Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 114 (2005) 153–174
www.elsevier.com/locate/entcs

There are no Aspects

Davy Suvée[1] , Wim Vanderperren[2] , Dennis Wagelaar and Viviane Jonckers

System and Software Engineering Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium

{*dsuvee, wvdperre, dwagelaa, vejoncke*}*@vub.ac.be, http://ssel.vub.ac.be*

Abstract

In this paper, we claim that a specialized aspect module is not required. Instead, we propose an expressive aspect-oriented composition mechanism which can be applied upon existing modules. At the design level, the CoCompose modeling framework is introduced which is based on Model Driven Development. CoCompose allows step-wise refinement from a high-level design to the lowest level design or code level. Using these refinements, CoCompose postpones the decision concerning the modularization construct that is chosen for a particular concern. At the lowest level design however, a specialized aspect modularization construct still needs to be chosen because current aspect-oriented technologies typically introduce an aspect module. For resolving this issue, the FuseJ programming language is proposed that allows implementing all possible concerns as regular components. FuseJ introduces an expressive component composition mechanism that supports both regular and aspect-oriented compositions between components. As such, a seamless transition from design to implementation is achieved because no specialized aspect modules exist both at the design and implementation level.

Keywords: Component-Based Software Development, Aspect-Oriented Software Development, Composition Mechanisms, Model-Driven Development.

[1] The author is supported by a doctoral scholarship from the Flemish Institute for the Improvement of the Scientifical-Technological Research in the Industry (IWT or in Flemish: "Vlaams instituut voor de bevordering van het wetenschappelijk-technologisch onderzoek in de industrie")

[2] The author is supported by a doctoral scholarship from the Fund for Scientific Research (FWO or in Flemish: "Fonds voor Wetenschappelijk Onderzoek")

1 Introduction

Aspect-Oriented Software Development (AOSD) is a new development paradigm that aims at achieving a better separation of concerns than possible using standard object-oriented (OO) software engineering methodologies [15]. AOSD claims that some concerns of an application cannot be cleanly modularized with standard OO technologies as they are scattered over or tangled with the different modules of the system. Such a concern is called *crosscutting* because the concern virtually crosscuts the decomposition of the system. As a result, similar logic is repeated in different modules, with code duplication as a consequence. Due to this code duplication, it becomes very hard to add, edit or remove a crosscutting concern from the system. Typical examples of crosscutting concerns are debugging concerns such as logging [15] and contract verification [30], security concerns [31] such as confidentiality and access control and business rules [6] that describe business-specific logic.

Component-Based Software Engineering (CBSE) is another software engineering paradigm that aims at increasing reusability of individual components and component compositions. CBSE advocates very low coupling between components and high cohesion of single components. Furthermore, components are black-box[3] entities, which are independently deployable [29] . In fact, when CBSE is employed, a component should never explicitly rely onto other specific components in order to increase reusability. As a consequence, CBSE suffers greatly from crosscutting concerns and tangled code because a lot of concerns are spread over and repeated among different components in order to keep the coupling as low as possible. As a result, aspect-oriented ideas are very welcome in the component-based world.

Currently, a wealth of technologies are available that integrate aspect-oriented ideas into component-based software engineering. Examples are JAC [21], JBoss/AOP [13], EAOP [9], OIF [11] and JAsCo [28]. All of these approaches focus at introducing new programming languages or frameworks in order to modularize crosscutting concerns. Support for aspect-oriented ideas during the early cycles of component-based software engineering is still not yet fully explored. Even though several production-quality aspect-oriented technologies exist, it seems to be very difficult to recuperate aspect-oriented ideas in for example the design process. Currently, when designing a software application with aspects in mind, the crucial question is: “when to model concerns as an aspect?”. Indeed, one has to decide which concerns of the application

[3] There is currently no unanimous vision on CBSE. For example, several different approaches exist that motivate why black-box, grey-box or white-box component composition is the better choice. In this paper, we assume the Szyperski vision [29] on CBSE with black-box component composition.

at hand are modeled as aspects and which are modeled as components. In the last few years, work on the so-called "early aspects" [26,22,7] has been emerging. This work focuses on managing crosscutting properties at the early development stages of requirements engineering and architecture design. Although being able to identify possible crosscutting concerns at the very early development stages is an important contribution, the developer is still forced to choose a specific representation, such as an aspect or a component. Akşit et al [2] already motivated that this gives rise to problems when evolving the software, because changing the representation of a concern can have a deep impact on the software architecture. It is for instance possible that a certain concern is initially perfectly modularized by a regular component. However, the requirements of the application might change over time and as such the concern that is modularized as a component turns out to be crosscutting. This concern then has to be refactored into an aspect, which is a cumbersome and error-prone task. This problem is caused by the additional module construct (the aspect) introduced by aspect-oriented technologies in order to modularize a crosscutting concern. Inherently, the behavior of these concerns is not different from the behavior of non-crosscutting concerns; only the composition mechanism differs. When introducing a separate aspect module, the composition mechanism is in fact tangled with the behavior of the concern itself. As a result, other composition mechanisms are inherently ruled out.

The main claim of this paper is that a specialized aspect module should not exist. Instead, we would like to apply aspect-oriented composition mechanisms to existing module constructs. As such, software components do not need to be adapted in order to alter the composition mechanism. To support this claim, we propose both a modeling framework, called CoCompose, and a programming language, called FuseJ. Both approaches allow separating the composition mechanism from the behavior specification. CoCompose employs Model Driven Development (MDD) [16] in order to apply different composition mechanisms on generic design elements. CoCompose allows to refine a model with non-typed, generic concepts to a model that uses components, methods and events. Generic composition concepts can also be refined to both regular and aspect-oriented composition mechanisms. Using these MDD refinements, it is possible to postpone the choice for a specific implementation construct to the lowest level design.

FuseJ is a programming language that recuperates aspect-oriented ideas and allows implementing all concerns as Java Beans. In addition, FuseJ provides a strong composition mechanism that is able to describe both aspect-oriented and component-based compositions. As such, new composition mechanisms are made as unobtrusive as possible, e.g. when representing a software element as a method, it should be possible to compose this method with others as if it were an advice. This allows for postponing the choice for a specific composition mechanism and also enables employing this new composition mechanism on existing software modules.

