Adding Features to Component-Based Systems

Maritta Heisel[1] and Jeanine Souquières[2]

[1] Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Verteilte Systeme, 39016 Magdeburg, Germany, email: heisel@cs.uni-magdeburg.de
[2] LORIA—Université Nancy2, B.P. 239 Bâtiment LORIA, 54506 Vandœuvre-les-Nancy, France, email: souquier@loria.fr

Abstract. Features and components are two different structuring mechanisms for software systems. Both are very useful, but lead to different structures for the same system. Usually, features are spread over more than one component. In this paper, we aim at reconciling the two structuring mechanisms. We show how component orientation can support adding new features to a base system. We present a method for adding features in a systematic way to component-based systems that have been specified according to the method proposed by Cheesman and Daniels [5].

1 Introduction

In recent years, software engineering has seen quite a number of new concepts and techniques that promise substantial contributions to a further maturing of the field. In particular, object orientation must be named here. Although object orientation did not result in as major an increase of software re-use as was expected in the beginning, it is now widely used, and it forms the basis of other very promising new approaches in software technology:

Design patterns [8] allow one to represent and re-use previously acquired problem solving knowledge. At this time, the pattern approach is widely accepted, and patterns – that need not be object oriented any more – for almost every phase of the software development process have been developed. Examples are problem frames [13], architectural styles [19], and idioms [3].

Aspect-oriented programming [15] introduces a new programming paradigm on top of object-oriented programming. The idea is to write different programs for different aspects of a software system and then use special compilers to combine the different programs into one. Aspect-oriented programming allows for better mastering the complexity of software systems.

Finally, *component-based software construction* [22, 10] has emerged from object-oriented software development. Its goal is to develop software systems not from scratch but by assembling pre-fabricated parts, as is done in other engineering disciplines. These pre-fabricated parts are called components. [1] They are independently deploy-

[1] The term "component" is used differently in different contexts. Before component orientation came up, an arbitrary piece of software could be called a component. For example, in the context of software architecture, components are those units of software that perform computations (in contrast to connectors that connect components).

able pieces of software. The most important characteristics of software components are the following:

- All services a component provides and all services it requires are accessible only through well-defined *interfaces*.
- Components adhere to *component models*. A component model is designed to allow components to interoperate that are implemented according to the standards set by the model. Building a system from components means selecting components that adhere to a particular component model and composing them in a way that is suitable to achieve the desired system behavior. Examples of component models are *JavaBeans* [20], *Enterprise Java Beans* [21], *Microsoft COM*$^+$ [17], and the *CORBA Component Model* [18].
- Components are deployed in binary form. Access to the source code of the component may not always be possible. Hence, interface descriptions play an important role in component-based development [11].

Common aspects of components and object orientation are the encapsulation of data and functionality in one unit, the role of interfaces, and the concept of instantiation. However, component models have no counterpart in object-oriented software development, which also takes it for granted that the source code of all classes is available. This is, for example, important for inheritance, which is not present among components.

In our opinion, component technology may lead to a substantial progress in our ability to develop highly complex software in high quality and in a cost-effective way.

As we have argued, it is promising to assemble software from well-specified and well-engineered components. However, this is not the only appropriate structuring mechanism for large software systems. For users of such systems, it may be more useful to structure the system according to its functionality. This structuring need not coincide with the component structure. Instead, one may look at the system as offering some basic functionality that can be augmented by *features*. According to Turner et al. [23], a feature is "a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective." Features can be identified in almost every software system, and there are even proposals to base the whole software engineering process on features [23, 6]. Moreover, language constructs for describing features have been developed [9] that support the feature engineering process.

The concepts of features and objects or components, respectively, have been developed independently of each other, and it has turned out that, usually, features are not local to one class or component of a system. Instead, the realization of features involves several classes or components. However, both features and components are adequate and powerful structuring mechanisms of software systems. Hence, it is worthwhile to try and reconcile the two approaches. Related to this goal is Zave's architectural approach to feature engineering [25, 26], called Distributed Feature Composition (DFC). DFC proposes a pipe-and-filter architecture for feature-oriented systems, where features are treated as independent components.

In this article, we demonstrate how the component structure of a system can be exploited to integrate new features into the system in a systematic way. This component structure may be arbritary and need not be an instance of some architectural style

such as pipe-and-filter. Our method relies on the work of Cheesman and Daniels [5], who use different UML notations [2] in a process that starts out from a requirements description and then identifies the necessary components and interfaces, together with their dependencies. The interface operations are specified in terms of invariants and pre- and postconditions. In performing the process, which is described in Section 2, several intermediate documents are constructed, that can be used to find out where the system has to be changed in order to integrate a new feature.

