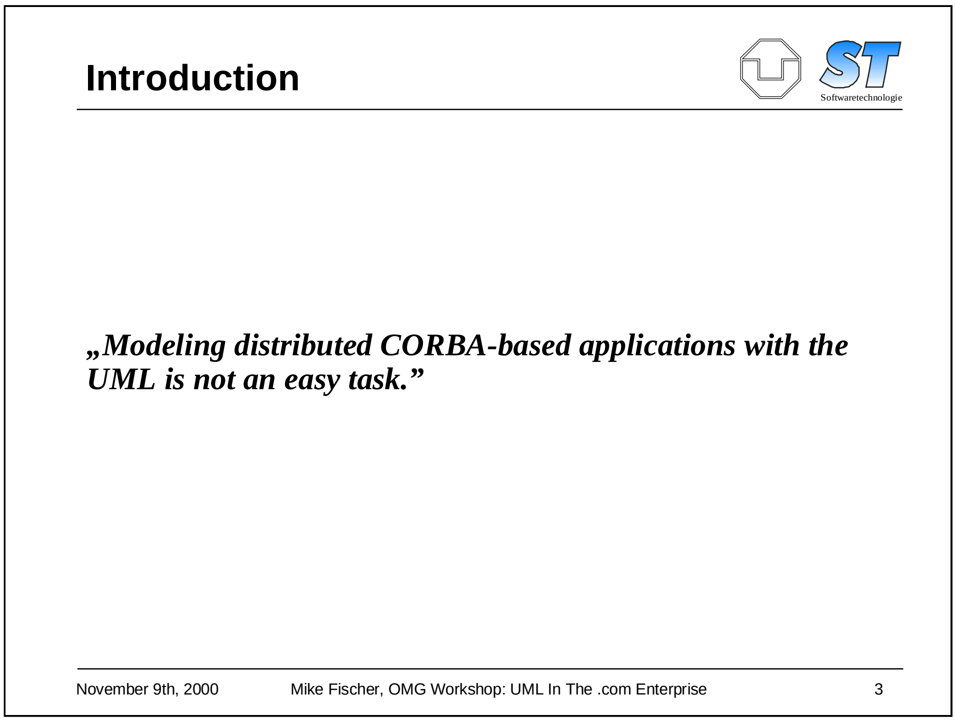
Abstract modeling of CORBA-based applications with UML

Mike Fischer
Mike.Fischer@inf.tu-dresden.de

Dresden University of Technology
Department of Computer Science
Software Engineering Group

Agenda

- Introduction in the problem field
- UML extensions (1) in order to
  - distinguish CORBA interfaces and their implementations
  - describe how an implementation realizes its CORBA interface
- UML extensions (2) in order to
  - model how elements make use of object services based on combination pattern
  - identifying of roles within CORBAservices (exemplary)
- Summary and Perspectives



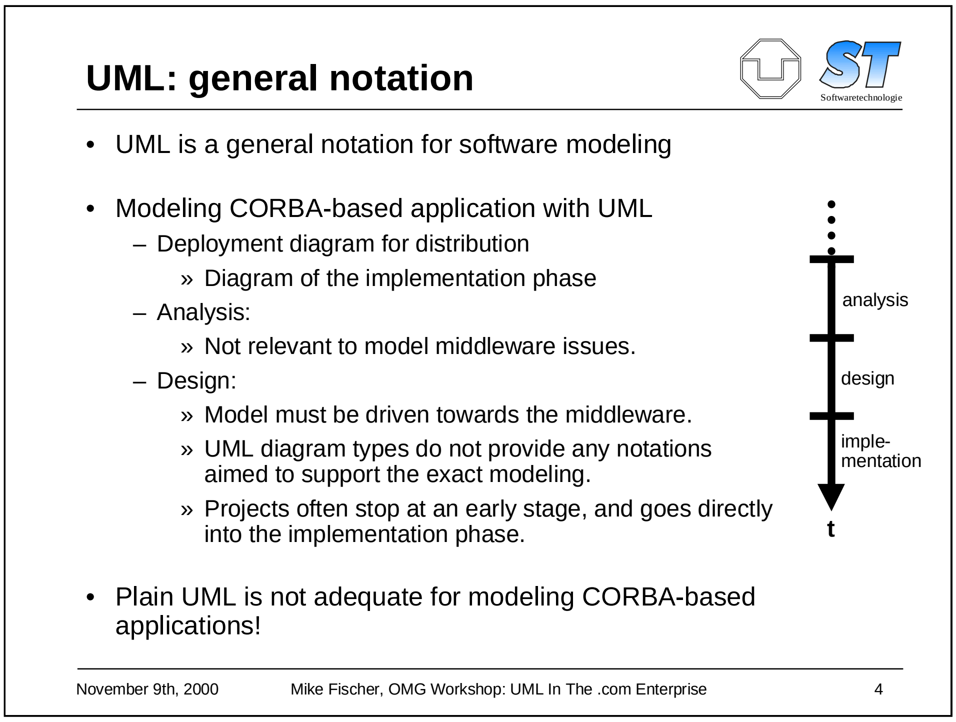
Introduction

Softwaretechnologie

„Modeling distributed CORBA-based applications with the UML is not an easy task.”