The next section introduces the CoCompose modeling framework and motivates why an explicit aspect-oriented composition mechanism is better than a separate aspect module at the design level. Afterwards, the FuseJ programming language is introduced that maps seamlessly onto the CoCompose ideas. In section 4, we present the tool support we are currently implementing for supporting the CoCompose/FuseJ approach. Section 5 discusses related work that also avoids introducing an extra modularization construct for aspects. Finally, we state our conclusions.

2 Design Level Composition: CoCompose

CoCompose is a Model Driven Development framework that can be used for the stepwise refinement [33] of software designs. In addition, CoCompose can automatically determine which refinement alternatives to use. Current OO design approaches, such as UML [17], do not explicitly support refinement alternatives. A refined version of the design is made, thereby prematurely eliminating other feasible refinements [2]. UML already starts halfway the refinement process, since it forces the developer to choose one of its specific constructs to represent a design element (e.g. class, operation, package, attribute, ...).

CoCompose uses generic *concepts* to represent design elements [4]. Concepts are explained in detail in subsection 2.1. Each concept can have several refinements, which are explained in detail in subsection 2.2. Concepts can reuse refinements of other concepts in two ways: (1) they can inherit the refinements from another concept or (2) another concept can superimpose its refinements onto this concept. By using refinements for concepts, concepts are employed to model compositions in a declarative way. The concept's refinements specify in more detail how the composition is accomplished. Finally, subsection 2.3 explains how code is generated from CoCompose models.

2.1 Concepts

Each design element is represented as a *concept* in CoCompose. Concepts can participate in *relationships* that specify to which other concepts they are

[4] The first version of CoCompose [32] used several elements for representing a design, whereas currently only "concepts" are employed.

related. Nested concepts are concepts that are contained within another concept. A concept named *AnInheritance*, for instance, contains a *Parent* concept and a *Child* concept that represent the two roles in an inheritance relationship. A *MyClass* concept could have a relationship to the *Parent* concept of the *AnInheritance* concept, which means that it plays the parent role in the inheritance relationship. These relationships are shown in Figure 1.

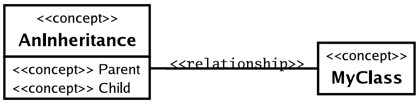


Fig. 1. Representing design elements as concepts

The UML language model can be mapped onto the CoCompose language model, such that design elements described using UML constructs have a CoCompose representation. This mapping is done on the meta-model level: each UML element type (e.g. class, operation, . . .) is mapped onto a CoCompose *concept* representing the element type (e.g. a concept named *Class*). Each actual design element described in UML is mapped onto a CoCompose concept, which inherits from the concept that represents the UML element type (e.g. a class named *MyClass* will map to a *MyClass* concept that inherits from a *Class* concept).

Figure 2 illustrates an CoCompose example model of a hotel booking system expressed in UML. The system consists of three services: a *booking service*, a *discount service* and a *payment service*. The *booking service* is responsible for booking hotels. The *discount service* is a generic service, which assigns discounts depending on business-specific properties. The *payment service* allows to charge a credit card in order to confirm a booking. These services are composed using two abstract collaborations: *BookingDiscount*, which applies a discount to a booking, and *BookingPayment*, which triggers a payment after a booking.

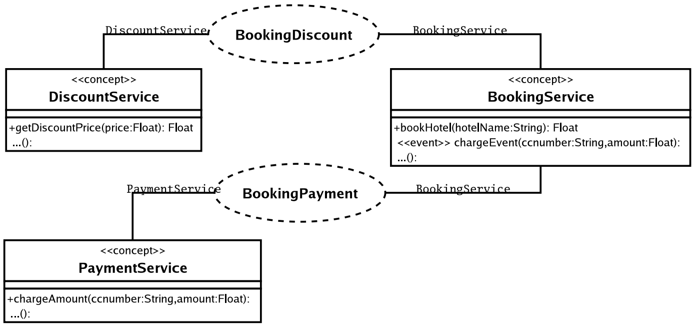


Fig. 2. An example model expressed in UML

Each element maps to a concept, which can inherit the *refinements* (see 2.2) from another concept that represents the element type. Figure 3 illustrates this mapping for the *BookingDiscount* collaboration and the *DiscountService* concept.[5] The *BookingDiscount* collaboration, maps to a concept inheriting from a *Collaboration* concept and contains a *DiscountService* concept and a *BookingService* concept that both inherit from a *CollaborationRole* concept. The *DiscountService* concept is already marked explicitly as an abstract concept, which means that it does not inherit from any type-representing concept.

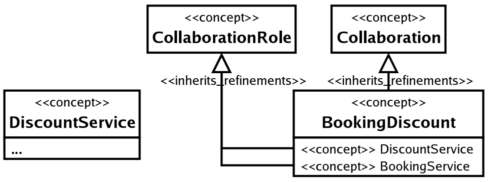


Fig. 3. Mapping the UML example to concepts

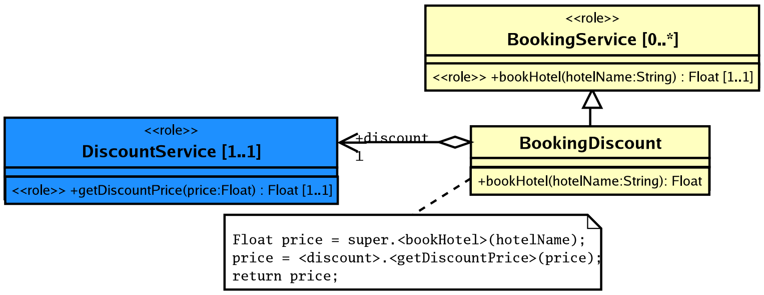
2.2 *Refining Concepts*

Each concept can have several refinements, which are either template designs, called *solution patterns*, programming level templates, called *implementation patterns*, or executable code generators, called *implementation generators*.

The *BookingDiscount* collaboration from Figure 2 has two *solution pattern* refinements: one using standard object-oriented composition and one using AspectJ composition.