Our method to add new features to component-based systems is then presented in Section 3 and illustrated in Section 4. In Section 5, we point out what conditions must be met in order to reconcile feature orientation and component orientation.

Note that we do not consider the problem of feature interaction in this article. Feature interaction occurs when the integration of a new feature into a system leads to contradictions or unwanted or unexpected system behavior. There are numerous approaches to detect feature interactions (see [12, 4] and the literature cited there), and our feature integration method as described in Section 3 should only be applied in connection with a thorough interaction analysis. [2]

2 A Method for Specifying Component-Based Systems

Cheesman and Daniels [5] propose a process to specify component-based software. This process starts from an informal requirements description and produces an architecture showing the components to be developed or re-used, their interfaces, and their dependencies. For each interface operation, a specification is developed, consisting of a precondition, a postcondition, and possibly an invariant. This approach follows the principle of *design by contract* [16]. Cheesman and Daniels' method ends with component specifications and neither considers the mapping of the developed specifications to a concrete component model nor the implementation of the specified components and interfaces.

During the application of Cheesman and Daniels' method, a number of intermediate documents (expressed in different UML notations) are generated. We will use these intermediate documents to trace the new requirements associated with a new feature in the component architecture of the system. In this respect, our feature integration method (to be described in Section 3) relies on Cheesman and Daniels' component identification and specification method. It only works when the necessary documents are present. Then, we are able to point out in which interface operations of which components changes will be necessary in order to integrate a new feature.

In the following, we summarize Cheesman and Daniels' method. In particular, we point out what documents are developed and how they depend on each other, thus allowing us to navigate among them. In Figures 1–6, we also present parts of the running example used in the book [5], namely a hotel room reservation system[3]. This example

[2] Even though the detection and elimination of feature interactions are important topics, they are not the subject of this paper. The most recent results in this area can be found in the proceedings of the *Feature Interaction Workshop*, which is held every two years.

[3] Details of this example can be found at http:www.umlcomponents.com.

will be taken up again in Section 4, where we will add new features to the hotel room reservation system.

The method consists of two phases, namely *requirements definition* and *specification*.

2.1 Requirements Definition

In the requirements definition phase, a *business concept model* is set up that clarifies the notions of the application domain. It is expressed as a UML class diagram. The business concept model for the hotel reservation system is shown in Figure 1.

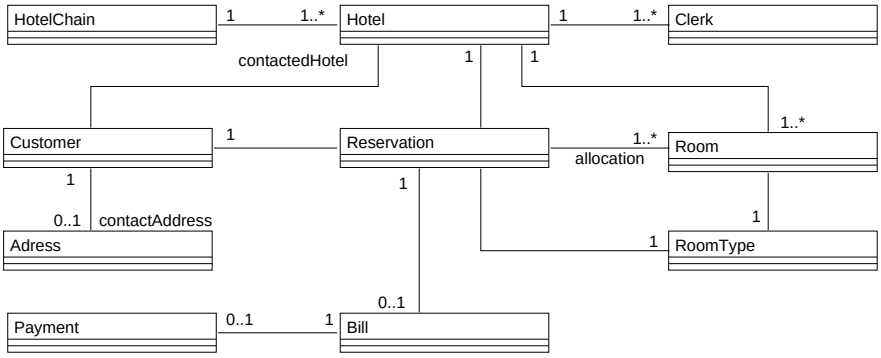

Fig. 1. Business concept model for the reservation system

Then, *business processes* relevant for the system to be constructed are expressed as activity diagrams. “Swim lanes” are used to express the responsibilities for the different steps of the processes. For each step, either an actor or the system to be constructed is responsible. On the basis of the business processes, *use cases* are identified and documented in a use case diagram.

For the hotel reservation system, we have use cases “Make a reservation”, “Update a reservation”, “Take up a reservation”, etc., and actors “ReservationMaker”, “Guest”, “BillingSystem”, etc. Note that the actor “BillingSystem” is not a person but an existing component that will be used by the hotel reservation system.

Each of the use cases is then described using scenarios, as shown in Figure 2. First, the main success scenario is given, which shows the case where everything works as expected. Then extensions are specified which describe alternatives or additions to the main success scenario. For example, if no room of the required type is available for the specified dates, then Step 3 of the main success scenario of Figure 2 is replaced by Step 3 of the **Extensions** section. Hence, this step is an alternative of the step given in the main success scenario. Step 3b), on the other hand, is an addition that is performed in the case that the alternative Step 3 cannot be performed successfully.

The description of the use cases concludes the requirements definition phase.

