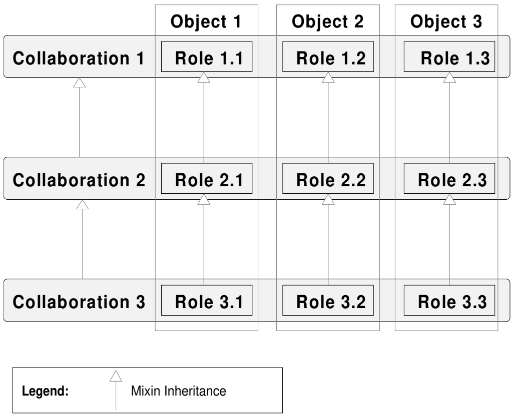


**Fig. 3.** Implementing collaboration-based Designs using Mixin Layers

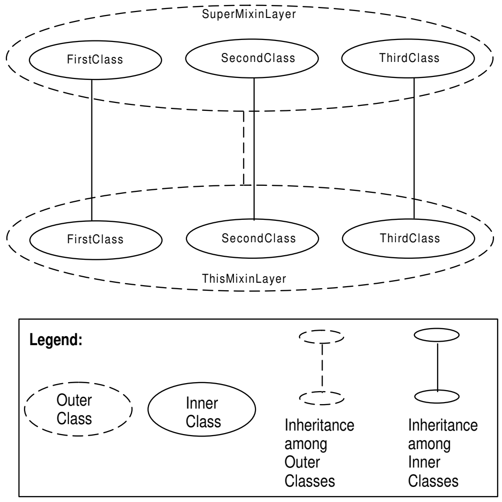


**Fig. 4.** Mixin Layers [17]

A good collaboration-based design is the one where collaborations are largely independent of each other. An effective, scalable way to directly map collaboration-based designs onto implementation are mixin layers. According to [17] 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" [17] as shown in figure 4. 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" [17]. It should be noted that mixin layers can be nested; inner mixins being mixin layers themselves.

As shown in figure 4 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.

Mixin layers are an effective means for stepwise refinement as the layers are reusable and exchangeable. Mixin layers have been successfully employed to build systems in GenVoca [2], [3], [4], [18]. The existing implementations for mixin layers [3], [17] 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 [3], [17] are by far the most effective means of realizing a collaboration-based design. However, they have a number of disadvantages:

- The proposed implementations [3], [17] are based on parameterized inheritance. This results in reliance on templates in C++ and a requirement to use a non-standard template extension for Java and other languages.
- There are considerable disadvantages to employ this approach in a distributed environment [1], [14]. The problem is that the refinement mechanism is based on inheritance which is not suitable for distributed systems [26] (CORBA, for example, does not define implementation inheritance [5]). 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* [23]. As outlined in [22] use of inheritance can lead to unimplemented, inconsistent methods and conflicts in method interfaces.

In this paper we propose an alternative approach to implement collaboration-based design and therefore achieve a functionality similar to mixin layers. We employ Aspect-Oriented Programming (AOP) to overcome the difficulties of existing approaches. Moreover we demonstrate how the Architectural Model may be applied as highest abstraction level for the generator.

## 4 Collaboration-based Generator using AOP

### 4.1 Component Combination via Aspects

Our 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 these constraints are met implicitly using inheritance [15]. Consider for example Role 1.1 and Role 2.1 in figure 3. 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. 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 5. The refinement aspects are shown in the figure as compositions (UML notation) encircled by brackets (keeping with the notation from [11]).

The explicit contracts conform to the properties of inheritance relationships at the same time avoiding the disadvantages of inheritance [23] (e.g. usability in a distributed environment [26]). Furthermore, the approach is more flexible since the constraints can be changed (relaxed or strengthened) if necessary.

All refinement aspects follow a common implementation pattern. In the following we introduce the algorithm we have employed. The algorithm is as follows [15]:

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 5, introduce a reference to Role 1.1 in Role 2.1.