Figure 4 shows the standard object-oriented solution pattern for the *Booking-Discount* collaboration, described in UML. It uses several *roles*, which serve as template parameters. Roles are annotated using the “role” stereotype. In this solution pattern two main roles are specified, namely *DiscountService* and *BookingService*. When this solution pattern is applied, the *DiscountService* and *BookingService* roles are replaced by the concepts that are linked to these roles. In Figure 2, the *BookingDiscount* collaboration has two collaboration roles that correspond with these solution pattern roles. The *BookingService* role is in this case filled by the *BookingService* concept from Figure 2, because it is linked to the *BookingService* collaboration role. Note that the *Booking-Service* and the *DiscountService* roles are modeled as UML classes. This means that any concept filling these roles inherits from the *Class* concept (see also Figure 3).

[5] Please note that the inheritance relationships shown in Figure 3 refer to inheritance of *refinements*, not standard UML class inheritance.



<<role>>
BookingService [0..*]
<<role>> +bookHotel(hotelName:String) : Float [1..1]
<<role>>
DiscountService [1..1]
<<role>> +getDiscountPrice(price:Float) : Float [1..1]
+discount
1
BookingDiscount
+bookHotel(hotelName:String): Float

```
Float price = super.<bookHotel>(hotelName);
price = <discount>.<getDiscountPrice>(price);
return price;
```

Fig. 4. The object-oriented solution pattern for BookingDiscount

The solution pattern defines two other roles, *bookHotel* and *getDiscount-Price*, which are nested roles. The concept that is refined does not need to explicitly contain a *bookHotel* and a *getDiscountPrice* concept. Instead, the concepts that fill the containing roles should provide these concepts: e.g. the *BookingService* concept from Figure 2 must contain a *bookHotel* operation [6].

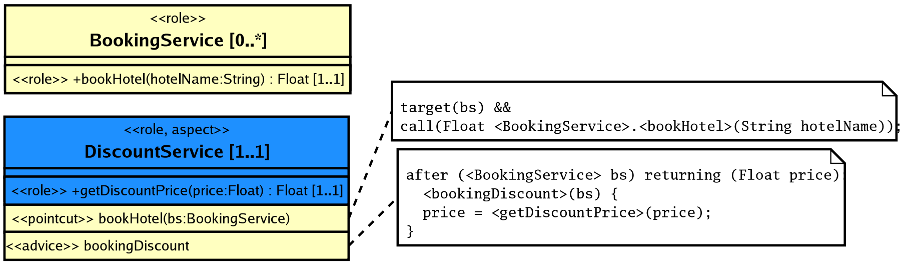
Each role defines a multiplicity constraint, which puts a limit on how many times a role can be filled. In Figure 4, for instance, the *DiscountService* role should be filled exactly once and the *BookingService* role can be filled zero or more times. Coloring is used to mark *role parts*: all concepts having the same color as a role are instantiated once for each time that role is filled. For instance, a *BookingDiscount* concept is created for each concept that fills the *BookingService* role.

The *bookHotel* operation of the *BookingDiscount* class has an *implementation pattern* refinement for Java, which is shown in Figure 4 as a comment. It refers to several other concepts and/or roles in the model, e.g. *bookHotel* in *BookingService* and the *discount* aggregation relation, which is shown by using enclosing `<>`.In order to use these concepts and roles, constraints such as "bookHotel must be a Method" need to be defined for the implementation pattern (in CoCompose, all elements are concepts and it is possible to leave the exact UML element type unspecified).

Assigning a discount to a price is an example of a business rule. In literature, business rules are already identified to be crosscutting concerns [6,20]. Figure 5 shows the AspectJ [14] solution pattern for the *BookingDiscount* collaboration. The UML notation is based upon [24]. In AspectJ, an aspect is used to implement the *DiscountService* concept. In comparison to the object-oriented solution pattern, the AspectJ solution pattern avoids introducing subclasses for every application of the *DiscountService* concept. As such, the *DiscountService* concept remains well-modularized. This AspectJ

[6] The parameter types and return type of the operations should also be marked as roles. In the example, this is ommited in order to preserve clarity.

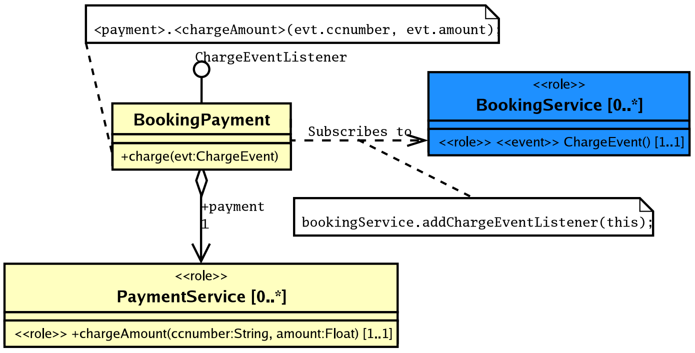
solution however requires to choose a specific aspect module construct. As a consequence, the lowest design and implementation level still suffer from the problems caused by tangling a composition mechanism with a modularization construct. In section 3, the FuseJ programming language is introduced in order to avoid this problem.



<<role>>
BookingService [0..*]
<<role>> +bookHotel(hotelName:String) : Float [1..1]
<<role, aspect>>
DiscountService [1..1]
<<role>> +getDiscountPrice(price:Float) : Float [1..1]
<<pointcut>> bookHotel(bs:BookingService)
<<advice>> bookingDiscount
target(bs) &&
call(Float <BookingService>.<bookHotel>(String hotelName));
after (<BookingService> bs) returning (Float price)
<bookingDiscount>(bs) {
price = <getDiscountPrice>(price);
}

Fig. 5. The AspectJ solution pattern for BookingDiscount

Figure 6 shows the solution pattern for the *BookingPayment* collaboration from Figure 2. It uses a component-oriented *BookingPayment* glue code class to invoke a *chargeAmount* operation whenever a *ChargeEvent* is fired.