Name	Make a reservation
Initiator	Reservation Maker
Goal	Reserve room(s) at a hotel

Main success scenario

1. Reservation Maker asks to make a reservation
2. Reservation Maker selects in any order hotel, dates and room type
3. System provides price to Reservation Maker
4. Reservation Maker asks for reservation
5. Reservation Maker provides name and postcode
6. Reservation Maker provides contact email address
7. System makes reservation and allocates tag to reservation
8. System reveals tag to Reservation Maker
9. System creates and sends confirmation by email

Extensions

3. Room not available
   a) System offers alternative dates and room types
   b) Reservation Maker selects from alternatives

3b) Reservation Maker rejects alternatives
   a) Fail

4. Reservation Maker declines offer
   a) Fail

6. Customer already on file (based on name and postcode)
   a) Resume 7

Fig. 2. Scenario of the use case "Make a reservation"

2.2 Specification

This phase comprises the tasks of *component identification*, *component interaction* and *component specification*.

Component Identification For component identification, a *business type model* is developed from the business concept model by adding further detail to the involved classes. For example, the class *Reservation* (see Figure 1) gets the attributes *resRef* : *String* and *dates* : *DateRange*.

In the business type model, *core types* are identified. These are the "essential" types of the application domain. They are the types that can in principle exist without association to other types. For example, a hotel can exist without a reservation, but not vice versa. For the hotel room reservation system, the core types are *Hotel* and *Customer*.

With this information, the components to be developed can be identified. We must develop one component for the main system and one for each core type. The main system will have one interface for each use case identified in the requirements definition phase. Moreover, we must take into account those actors that are not persons but other systems. In the case of the hotel reservation system, this is the billing system. Figure 3 shows the component architecture of the hotel reservation system.

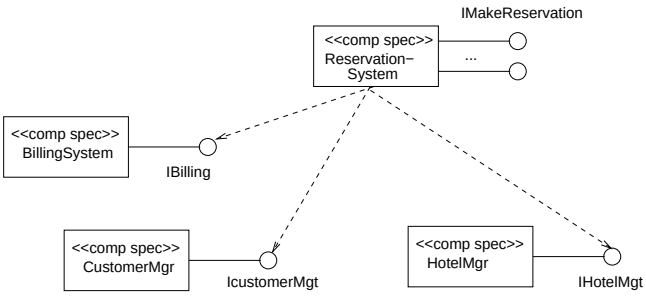

Fig. 3. Component architecture of the hotel reservation system

Next, the operations of each interface must be set up. These operations must allow the system to perform all the steps of the associated use case. For the interface *IMakeReservation*, the operations are derived as follows (see Section 5.2.1 of [5]): the system must allow the person making the reservation to get details of different hotels (Step 2 of the scenario of Figure 2). Moreover, pricing and availability information must be provided for a given room type and a given date range. Finally, it must be possible to actually create a reservation (Step 7). This leads to the following operations for the interface *IMakeReservation*:

$$getHotelDetails(in\ match : String) : HotelDetails[\,]$$

$$getRoomInfo(in\ res : ReservationDetails, out\ availability : Boolean, out\ price : currency)$$

$$makeReservation(in\ res : ReservationDetails, in\ cus : CustomerDetails, out\ resRef : String) : Integer$$

Component Interaction In this phase, collaboration diagrams are developed that show how each operation of an interface of the system to be developed must interact with the other components of the component architecture in order to fulfill its purpose. These collaboration diagrams then yield the necessary interface operations of the other components (those corresponding to the core types).

Figures 4 and 5 show the collaboration diagrams for all three operations of the *IMakeReservation* interface.

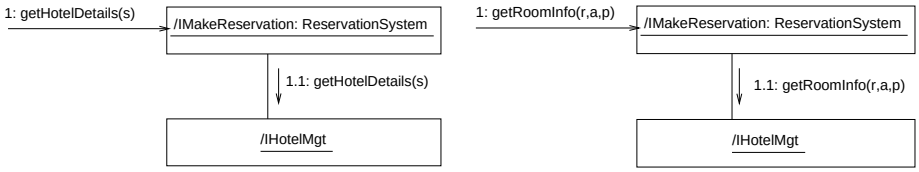

Fig. 4. Collaboration diagrams for the operations *getHotelDetails* and *getRoomDetails*