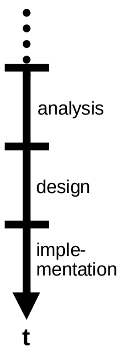
Modeling distributed CORBA-based applications with the UML™ is not an easy task. .....

What is a CORBA-based application? Shortly, it is a complex application comprising several distributed parts. This parts communicate together by using a Common Object Request Broker Architecture ([OMG98a]) conform implementation. CORBA allows a developer of such an application to achieve his tasks without considering: distribution, different languages, network protocol, and so on.



UML: general notation

- UML is a general notation for software modeling
- Modeling CORBA-based application with UML
  - Deployment diagram for distribution
    - » Diagram of the implementation phase
  - Analysis:
    - » Not relevant to model middleware issues.
  - Design:
    - » Model must be driven towards the middleware.
    - » UML diagram types do not provide any notations aimed to support the exact modeling.
    - » Projects often stop at an early stage, and goes directly into the implementation phase.
- Plain UML is not adequate for modeling CORBA-based applications!



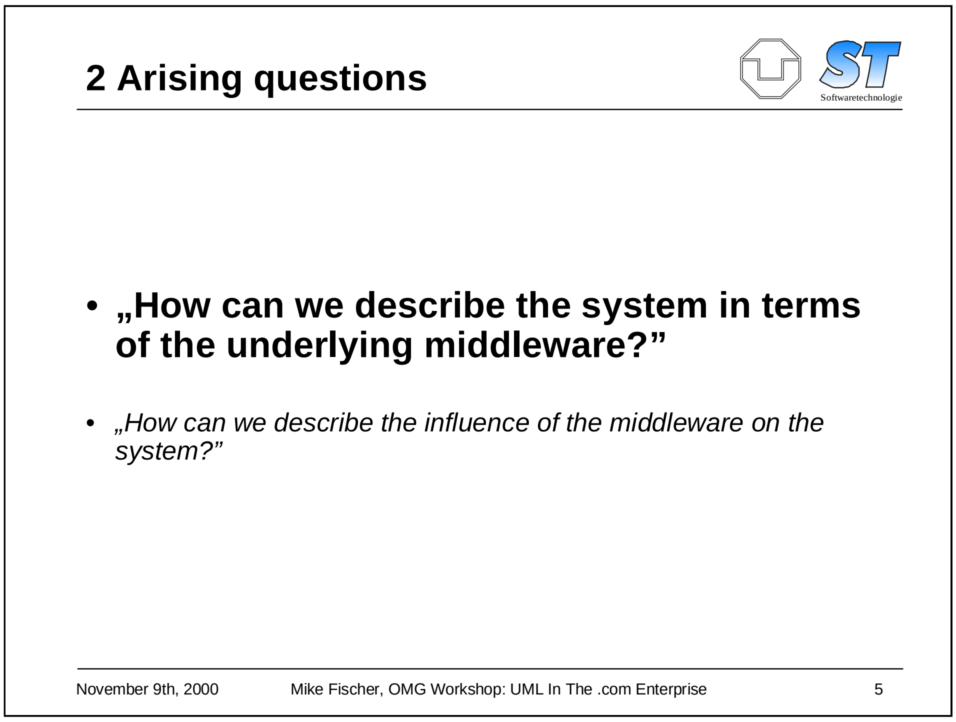
November 9th, 2000 Mike Fischer, OMG Workshop: UML In The .com Enterprise 4

As everybody knows, the UML is a general notation, and does not directly address modeling of (CORBA-) middleware-based applications ([OMG99b]). In the current practice a software designer can only use one UML diagram type that represents distribution: the deployment diagram. However, this is a diagram type corresponding to the implementation phase of the software development process, which shows run-time processing elements, and the software entities living on them ([OMG00a]).

Which diagram types can developers use in the other phases?

The analysis mainly focuses on a domain model, independently from the middleware issues.

During the design phase, other diagram types like class diagrams, sequence diagrams and activity diagrams can be used to model distributed middleware-based application systems. But currently, these diagram types do not provide any notations aimed to support the exact modeling of such systems. For this reason, the modeling of such systems often stop in an early step of the design phase and goes directly into the implementation phase. The observation of the realization of a distributed CORBA-based card game (Rommé), and the realization of a partly RMI-based virtual web-based community confirmed this statement.