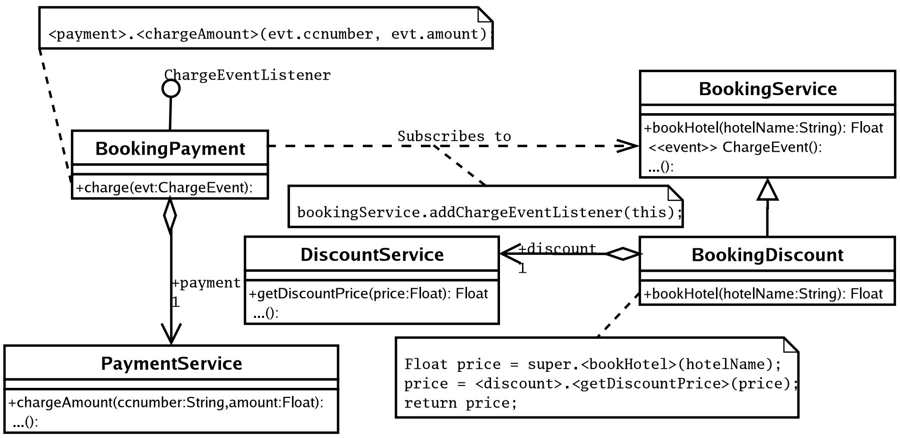
<payment>.<chargeAmount>(evt.ccnumber, evt.amount)
ChargeEventListener
BookingPayment
+charge(evt:ChargeEvent)
Subscribes to
<<role>>
BookingService [0..*]
<<role>> <<event>> ChargeEvent() [1..1]
bookingService.addChargeEventListener(this);
+payment
1
<<role>>
PaymentService [0..*]
<<role>> +chargeAmount(ccnumber:String, amount:Float) [1..1]

Fig. 6. The solution pattern for BookingPayment

Using solution pattern refinements, a design can be refined up to the level of the actual implementation language constructs (e.g. classes and methods, but also AspectJ aspects, pointcuts and advices). Figure 7 shows the object-oriented refined version and Figure 8 shows the AspectJ refined version of the example design in Figure 2.

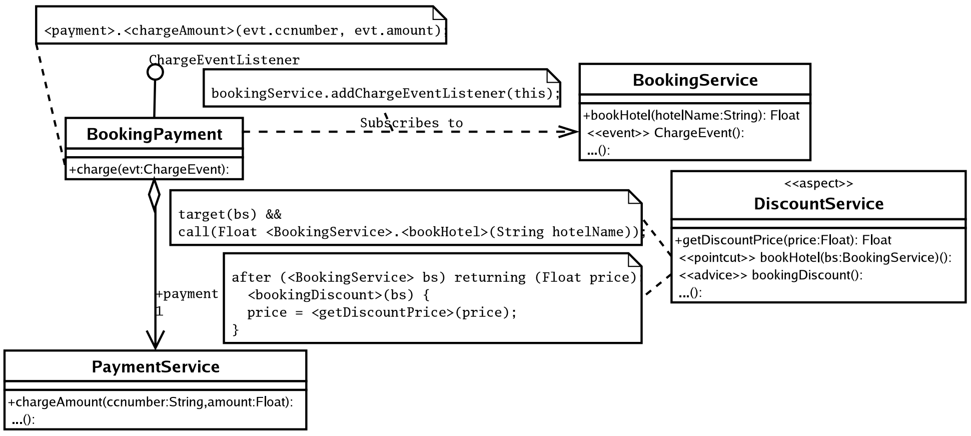
2.3 Code Generation

The code generation process is based upon the original code generation strategy employed in the previous CoCompose approach [32]. In order to generate code from concepts, *implementation generators* are defined for each concept



<payment>.<chargeAmount>(evt.ccnumber, evt.amount)
ChargeEventListener
BookingPayment
+charge(evt:ChargeEvent):
Subscribes to
BookingService
+bookHotel(hotelName:String): Float
<<event>> ChargeEvent():
...():
bookingService.addChargeEventListener(this);
DiscountService
+getDiscountPrice(price:Float): Float
...():
+discount
1
BookingDiscount
+bookHotel(hotelName:String): Float
+payment
1
PaymentService
+chargeAmount(ccnumber:String,amount:Float):
...():
Float price = super.<bookHotel>(hotelName);
price = <discount>.<getDiscountPrice>(price);
return price;

Fig. 7. The example model after applying the standard, object-oriented solution patterns



<payment>.<chargeAmount>(evt.ccnumber, evt.amount)
ChargeEventListener
bookingService.addChargeEventListener(this);
BookingService
+bookHotel(hotelName:String): Float
<<event>> ChargeEvent():
...():
BookingPayment
+charge(evt:ChargeEvent):
Subscribes to
<<aspect>>
DiscountService
+getDiscountPrice(price:Float): Float
<<pointcut>> bookHotel(bs:BookingService)():
<<advice>> bookingDiscount():
...():
target(bs) &&
call(Float <BookingService>.<bookHotel>(String hotelName))
after (<BookingService> bs) returning (Float price)
<bookingDiscount>(bs) {
price = <getDiscountPrice>(price);
}
+payment
1
PaymentService
+chargeAmount(ccnumber:String,amount:Float):
...():

Fig. 8. The example model after applying the AspectJ solution patterns

that represents an implementation language construct. The Java implementation generator for a *Class* concept, for instance, generates Java class skeleton code and pastes in an *implementation pattern*, if any. While the *Class* concept itself does not have any implementation patterns, a concept inheriting from *Class* may have one and can reuse the *Class* implementation generator.

Not only the module constructs for Java (e.g. class, method, ...) have implementation generators, but also the composition constructs (e.g. inheritance). The Java implementation generator for the *Inheritance* concept generates an *extends* clause.

As the implementation generators for Java have common knowledge about the structure of a Java program, they can work together on composing the separate Java elements. When for certain elements no implementation pattern or implementation generator is defined, complete code generation is not possible and a warning is issued.[7]

3 Implementation Level Composition: FuseJ

In the previous section, CoCompose proposes a modeling approach that allows generic compositions between generic concepts which can be refined towards concrete constructs and composition mechanisms. However, the composition mechanism is still tangled with the basic behavior at the lowest design or code level, when traditional aspect-oriented technologies are used as refinement targets. In order to overcome this problem, the FuseJ language is proposed. The idea of FuseJ is to implement all concerns required by a software system as regular components and to introduce an expressive component composition mechanism which allows specifying both regular and aspect-oriented interactions among components. To this end, we propose a new unified component-oriented architecture [27], which makes no distinction between regular and aspect-oriented components at implementation time, at assembly time and at run-time. In the next paragraphs, the first ideas and concepts of this unified component architecture are introduced. To make our ideas more concrete, we present the FuseJ language, which is a practical implementation of the unified component-oriented architecture onto an underlying component model, in this case Java Beans. Afterwards we show how a seamless transition from design to implementation level is achieved, when CoCompose is able to refine its generic compositions towards the FuseJ language.