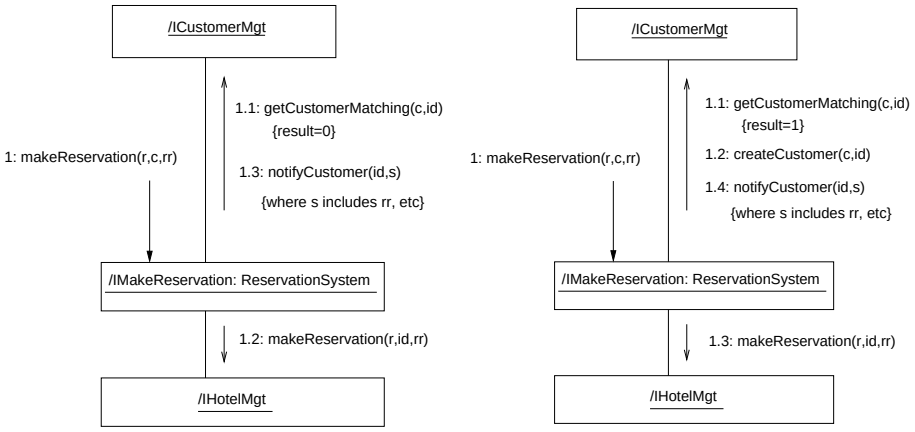

Fig. 5. Collaboration diagrams for the operation *makeReservation*

From these collaboration diagrams, we can conclude that the interface *IHotelMgt* must offer the operations *getHotelDetails* (with different parameters than the operation with the same name of the interface *IMakeReservation*), *getRoomInfo* and *makeReservation*. The interface *ICustomerMgt* must offer the operations *getCustomerMatching*, *notifyCustomer*, and *createCustomer*.

Component Specification At this stage, all necessary interface operations have been identified, together with their parameters and results. In the last stage of the specification phase, the semantics of these operations is specified in terms of pre- and postconditions. The precondition expresses conditions that must be met for the operation to be successfully applied. The postcondition of an operation expresses the effect of the operation under the condition that the precondition holds. Moreover, general business rules may be expressed as invariants.

The specifications are expressed in the object constraint language OCL of UML [24]. As an example, we give the specification of the operation *MakeReservation* of the Interface *IMakeReservation* in Figure 6. This operation will be changed by our feature integration method in Section 4.

3 A Method to Integrate New Features into Component-Based Systems

Our method for feature integration assumes that all the documents described in the previous section are available, in particular:

- a complete class diagram
- a use case model of the main system
- scenarios describing the various use cases

```
makeReservation(in res: ReservationDetails,
    in cus: CustomerDetails, out resRef: String): Integer
pre:
  -- the hotel and room type specified are valid
     hotel.id -> includes(res.hotel) and
     hotel.room.roomType.name -> includes(res.roomType)
post:
  result=0 implies
       -- a reservation was created
       -- note invariant on room Type governing max no
          of reservations
       let h = hotel -> select(x | x.id=res.hotel)
          -> asSequence -> first in
         (h.reservation - h.reservation@pre) -> size=1 and
         let r = (h.reservation - h.reservation@pre)
                  -> asSequence -> first in
           r.resRef = resRef and
           r.dates = res.dateRange and
           r.roomType.name = res.roomType and
           not r.claimed and
           r.customer.name = cus.name and
           cus.postCode -> notEmpty implies
                     cus.postCode = r.customer.postCode and
           cus.email -> notEmpty implies
                     cus.email = r.customer.email
       -- result=1 implies customer not found and unable
                  to create
       -- result=2 implies more than one matching customer
```

Fig. 6. Specification of the operation *MakeReservation* of the Interface *IMakeReservation*

- collaboration diagrams describing the interaction of the main system with other components
- specifications of all interface operations, for the main system as well as for the used components

Based on these documents, we can add a new feature to the main system in a systematic way. As already mentioned in the introduction, we assume that a feature interaction analysis has been performed, and that the requirements that express the properties of the new feature have been adjusted to take into account the result of the interaction analysis.

The goal of our feature integration method is to find out in which places the existing system must be modified in order to accommodate a new feature, and to give guidance how to change the documents listed above. The method starts by considering the more general documents, and then gradually proceeds to take into account the more detailed documents, such as the interface operation specifications. Figure 7 gives an overview of the method and the documents involved.