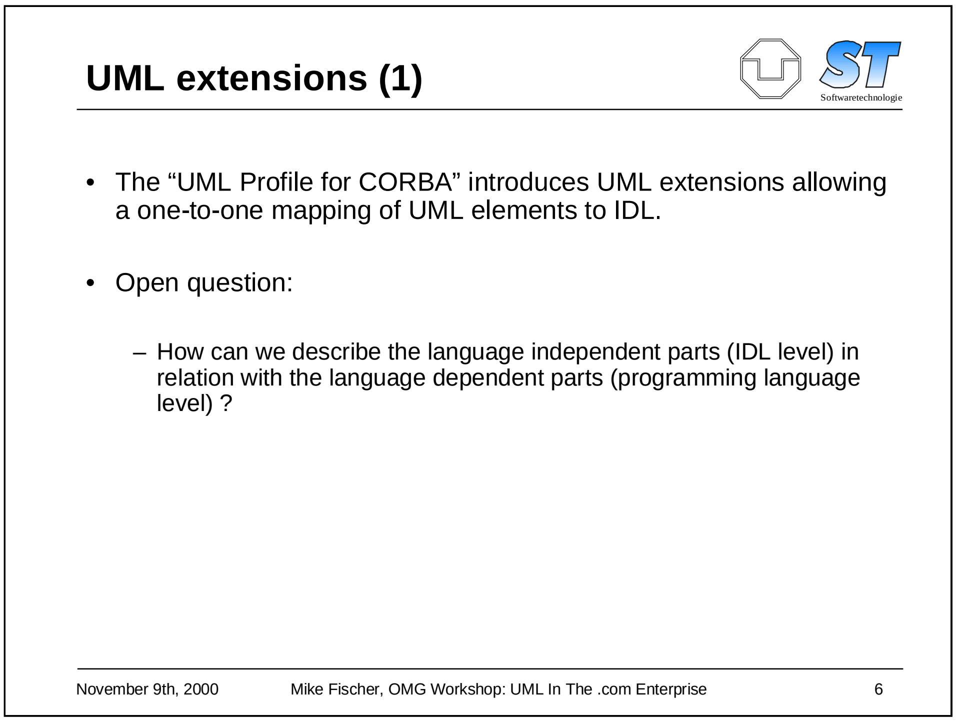


2 Arising questions

- **„How can we describe the system in terms of the underlying middleware?"**
- *„How can we describe the influence of the middleware on the system?"*

This presentation gives an abstraction for an expressive modeling of CORBA-based applications with UML, which helps to answer these both questions.

Firstly, we will discuss a solution for the first question.



UML extensions (1)

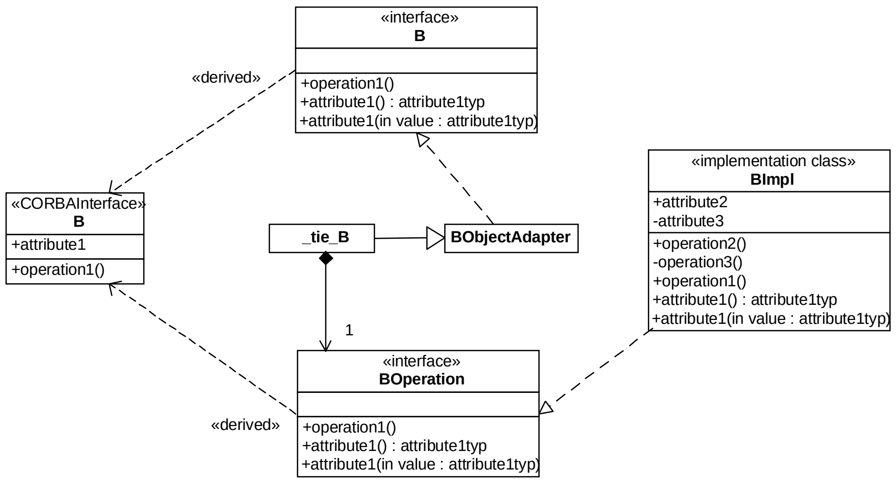
- The “UML Profile for CORBA” introduces UML extensions allowing a one-to-one mapping of UML elements to IDL.
- Open question:
  - How can we describe the language independent parts (IDL level) in relation with the language dependent parts (programming language level) ?

November 9th, 2000 Mike Fischer, OMG Workshop: UML In The .com Enterprise 6

In 1999, the OMG started a standardization process for a ”UML Profile for CORBA” ([OMG99d]), and so gave the answer to the first question: ”How can we describe the system in terms of the underlying middleware with UML?”. At the present time, the standardization process is finished ([OMG00c]), the specification is recently adopted. The proposed profile (What is a profile? - see [OMG99a]) introduces UML extensions (mainly based on stereotypes) allowing a one-to-one mapping of UML elements to IDL constructs like interfaces, structures or IDL data types, needed for a language independent description of the system. Accordingly, it is now possible to specify UML models in terms of CORBA IDL. In the appendix of this profile specification ([OMG00b]) a complete example (for the OMG Task and Session Service) shows that the introduced elements work perfectly for the specification of CORBAservices by the way of IDL constructs.