3.1 Unified Component-Oriented Architecture

In order to introduce an explicit aspect-oriented composition mechanism at the implementation level, a three-layered component-oriented architecture is proposed, featuring a *component layer*, a *gate layer* and a *connector layer*. Components are part of the *component layer* and the composition (both regular and aspect-oriented) between these components is mapped upon a combination of the *gate layer* and the *connector layer*. Figure 9 sketches the architecture of this three-layered model.

Each concern required by the software system is mapped upon a component which is situated in the *component layer*. Their behavior is implemented by means of some base component language and no specific language constructs are provided for specifying possible aspect-oriented interactions. Hence, con-

[7] The generated code is not intended to be edited by the developer: the developer can insert his code in the form of implementation patterns.

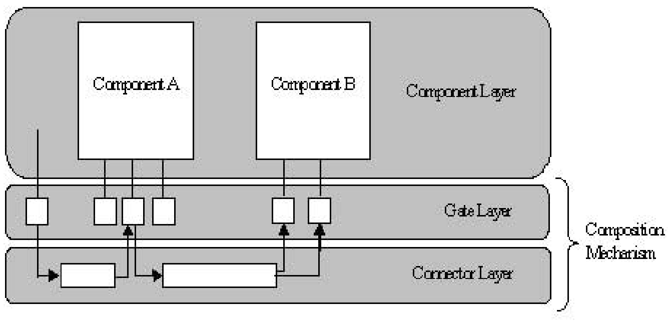


Fig. 9. Unified component architecture

cerns which are typically captured by means of an aspect module, are now described as regular components. As already mentioned in the introduction, we consider each component contained within this component layer to be a black-box entity which is specified and deployed independently from other specific components.

Each component contained within the *component layer* offers a number of services, which we call *features*. These features provided by a component can however not be accessed directly. All communication with or from a component needs to pass through *gates*, which are located in the *gate layer*. Gates are the single entrance and exit points of a component, and provide access to some feature the component implements. A gate can thus be observed as some kind of guardian of the internals of a component. A gate is mapped onto one or more methods implemented within the component. Gates are defined to be two-way channels. Incoming communication has following semantics: "Execute the feature of the component the gate provides access to". Outgoing communication on the other hand, has following semantics: "Whenever the feature of the component the gate provides access to, is executed, do something else". This "something else" depends on the gate(s) of some other component(s) the outgoing communication is connected to. The concept of incoming and outgoing communication allows a gate to be involved in both regular and aspect-oriented compositions at the same time. Logging for instance is a typical example of a concern which is implemented by means of an aspect module. In case of the unified component-oriented architecture however, this logging concern is provided by one or more gates implemented by a regular component. The specification of how this logging concern interacts (in a regular or an aspect-oriented way) with the features provided by other components within the software system is not specified in the gate itself, but deferred until the component composition process.

The interaction between gates is described by making use of *connectors*, which are situated in the *connector layer*. A connector connects the outgoing communication of one (or more) gate(s) with the incoming communication of one (or more) gate(s). Connectors are responsible for describing both regular and aspect-oriented compositions between components, as this description is omitted in both the component and the gate implementation. Connectors which specify regular component compositions are quite similar to the connectors found in most component models. They are used for glueing together the gates of two components by resolving mismatches between method names or argument types. Connectors are however also able to specify aspect-oriented composition between components. In this case, their corresponding gates are glued together in an aspect-oriented way.

Deferring the aspect-oriented interaction specification to the component composition mechanism, in this case the gates and the connectors, has multiple advantages at the implementation level. The reusability of a component is increased, as a component developer does not need to decide at development time whether a component is supposed to interact in a regular or an aspect-oriented way with the other components which are available within the software system. All concerns are implemented as regular components, and it is the connector that is responsible for specifying how the interaction between components takes place. As a result, the features of a component can be reused in both regular and aspect-oriented compositions at the same time. This concept even allows existing components to be reused within an aspect-oriented context.

3.2 Practical Implementation: FuseJ

In this section, the FuseJ language is introduced. The FuseJ language illustrates the idea of having a unified component-oriented architecture at the implementation level by mapping the concepts introduced above onto a real-world component model, in this particular case *Java Beans.* A simple prototype language for gates as well as for connectors is presented. Take in mind that at the moment, this simple prototype language does not support all possible aspect-oriented interactions among components. Indentifying and respresenting this set of required interactions is subject to future research.

The hotel booking case study introduced in the previous section makes use of three components: a *BookingService*, a *PaymentService* and a *Discount-Service* component. The first two components provide regular component features, such as "book a hotel" or "bill a particular amount of money onto a customers credit card". The latter component implements a set of business-rules which assigns a particular discount depending on a set of business-specific conditions. Business-rules are typical examples of concerns which are implemented by means of aspect modules, as business-specific information is often scattered and tangled with the base implementation of the software system. In FuseJ however, all three components are implemented as regular Java Beans, independent of the fact whether they are involved into regular or aspect-oriented compositions.

Each component within the hotel booking application is supplied with a gate-interface which describes the features it provides. Figure 10 shows the implementation of the gate-interfaces of the three components mentioned above.

A gate specification typically consists out of two parts. A *pattern part* which describes the mapping of the gate upon the internal implementation of the component and an *expose part* which exposes the gate properties (input arguments, return value, . . .). The gate-interface of the *BookingService* component for instance provides two gates. The pattern part of the *BookHotel*-gate (line 4 to 6) maps this gate upon the *bookHotel*-method of the *BookingService* component (line 5). The *BookHotel*-gate exposes the *inputHotelName* property (line 8) that represents the name of the hotel that is given as input. The *outputPrice* property (line 9) exposes the return value of the *bookHotel*-method upon which the gate is mapped. Likewise, the *ChargeForHotel*-gate is mapped upon the event which is thrown in order to charge the customer for the hotel booking (line 13 to 20).

For combining several, independent components into a working software system, connectors are employed. These connectors are responsible for connecting one (or more) gate(s) and to describe how these gates should be composed (regular or aspect-oriented). The hotel booking case study requires two connectors in order to fulfill its functionality. The first connector is responsible for implementing the assignment of a discount when somebody books a hotel. The second connector is responsible for billing an amount of money onto a customers credit card when a hotel is booked. This latter connector is a typical example of a regular, component-based composition between components. The *BookingService*-component throws a *ChargeEvent* and the *PaymentService*-component needs to take the right steps when it receives such a *ChargeEvent*. The implementation of this *BookingPayment*-connector is shown in Figure 11.