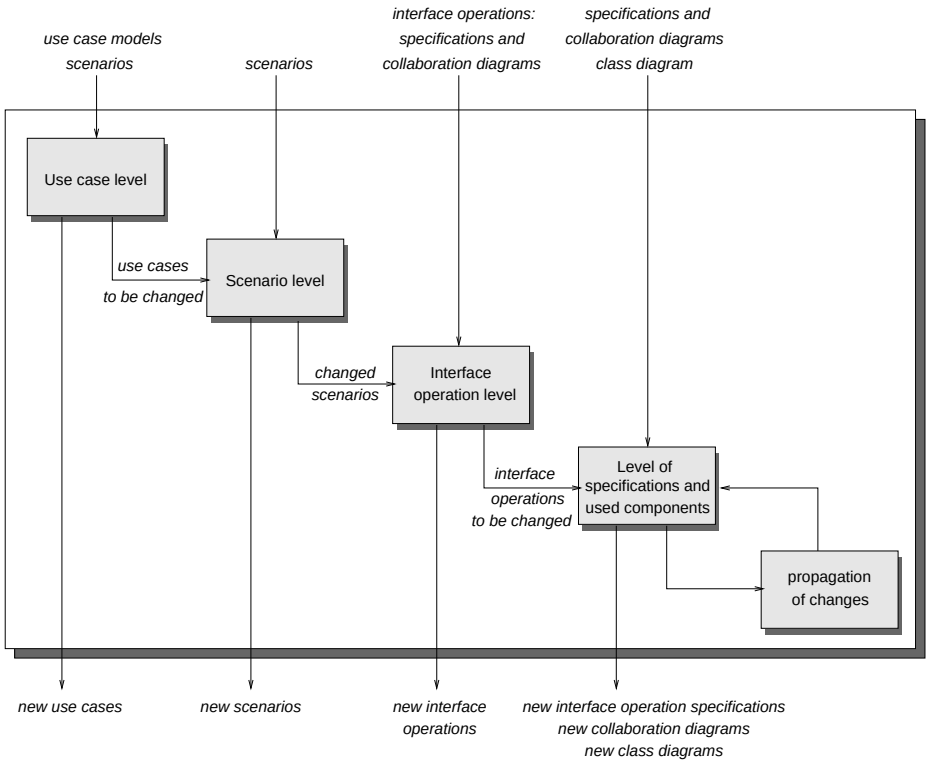

Fig. 7. Overview of the feature integration method

Note that our method ends with a changed set of specification documents. The feature is integrated into the existing component-based system by adjusting the implementation of the components to the changed specifications. As Cheesman and Daniels [5], we neither consider how these specifications are mapped to a concrete component model, nor how the changes in the implementation are actually performed, because these activities are context-dependent to a large extent. To adjust the implementation to the new specification, the source code must be accessible. Hence, our method can be applied when component producer and component consumer belong to the same organization (as can be assumed to be the case for many feature-based systems, for example in telecommunications), or when the component consumer may ask the component producer to perform the required changes. In a situation where someone purchases mass-produced COTS[4] components on the market, our method is not likely to be applicable, because neither the required development documents nor the source code will be available.

Moreover, our method is not primarily intended to be used in an incremental development process such as extreme programming (XP) [1] or feature-driven development

[4] Commercal off the shelf

(FDD)[5] ([6], Chapter 6). We envisage more the situation where an existing system is enhanced by new features after it has been in operation for some time.

In the following, we describe the different steps of the method. For each step, we state which documents are needed and which documents are produced when performing the step, and we give a description of what has to be done. In Section 4, the method is applied to the hotel room reservation system introduced in Section 2.

Note that not always all output items are necessarily generated. When we say for example that the output of one step is a list of something, this list may be empty.

1. **Use case level**

   input: use case model,
   scenarios
   output: list of use cases to be changed,
   list of new use cases

   (a) First, we must decide which of the existing use cases are affected by introducing the new feature. A use case is affected if its scenarios may proceed differently than before. The use case diagram serves as a basis for this decision. If the decision cannot be made on the basis of the use case model alone, one can also have a look at the scenarios.
   (b) Second, we must decide if the introduction of the new feature leads to new use cases. This decision is based on the same documents as Step 1a. If new use cases must be introduced, then the method described in Section 2 should be applied, starting from the new use cases. Here, our method does not introduce anything new, so we will not further discuss this case in the following.

   The following steps have to be performed for each use case that must be changed.

2. **Scenario level**

   input: scenarios describing the use case to be changed
   output: new scenarios, describing the modified behavior resulting from the new feature

   In this step, we set up new scenarios by modification of the old ones. The new scenarios describe the system behavior after integration of the new feature. Modifying the scenarios may mean:
   - changing the order of actions
   - changing parameters of actions
   - introducing new actions

[5] FDD is an incremental development process, much like XP. Instead of user stories, feature sets are used to structure the iterations of the developement process. In FDD, features are defined to be “small ’useful in the eyes of the client’ results” and required to be implementable in two weeks, much like user stories in XP. In telecommunications, however, features may be larger entities of functionality. Moreover, the components mentioned by Coad et al. are not components in the sense of component-based development, see the footnote in the introduction.

– deleting actions

For the resulting scenarios, the following steps have to be performed.

3. **Interface operation level**

input: old scenarios,
new scenarios as developed in Step 2,
specifications and collaboration diagrams for the existing interface operations
output: list of interface operations to be changed,
list of new interface operations,
notes indicating how to change the global control structure of the interface implementation,
list of interface operations to delete

We must now investigate the effect of the changed scenarios on the interface operations of the use case under consideration. Four cases must be distinguished, depending on how the scenarios have been changed.