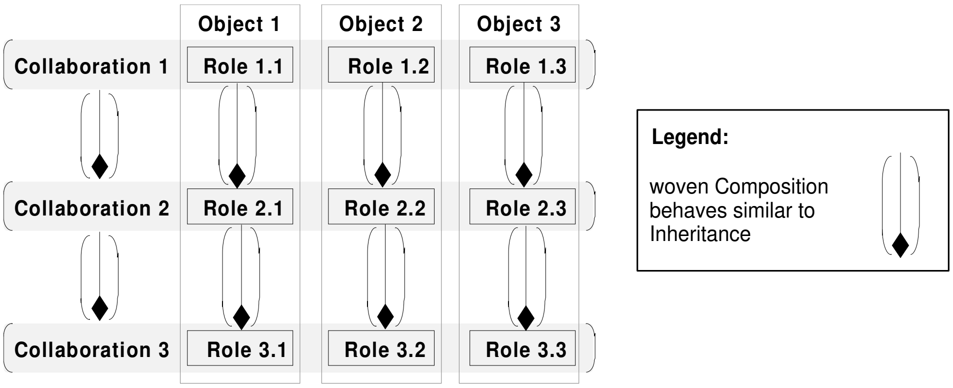


**Fig. 5.** Composition-based Refinement using Aspects

2. Introduce into the super-role public accessor methods for all protected members, i.e. introduce into Role 1.1 (figure 5) 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).
3. Augment the constructor of the sub-role with code to instantiate the super-role. For example (figure 5) 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 5) advise all those accessor methods within Role 2.1 which intend to access or refine members of Role 1.1.

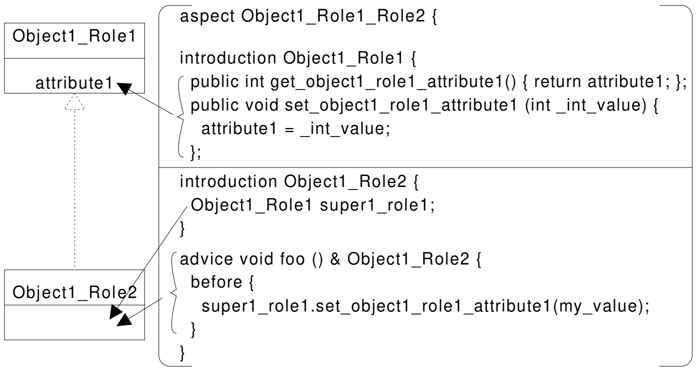


**Fig. 6.** Components Composition realized in an Aspect (notation is referred to AspectJ0.4beta7 (XeroxParc) [27] syntax)

Explicit contracts (cf. figure 6) 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.

### 4.2 Collaboration-based Hierarchy

According to the GenVoca approach a system consists of different layers (collaborations). These layers may be connected by mixin relationships. The highest layer of this system is the most abstract one while the lowest layer is the most concrete. We propose to take the abstract component framework introduced in section 2 as highest level of the generator.

The components of the different layers are combined as demonstrated in section 4.1. The rules for the component combinations may be defined in a script file (e.g. using GenVoca notation). According to such a script the generator can build the various layers from pre-manufactured components and generate the relationships between these components. The generation is done automatically by aspects that are woven into the components (cf. section 4.1).

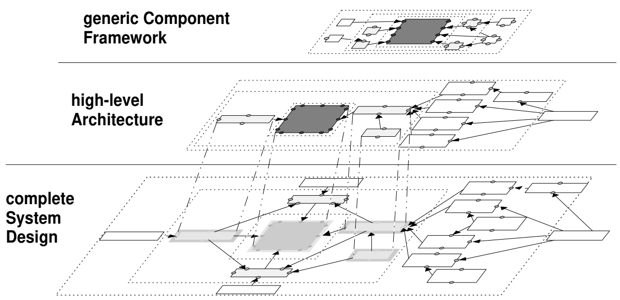


**Fig. 7.** Hierarchical generative Architecture

Figure 7 depicts a hierarchical system generated by an aspect-based generator. The components of the top layer are refined or reused in the lower layers (the reused components are colored light-gray).

## 5 Conclusion

Due to the high degree of component reuse the generation of software simplifies and accelerates the system development process considerably. Now the question is how to structure the components in order to not only plug them into each other but also make them play together.

In this paper we introduce an abstract high level architectural model for a component framework. This component framework distinguishes between three types of components: *Framework Backbone*, *Basic Components* and *Additional Components* [12]. These component types represent a specific behavior and responsibility within a system. *Basic Components* and *Additional Components* are arranged in two layers around the *Framework Backbone* component. Additionally we defined two different kinds of relations (*uses* and *customizes*) among the components.

This architectural model serves as base of a GenVoca type generator. However we improve the generation of components or systems, respectively, by replacing the mixin inheritance through an aspect-oriented approach. We avoid problems like the fragile base class problem and provide aggregations with reliable interfaces as contracts between the components. Thus, such a generator may also be applied for the development of distributed CORBA-based systems where implementation inheritance is prohibited. The aggregation relationships are realized as aspects.

A concept to verify the component combination during the generation process may be found in [21].