A connector specifies one (or more) *gate composition(s)*. A gate composition is typically build out of two parts. A *connection*-part (line 3 to 8), which interconnects two gates, and a *mapping*-part (line 9 to 14), which is responsible for specifying the mapping between the concerned gate-properties. In case of the *BookingPayment*-connector, the *ChargeAmount*-gate of the *PaymentService*-component is connected to the *ChargeForHotel*-gate of the *BookingService*-component. The *mapping*-part of the *BookingPayment*-connector is

```
ginterface BookingService {

  gate BookHotel {
    pattern {
      Float BookingService.bookHotel(String hotelname);
    }
    expose {
      String inputHotelName = hotelname;
      Float outputPrice = returnvalue;
    }
  };

  gate ChargeForHotel {
    pattern {
      void BookingService.fireChargeRequest(ChargeEvent event);
    }
    expose {
      ChargeEvent chargeEvent = event;
    }
  };

}
```

```
ginterface DiscountService {

  gate Discount {
    pattern {
      Float DiscountService.getDiscountPrice(Float price);
    }
    expose {
      Float inputPrice = price;
      Float outputPrice = returnvalue;
    }
  };

}
```

```
ginterface PaymentService {

  gate ChargeAmount {
    pattern {
      void PaymentService.chargeAmount(String ccnumber, Float amount);
    }
    expose {
      String inputCCNumber = ccnumber;
      String inputAmount = amount;
     }
  };

}
```

Fig. 10. Gate-interfaces of the hotel booking application

responsible for specifying some gate property translations, in this case the mapping of the *chargeEvent*-property of the *ChargeForHotel*-gate upon the *inputCCNumber* and the *inputAmount* properties of the *ChargeAmount*-gate. The resulting effect of this connector is that the customer is charged by means of the *PaymentService* whenever he/she books a hotel. The implementation of the *BookingDiscount*-connector, which is responsible for implementing the

```
connector BookingPayment {

  connect {
    PaymentService.ChargeAmount;
  }
  to {
    BookingService.ChargeForHotel;
  }
  where {
    PaymentService.ChargeAmount.inputCCNumber =
      BookingService.ChargeForHotel.chargeEvent.visaNumber;
    PaymentService.ChargeAmount.inputAmount =
      BookingService.ChargeForHotel.chargeEvent.amount;
  };

}
```

Fig. 11. BookingPayment connector describing a regular component composition

assignment of a discount to the price of a hotel booking, is shown in Figure 12
.

```
connector BookingDiscount {

  connect {
    DiscountService.Discount;
  }
  after {
    BookingService.BookHotel;
  }
  where {
    DiscountService.Discount.inputPrice =
      BookingService.BookHotel.outputPrice;
  };

}
```

Fig. 12. Discount connector describing an aspect-oriented component interaction

The *BookingDiscount*-connector is responsible for specifying the composition between the *BookHotel*-gate of the *BookingService*-component and the *Discount*-gate of the *DiscountService*-component. The *BookingDiscount*-connector is responsible for specifying an aspect-oriented composition. An aspect-oriented connector is specified in a similar way as a regular component composition connector. The connection-part now specifies an aspect-oriented interaction between the related gates. In this particular case, the *after*-clause specifies that the return value of the *BookHotel*-gate should be replaced with the return value of the *Discount*-gate. Again, a *mapping*-part is provided which maps the *inputPrice*-property of the *Discount*-gate upon the *outputPrice*-property of the *BookHotel*-gate. The resulting effect of this connector is that a discount price is calculated for a booked hotel.

3.3 CoCompose Revisited

In section 2, generic compositions between concepts are introduced which can be refined towards regular or aspect-oriented composition mechanisms. However, the decision of whether a concept should be mapped upon an aspect or a component still needs to be made at the lowest level of design. In FuseJ, all concerns of a software system are described as regular components, and an expressive composition mechanism is employed for combining them. As a result, FuseJ eases the refinement process at the lowest level of design.

Figure 13 shows the FuseJ solution pattern for the *BookingDiscount* collaboration. Apart from the roles, several FuseJ-specific stereotypes such as *connector*, *connect* and *after* are employed.

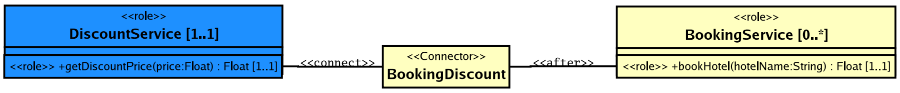


Fig. 13. The FuseJ solution pattern for BookingDiscount

The FuseJ solution pattern shows how the declarative *BookingDiscount* collaboration from Figure 2 can almost directly be translated into a FuseJ *connector*. The object-oriented solution pattern shown in Figure 4 is forced to introduce a new *BookingDiscount* class to implement the composition. The AspectJ solution pattern shown in Figure 5 needs to map the *DiscountService*-concept upon an AspectJ *aspect* in order to enable to aspect-oriented interaction. In the FuseJ solution pattern however, a direct mapping between CoCompose and FuseJ is possible, as concepts are refined to regular components and collaborations between concepts are refined to connectors. As such, FuseJ is a much easier refinement target for CoCompose MDD refinements and allows for better traceability between design and implementation.

Figure 14 shows the FuseJ solution pattern for the *BookingPayment* collaboration. Remember that in FuseJ, connectors can also be used to describe component interactions through events. Figure 15 shows the FuseJ refined version of the example design in Figure 2.