(a) If the order of actions has been changed, we must take a note that later the program that implements the global control structure of the interface must be changed accordingly. As Cheesman and Daniels' method does not require to explicitely record the order in which the different interface operations are invoked, there is no specification document which we could change to take into account the changed order of actions other than in the new scenarios. It would be suitable to introduce one more specification document that specifies in which order the different interface operations may be invoked, as is done in the Fusion method [7], for example.

(b) If parameters of actions have been changed in Step 2, then this will result in changing existing interface operations. Those interface operations that were introduced in order to take into account the scenario action in question must be changed.

(c) If new actions have been introduced into the scenarios, we must check if these can be realized using the existing interface operations (or modifications thereof), or if new interface operations must be introduced. Any new interface operation must be recorded.

(d) If actions are deleted from the scenarios, it must be checked if the corresponding interface operations are still necessary. However, operations can only be deleted if the new feature replaces the old functionality instead of supplementing it.

The next step of our method concerns those interface operations that must be changed. For the new interface operations, we proceed as described in Section 2. Again, we will not consider the new interface operations any more in the following.

4. **Level of specifications and used components**

   input: list of interface operations to be changed,
   together with their specifications and collaboration diagrams,
   class diagram
   output: new class diagram,
   new collaboration diagrams and specifications of the interface operations to be changed,
   list of interface operations of used components that must be changed,
   list of new interface operations of used components,
   list of interface operations of used components to be deleted

   – For each interface operation that must be changed, we must first decide on its new parameters. To take into account parameter changes, we need the class diagram which gives information on the classes used as parameters or results of the interface operations. All changes that concern classes or attributes of classes must be recorded in the class diagram.
   – Next, we consider the collaboration diagrams. If parameters of the operation have been changed, then this must be recorded in the collaboration diagram. Moreover, it must be decided if the component interactions as described by the collaboration diagram must be changed. The possible changes correspond to the changes as described in Step 2, and the collaboration diagram must be adjusted accordingly. Updating the collaboration diagrams results in lists of interface operations of used components to be changed, newly introduced, or deleted.
   – Finally, the specification of each operation must be updated to take into account its new behavior.

5. **Propagation of changes**

   input: list of interface operations of used components that must be changed,
   together with their specifications and collaboration diagrams,
   list of new interface operations of used components
   output: new collaboration diagrams and specifications for the operations given as input

   Step 4 resulted in a list of new interface operations of used components and a list of interface operations of used components to be changed. The newly introduced operations are specified according to the method of Section 2, and the operations to be changed are treated as described in Step 4. This procedure is repeated until no more changes are necessary or we have reached basic components that do not use other components.

Note that after one feature has been integrated, we end up with the same – but updated – set of documents as we started out. Carrying out our method amounts to maintaining the development documents of the system. Hence, several features can be integrated successively in the same way.

4 Adding Features to a Hotel Reservation System

In Section 2, we have introduced a basic hotel room reservation system that allows its users to make reservations of hotel rooms for a given time frame. One could imagine, however, that the functionality of that system could be enhanced in several ways. These enhancements can be expressed as features.

In this section, we add the following new features to the base system.

Subscription A customer can make several reservations at the same time, and for regular time intervals. For example, it is possible to reserve a room for two days every Monday during the next two months.

AddditionalServices The hotel may offer additional services to be booked together with a room, for example wellness facilities.

For these features, we will carry out our feature integration method described in the previous section. We demonstrate that using our method large parts of the feature integration process can be carried out in a routine way.

4.1 Adding the Subscription Feature

We describe the steps to be carried out one by one and point out which documents are used and produced.

Step 1: Use case level The starting point of our method is the use case model of the system (which we did not show in Section 2) and the scenarios describing the use cases. We have to decide which of the use cases are affected by the new feature. The use case "Make a reservation" must be changed. In the following, we will only consider that use case, although the use case "Update a reservation", for example, would also be affected. The procedure is the same for all affected use cases, however. No new use cases need to be introduced to accommodate the feature.

Step 2: Scenario level We now consider the scenario description given in Figure 2 and determine how it must be changed to allow for subscription reservations.

Here, a general decision must be taken. Of course, even in the presence of the subscription feature, it must still be possible to make "simple" reservations. This can be achieved in two ways: either, we define one more alternative main success scenario that describes subscription reservations. This is possible, because one use case can have more than one associated success scenario. Or, we define the modified behavior as extensions of the main success scenario of Figure 2. For the purposes of this paper, we decide to take the second option.

Hence, the following modified or new actions are introduced in the **Extensions**-section of the scenario description. Steps 1 and 3 are modified, and a new Step 2a is added:

1. Reservation Maker asks to make a *subscription reservation*
2a. Reservation Maker provides subscription details

The text of Step 3 remains the same, but we must make a note that the procedure to calculate the price of the reservation must probably be modified.