## References


1. Batory, D., Johnson, C., MacDonald, B., VonHeeder, D.: FSATS: an Extensible Fire Support Simulator. To appear in Proceedings of ICSE 2000. (2000)
2. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: Proceedings of 5th Int. Conf. on Software Reuse'98. (1998)
3. Batory, D., Smaragdakis, Y.: Building Product-Lines with Mixin Layers. In: Proceedings of ECOOP'99 Workshop on Product-Line Architectures Donohoe, P. editor, Kluwer Academic Publishers (1999)
4. Batory, D., Smaragdakis, Y., Coglianese, L.: Architectural Styles as Adaptors. In: Proceedings of ECOOP'99 Workshop on Product-Line Architectures Donohoe, P. editor, Kluwer Academic Publishers (1999)
5. The Common Object Request Broker: Architecture and Specification.Object Management Group, February (1998)
6. Doscher, D., Hodges, R.: SEMATECS'S Experiences with the CIM Framework. In: Communications of the ACM. vol. 40, no. 10, October (1997) 82 – 84
7. Fayad, M. E., Schmidt, D. C.: Object-Oriented Application Frameworks. In: Communications of the ACM. vol. 40, no. 10, October (1997) 32 – 38
8. McIlroy, M. D.:Mass-produced software components. In Software Engineering Concepts and Techniques, Proceedings of 1968 North Atlantic Treaty Organisation (NATO) Conference on Software Engineering Garmisch-Partenkirchen, Buxton, J.M. and Maur, P. and Randell, B. editors, New York, (1976), 88 – 98
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of ECOOP'97, 11th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science. Springer, June (1997) 220 – 242
10. Küchlin, W., Gruhler, G., Speck, A., Lumpp, T.: HIGHROBOT: A High-performance Universal Robot Control on Parallel Workstations. In: Proceedings

of ECBS'97, Int'l IEEE Symposium and Workshop on Engineering of Computer Based Systems. IEEE Computer Society Press, Monterey, (1997), 444 – 451
11. Lopes, C., Kiczales, G.: Tutorial at STJA: Aspect Oriented Programming with AspectJ. Smaltalk and Java in Education and Industry, Erfurt October (1998)
12. Parsons, D., Rashid, A., Speck, A., Telea, A.: A "Framework" for Object Oriented Frameworks Design. In: Proceedings of TOOLS EUROPE'99. ACM Press, Nancy, France, June (1999), 141 – 151
13. Pree,W.: Komponenten basierte Softwareentwicklung mit Frameworks. dpunkt, Heidelberg (1997)
14. Pulvermüller, E., Klaeren, H., Speck, A.: Aspects in Distributed Environments. In: Proceedings of the GCSE'99, First International Symposium on Generative and Component-Based Software Engineering, Erfurt, September (1999)
15. Pulvermüller, E., Speck, A., Rashid, A.: Implementing collaboration-based Design using Aspect-Oriented Programming. In: Proceedings of TOOLS USA 2000, St. Barbara, Juli/August (2000)
16. Schmid, H.A.: Systematic Framework Design by Generalization. In: Communications of the ACM. vol. 40, no. 10, October (1997) 48 – 51
17. Smaragdakis, Y., Batory, D.: Implementing Layered Designs with Mixin Layers. In: Proceedings of ECOOP'98, 12th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science LNCS 1445. Springer, June (1998) 550 – 570
18. Smaragdakis, Y., Batory, D.: Implementing Reusable Object-Oriented Components. In: Proceedings of 5th Int. Conf. on Software Reuse'98. (1998)
19. Speck, A., Gruhler, G., Küchlin, W.: Object-Oriented Robot Control Framework. In: Proceedings of IECON'98, The 24th Annual Conference of the IEEE Industrial Electronics Society. Aachen, IEEE Press (1998) 1663 – 1666
20. Speck, A.: Component-based Control System. In:Proceedings of ECBS'2000, Int'l IEEE Symposium and Workshop on Engineering of Computer Based Systems. Edinburgh, IEEE Computer Society Press (2000) 176 – 184
21. Speck, A., Pulvermüller, E., Mezini, M.: Reusability of Concerns. In: Proceedings of ECOOP Workshop Aspects and Dimensions of Concerns 2000 Sophia Antipolis, France, 11 – 12 June (2000)
22. Steyaert, P., Lucas, C., Mens, K., D'Hondt, T.: Reuse Contracts: Managing the Evolution of Reusable Assets. In: Proceedings of OOPSLA'96, (1996) 268 – 285
23. Szyperski, C.: Component Software. Addison-Wesley, ACM-Press, New York (1997)
24. VanHilst, M., Notkin, D.: Using Role Components to Implement Collaboration-Based Designs. In: Proceedings of OOPSLA'96, (1996) 359 – 369
25. VanHilst, M., Notkin, D.: Using C++ Templates to Implement Role-Based Designs. In: Proceedings of JSSST Int. Symposium on Object Technologies for Advanced Software. Springer, (1996) 22 – 37
26. Wegner, P.: The Object-Oriented Classification Paradigm. In: Research Directions in Object-Oriented Programming. Wegner, P., and Shriver, B. editors MIT Press, (1987) 479 – 560
27. XEROX Palo Alto Research Center: Homepage of AspectJ. `http://www.parc.xerox.com/aop/aspectj`, (1999)