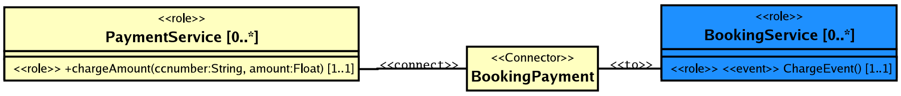


Fig. 14. The FuseJ solution pattern for BookingPayment

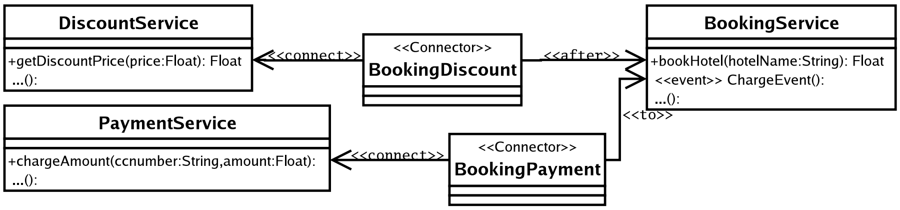


Fig. 15. The example model after applying the FuseJ solution patterns

4 Tool Support

We are currently developing tool support in order to support the approach elucidated in this paper. For the CoCompose modeling framework, a visual drawing editor is developed as a plug-in to the Eclipse IDE Framework [10]. This editor explicitly allows to visualize the CoCompose concepts. Models are stored in CoCompose-specific XMI [18]. We are still working on UML XMI importing and exporting. In the future, we plan to develop several transformation tools for automating the refinement steps and code generation.

For FuseJ, a set of compilers are being developed which enable the compilation of gate and connector specifications. These compilers translate FuseJ specifications into regular Java classes which can be employed within the FuseJ run-time infrastructure. For implementing this run-time infrastructure, we consider the use of *dynamic proxies* [25]. These dynamic proxies facilitate the creation of proxy classes on the fly. Given a set of interfaces, an object that supports each of these interfaces can be created at run-time. When the proxy receives a message, it passes it to an *InvocationHandler*, which is responsible for executing some additional behavior. In our case, an invocation handler is responsible for enabling aspect-oriented compositions between the gates of several components.

5 Related Work

Recently, the Object Management Group has introduced the Model-Driven Architecture (MDA) standard [16]. MDA forms an abstraction layer to specific implementation platforms. It uses model transformations to refine a high-level design (described using a Platform-Independent Model or PIM) to a platform-specific design (described using a Platform-Specific Model or PSM). Several layered PSMs can be defined to gradually refine the design. CoCompose fits in the MDA vision as it also uses several layered refinements, which are described using meta-level solution patterns. These form the intermediate Platform Models (PMs) that define the transformation from a Platform Independent Model (PIM) to a Platform Specific Model (PSM).

In the context of MDA, an approach based on graph transformations has been proposed to transform UML design models to implementation [1]. UMLAUT [12] is a generic UML transformation framework, which can for instance be used for design pattern generation and aspect weaving. The framework is built upon the UML meta-model and allows for defining your own transformations, which can be stored in a transformation library. UMLAUT is not aware of the concept of refinement and requires the developer to manually choose the refinements. UML models can also be used within CoCompose via mapping. This way, the CoCompose mechanism can be applied to different versions of UML by creating only a new mapping.

In *generative programming* [8] and *step-wise refinement* [4], features and feature models are used to create a family of software systems instead of a single system. Features can be optional or mandatory for a software system, depending on the presence of other features. CoCompose can model optional features through solution pattern roles, which can be left unfilled. Alternative features can be modeled by alternative concept refinements.

Also at the implementation level, several aspect-oriented technologies are introduced that do not require a specialized aspect module. Multi-Dimensional Separation Of Concerns (MDSOC), for example, aims at decomposing software so that it encapsulates all relevant kinds (dimensions) of concerns simultaneously, without one dominating the others [19]. The current practical realization of MDSOC is HyperJ for Java [20]. HyperJ captures every concern (crosscutting or not) of an application in a so called *hyperslice*. Similar to FuseJ, HyperJ employs pure Java for describing hyperslices, allowing easier integration of existing modules. *Hypermodules* are used to compose a set of hyperslices in order to form an application or new hypermodule. Technically, HyperJ merges the hyperslices, which are essentially Java classes, using byte-code transformations. In CBSE, the black-box idea is very important, because it decreases the coupling between the different components to the explicit component interfaces only. As such, HyperJ does not comply well with the component-based philosophy. FuseJ, on the other hand, allows Java beans to remain first class entities, even at run-time. As such, replacing or deleting a component dynamically is for example impossible.

Composition Filters [5] is another very different aspect-oriented approach. The Composition Filters model allows expressing crosscutting concerns by attaching filters to existing classes. ConcernJ [23] is one of the practical realizations of the Composition Filters approach that modularizes all concerns of an application into the concern construct. As such, no specific aspect construct is required. The ConcernJ language does however induce a major refactoring of existing code in order to be able to modularize normal classes as concerns. FuseJ is backward compatible as regular Java Beans can be immediately incorporated in the approach. Furthermore, likewise to HyperJ, ConcernJ invasively alters the concerns in order to insert crosscutting behavior. This property also renders the ConcernJ approach less suitable in a component-based context.

Invasive software composition is a component-based approach that unifies several software engineering techniques, such as generic programming, architecture systems and aspect-oriented programming [3]. Invasive Software Composition aims at improving the reusability of software components. To this end, software components are equipped with both explicit and implicit hooks. The hooks are then composed using an explicit and separate composition mechanism. These hooks are in fact very similar to the gate concept of FuseJ. Gates in FuseJ are however only able to depend on the component's public interface, while hooks can be located at any programming construct. As such, hooks are able to describe a finer level of granularity and the resulting composition has more expressive power. Components in invasive software composition are however less loosely coupled in comparison to FuseJ as they are able to depend on each others internals. Technologically, Invasive Software Composition merges the components to one unified application. Undoubtedly, merging components renders a very efficient result. The drawback is that components lose their identity at run-time. FuseJ allows the components, and their corresponding connectors, to remain first-class, even at run-time.

6 Conclusions

In this paper, we claim that a specialized aspect module should not exist. In order to support our claim, both a modeling framework and a programming language are proposed that do not introduce a specialized aspect module. Instead, a powerful composition mechanism is provided that supports aspect-oriented composition. The CoCompose modeling framework is based on MDD and allows step-wise refinement from a high-level design to the lowest level design or code level. Using these refinements, CoCompose allows postponing the decision concerning the modularization construct that is chosen for a particular concern. The drawback of targeting traditional aspect-oriented programming languages in a refinement is that a specific aspect module has to be selected in order to modularize certain concerns at this refined design level. Therefore, the FuseJ programming language is introduced as a better target for implementing CoCompose designs. The FuseJ programming language allows to implement all concerns as regular components and provides an explicit composition mechanism that supports aspect-oriented composition by means of gates and connectors. As such, a separate aspect module does not exist, even at the implementation level and a seamless transition from design to implementation level is achieved.