Step 3: Interface operation level Based on the modified scenario constructed in the previous step, we must decide how the introduced changes affect the operations of the interface *IMakeReservation*. We cannot delete any operation, but we need a new one called *enterSubscriptionDetails*. This function, which we must specify according to the method of Section 2[6], will yield an object containing all details necessary for a subscription reservation.

Of the existing operations, we decide that *getHotelDetails* and *getRoomInfo* may remain unchanged, whereas *makeReservation* must be changed.

Step 4: Level of specifications and used components We must now update the documents describing the operation *makeReservation*, which has the following profile:

$$\begin{aligned} &\mathit{makeReservation}(\mathit{in\,res} : \mathit{ReservationDetails}, \mathit{in\,cus} : \mathit{CustomerDetails}, \\ &\qquad\qquad \mathit{out\,resRef} : \mathit{String}) : \mathit{Integer} \end{aligned}$$

We see that the change manifests itself in the parameter type *ReservationDetails*, which must be changed. In the class diagram (see Figure 1), the class *Reservation* must be adjusted. This class had got an attribute $\mathit{dates} : \mathit{DateRange}$. This attribute must now contain a nonempty collection of date ranges instead of one date range. We change the attribute declaration accordingly: $\mathit{dates}[1..*] : \mathit{DateRange}$.

Next, the collaboration diagrams of the operation *makeReservation* (see Figure 5) must be considered. The collaboration diagram itself need not be changed. However, the parameter *r* of type *ReservationDetails* is passed on to the operation *makeReservation* of the interface *IHotelMgt*. Hence, this operation must be checked for necessary changes, too.

Finally, we must update the specification of *makeReservation* given in Figure 6. An inspection shows that all the changes are hidden in the *res* of type *ReservationDetails*. This concerns the line `r.dates = res.dateRange` of the specification. Hence, the text of the specification remains the same.

Step 5: Propagation of changes The changes performed in the previous steps must be propagated in the documents concerning the operation *makeReservation* of the interface *IHotelMgt*, as was found out in Step 4. We do not present this propagation in the present paper.

This concludes the integration of the subscription feature into the hotel room reservation system. We did not introduce any new use cases, but we introduced new actions in the scenario of an affected use case, which resulted in a new interface operation. Another interface operation was changed, but the change concerned only one of the types involved in the operation.

[6] As the specification of interface operations is not part of our method, we will not present specifications of newly introduced operations in this paper. We only demonstrate how changes are made in a systematic fashion.

4.2 Adding the AdditionalService feature

We now integrate this second feature, starting from the same set of documents as in the Section 4.1. Note that for reasons of simplicity, we add the feature to the base system and not to the system that results in having added the subscription feature.

Step 1: Use case level As in Section 4.1, we judge that no new use cases are necessary, and that the use case "Make a reservation" is affected by the billing feature.

Step 2: Scenario level We change the scenario of Figure 2 by introducing a new step in the **Extension** section:

2a. Reservation Maker selects additional services

As in Section 4.1, the text of Step 3 remains the same, but we must make a note that the procedure to calculate the price of the reservation must probably be modified.

Step 3: Interface operation level As far as the operations of the interface *IMake-Reservation* are concerned, we can certainly not delete any of them. We decide that information on the additional services provided belongs to the contacted hotel. Hence, we need no new interface operation, but we must change the operations *getHotelDetails* and *makeReservation*. The operation *getRoomInfo* remains unchanged.

Step 4: Level of specifications and used components To take into account that additional services are associated with a hotel and can possibly be associated with a reservation, we must update the class diagram by introducing a class *AdditionalServices* with associations to the classes *Hotel* and *Reservation*, as shown in Figure 8.

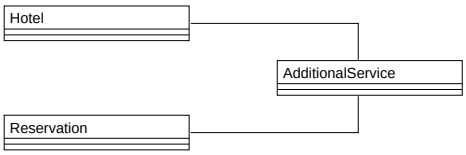

Fig. 8. Part of new class diagram for the hotel reservation system

We now consider the updates necessary for the operation *getHotelDetails*. The parameter of *getHotelDetails* does not change, and also the collaboration diagram remains the same as the one shown in Figure 4. However, we must note that the operation *getHotelDetails* of the interface *IHotelMgt* must be changed. This operation must now also make available information about the additional services provided by the hotel. Since we have not presented the specification for *getHotelDetails* in Section 2, we do not discuss the changes of the specification here.

The changes that are necessary for the operation *makeReservation* resemble the ones described in Section 4.1. The reservation details must be changed to take into account a possible reservation of additional services.

Step 5: Propagation of changes The changes performed in the previous steps must be propagated in the documents concerning the operations *getHotelDetails* and *makeReservation* of the interface *IHotelMgt*, as was found out in 4. We do not present this propagation in the present paper.

When integrating the additional services feature, we neither introduced new use cases nor new interface operations. However, we changed the class diagram substantially, and consequently had to change two of the three existing interface operations. The most substantial changes, however, are the ones to be performed in the *HotelMgr* component.

This shows that it can hardly be avoided that several components are affected by introducing a new feature. However, by tracing the development documents as we have demonstrated in this section, we do not get lost in the component structure, but are guided to the components that must be changed.

5 Conclusions

As discussed in Section 1, both features and components are powerful and adequate structuring mechanisms for software systems. However, they lead to different system structures. This is due to the fact that components structure the system from a developer's point of view, whereas features structure the system from a user's point of view. Of course, both of these views are important and should be used when describing software systems.

This dichotomy leads us to the question how both structuring mechanisms can be used in parallel without incoherences. The present paper proposes an approach how to reconcile the two structuring mechanisms of features and components. On the one hand, the systems we consider are structured according to a component architecture. This structure reflects the implementation units of the system. On the other hand, we consider our systems to consist of a base system to which features can be added. This second structure, however, is not directly reflected in the implementation of the software system.

To bridge the gap between the two system structures, we need to map the feature structure onto the component structure. In particular, this means that we must *trace* each feature in the component structure. This approach is similar to requirements tracing as performed in requirements engineering [14]. To allow for the tracing, a number of intermediate documents are necessary, together with the dependencies between them. For example, scenarios are derived from use cases, interface operations are derived from scenarios, and collaboration diagrams are derived from interface operations. Thus, following the dependency links between the different documents allows us to determine how the realization of a feature is distributed in the component architecture. Integrating a feature is then performed by following those paths in the dependency structure where changes occur. As we have shown in Sections 3 and 4, this traversal of the dependency structure is possible in a systematic way.

Hence, our method does not change the fact that features are distributed over several classes or components, but it helps to deal with this fact in a satisfactory way.

Acknowledgments. We thank Hung Ledang and Thomas Santen for their comments on this paper.

References

1. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unifi ed Modeling Language User Guide*. Addison-Wesley, 1999.
3. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.
4. Muffy Calder and Alice Miller. Detecting feature interactions: how many components do we need? In H.-D. Ehrich, J.-J. CH. Meyer, and M.D. Ryan, editors, *Objects, Agents and Features. Structuring mechanisms for contemporary software*, this volume. Springer-Verlag, 2004.
5. John Cheesman and John Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.
6. Peter Coad, Eric Lefebvre, and Jeff De Luca. *Java Modeling In Color With UML*. Prentice Hall, 1999.
7. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.
9. Stephen Gilmore and Mark Ryan, editors. *Language Constructs for Describing Features*. Springer-Verlag, 2000.
10. George T. Heineman and William T. Councill. *Component-Based Software Engineering*. Addison-Wesley, 2001.
11. Maritta Heisel, Thomas Santen, and Jeanine Souquières. Toward a formal model of software components. In Chris George and Miao Huaikou, editors, *Proc. 4th International Conference on Formal Engineering Methods*, LNCS 2495, pages 57–68. Springer-Verlag, 2002.
12. Maritta Heisel and Jeanine Souquières. A heuristic algorithm to detect feature interactions in requirements. In Stephen Gilmore and Mark Ryan, editors, *Language Constructs for Describing Features*, pages 143–162. Springer-Verlag, 2000.
13. Michael Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.
14. Matthias Jarke. Requirements tracing. *Communications of the ACM*, pages 32–36, December 1998.
15. Gregor Kiczales. Aspect oriented programming. *ACM SIGPLAN Notices*, 32(10):162–162, October 1997.
16. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.
17. Microsoft Corporation. *COM*$^+$, 2002. http://www.microsoft.com/com/tech/COMPlus.asp.
18. The Object Mangagement Group (OMG). *Corba Component Model, v3.0*, 2002. http://omg.org/technology/documents/formal/components.htm.
19. Mary Shaw and David Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
20. Sun Microsystems. *JavaBeans Specifi cation, Version 1.01*, 1997. http://java.sun.com/products/javabeans/docs/spec.html.
21. Sun Microsystems. *Enterprise JavaBeans Specifi cation, Version 2.0*, 2001. http://java.sun.com/products/ejb/docs.html.

22. Clemens Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1999.
23. C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, 1999.
24. Jos Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.
25. Pamela Zave. Architectural solutions to feature-interaction problems in telecommunications. In K. Kimbler and W. Bouma, editors, *Proc. 5th Feature Interaction Workshop*, pages 10–22. IOS Press Amsterdam, 1998.
26. Pamela Zave. Feature-oriented description, formal methods, and DFC. In Stephen Gilmore and Mark Ryan, editors, *Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2000.