The proposed FuseJ gate and connector language is only a first prototype of our ideas. At the moment, the expressiveness of the FuseJ connectors does not cover the complete aspect-oriented composition possibilities, as only simple aspect compositions, such as *before* and *after* are supported. Identifying whether and how other aspect-oriented composition mechanism can map in this model is subject to further research. A possible problem with the CoCompose approach is scalability. Because every design element can have multiple refinements, a multitude of possible refinement combinations are available. Experience with the first version of CoCompose that targets Java, ConcernJ and JAsCo already suggests that scalability issues can be overcome. In the future, real-life experiments of CoCompose/FuseJ have to reveal whether this approach is feasible.

References

[1] Agrawal, A., T. Levendovszky, J. Sprinkle, F. Shi and G. Karsai, *Generative programming via graph transformations in the model-driven architecture*, in: *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, USA, 2002.

[2] Akşit, M. and F. Marcelloni, *Deferring elimination of design alternatives in object-oriented methods*, Concurrency and Computation: Practice and Experience **13** (2001), pp. 1247–1279.

[3] Aßmann, U., "Invasive Software Composition," Springer, 2003, 1st edition.

[4] Batory, D., J. N. Sarvela and A. Rauschmayer, *Scaling step-wise refinement*, in: *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, Portland, USA, 2003, pp. 187–197.

[5] Bergmans, L., M. Akşit and B. Tekinerdoğan, *Aspect composition using composition filters*, in: Kluwer, editor, *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2001.

[6] Cibran, M., M. D'Hondt and V. Jonckers, *Aspect-oriented programming for connecting business rules*, in: *Proceedings of the 6th International Conference on Business Information Systems*, Colorado Springs, USA, 2003, pp. 1–6.

[7] Clarke, S. and R. Walker, *Towards a standard design language for aosd*, in: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, 2002, pp. 113–119.

[8] Czarnecki, K. and U. Eisenecker, "Generative Programming: Methods, Tools, and Applications," Addison Wesley, Reading, Massachusetts, USA, 2000.

[9] Douence, R., O. Motelet and M. Südholt, *A formal definition of crosscuts*, in: *Proceedings of the 3rd International Conference on Reflection*, Kyoto, Japan, 2001, pp. 170–186.

[10] Eclipse Consortium, "Eclipse IDE Framework," http://www.eclipse.org/.

[11] Filman, R., *Applying aspect-oriented programming to intelligent systems*, in: *Proceedings of the ECOOP 2000 workshop on Aspects and Dimensions of Concerns*, Cannes, France, 2000.

[12] Ho, W.-M., F. Pennaneac'h and N. Plouzeau, *Umlaut: A framework for weaving uml-based aspect-oriented designs*, Technology of Object-Oriented Languages and Systems (TOOLS Europe) **33** (2000), pp. 324–334.

[13] JBOSS Group, "JBoss/AOP website," http://www.jboss.org.

[14] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of aspectj*, Lecture Notes in Computer Science **2072** (2001), pp. 327–355.

[15] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, *Aspect-oriented programming*, in: *Proceedings European Conference on Object-Oriented Programming*, Atlanta, USA, 1997, pp. 220–242.

[16] Miller, J. and J. Mukerji, "MDA Guide," Object Management Group, Inc (2003), version 1.0.

[17] Object Management Group, Inc, "Unified Modeling Language: Superstructure," (2003), version 2.0.

[18] Object Management Group, Inc, "XML Metadata Interchange (XMI) Specification," (2003), version 2.0.

[19] Ossher, H. and P. Tarr, *Multi-dimensional separation of concerns and the hyperspace approach*, in: Kluwer, editor, *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[20] Ossher, H. and P. Tarr, *Using multidimensional separation of concerns to (re)shape evolving software*, Communications of the ACM **44** (2001), pp. 43–50.

[21] Pawlak, R., L. Seinturier, L. Duchien and G. Florin, *Jac: A flexible solution for aspect-oriented programming in java*, in: *Proceedings of the 3rd International Conference on Reflection*, Kyoto, Japan, 2001, pp. 1–24.

[22] Rashid, A., A. Moreira and J. Araujo, *Modularisation and composition of aspectual requirements*, in: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, USA, 2003, pp. 11–20.

[23] Salinas, P., "Adding Systemic Crosscutting and Super-Imposition to Composition Filters," MSc. Thesis, Vrije Universiteit Brussel, Belgium, 2001.

[24] Stein, D., S. Hanenberg and R. Unland, *An uml-based aspect-oriented design notation for aspectj*, in: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, 2002, pp. 106–112.

[25] Sun Microsystems, Inc., "Dynamic Proxies," http://java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/Proxy.html.

[26] Sutton, S. and I. Rouvellou, *Modeling of software concerns in cosmos*, in: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, 2002, pp. 127–133.

[27] Suvée, D., *Fusej: Achieving a symbiosis between aspects and components*, in: *Proceedings of the 5th GPCE Young Researchers Workshop*, Erfurt, Germany, 2003.

[28] Suvée, D., W. Vanderperren and V. Jonckers, *Jasco: an aspect-oriented approach tailored for component based software development*, in: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, USA, 2003, pp. 21–29.

[29] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," Addison Wesley, Reading, Massachusetts, USA, 1998, 1st edition.

[30] Vanderperren, W., D. Suvée and V. Jonckers, *Combining aosd and cbsd in pacosuite through invasive composition adapters and jasco*, in: *Proceedings of Node 2003 International Conference*, Erfurt, Germany, 2003, pp. 36–50.

[31] Vanhaute, B., B. D. Win and B. D. Decker, *Building frameworks in aspectj*, in: *Proceedings of the ECOOP 2001 workshop on Advanced Separation of Concerns*, Budapest, Hungary, 2001.

[32] Wagelaar, D. and V. Jonckers, *A concept-based approach to software design*, in: *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications*, Marina del Rey, USA, 2003, pp. 307–314.

[33] Wirth, N., *Program development by stepwise refinement*, Communications of the ACM **14** (1971), pp. 221–227.