Is the problem solved? Not completely! Unfortunately, a model of a distributed middleware-based system is generally not only composed of a language independent description part (IDL constructs). Therefore, the proposed solution of the OMG ([OMG00b]) could be enhanced by artifacts and mechanisms that allow the description of the language dependent parts (e.g. the implementation of a CORBA object) in relation with the language independent parts.

Current solution



November 9th, 2000 Mike Fischer, OMG Workshop: UML In The .com Enterprise 7

Currently, with plain UML and the CORBA profile the relationship between an CORBA interface and its implementation can be modeled using a dependency stereotyped with `<<derived>>`. This solution has several drawbacks. The dependency under-specifies the present relationship, because:

- The servant is only indirectly derived from the CORBA interface (CIF).
- Some operations of the servant are indirectly derived from the CIF.
- The approach used to realize the servant is not specified.

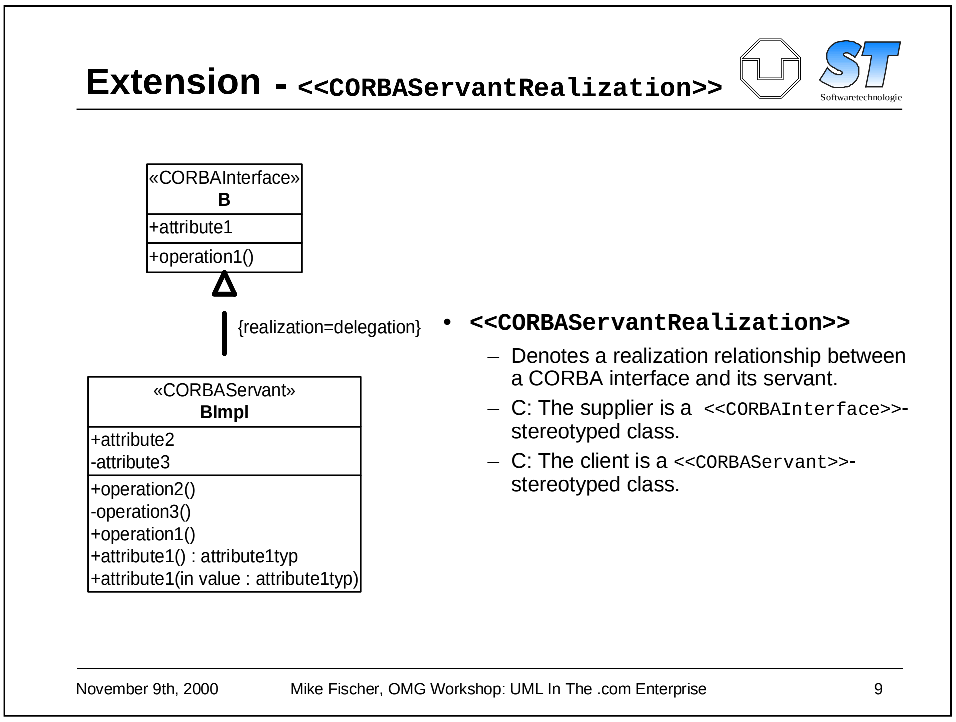
Current CASE tool environments ([Prin00a], [Rati00a], [Toge00a]) mostly ignore dependencies. Consequently, this results in useless code generation functions. The informal model is too vague! Furthermore, this is strengthened by the several, independent steps that have to be performed for code generation: A CASE tool generates IDL code, and language dependant code, whereas an IDL compiler generates language mappings including servant code. Both steps and tools used are independent from each other. Thus, the generated code can include name conflicts, and duplicates of some classes: one generated by the case tool, and the other one generated by the IDL compiler. But, each of the both classes include useful code! Using ”copy+paste” to get one class from the two classes is not a viable solution. Both steps must be executed under single control.

Modeling this relationship in more detail results in the model shown. It shows a class diagram in which a servant (`BImpl`) of a CORBA interface `B` is realized using the delegation approach. This model is close to reality. It avoids the under-specification. **However, the model is too large!** The model shows something like an implementation pattern used over and over again. In current practice there exist two well-known approaches for realizing a servant. These are the **generalization** and the **delegation** approach used here. The concrete realization of these approaches depends on the language mappings, but the overall idea is the same. Using generalization, the servant is derived from the skeleton (object adapter), which an IDL compiler generates for the remote object. Using delegation, the servant is decoupled from the inheritance hierarchy by a tie-object, which delegates all requests to the servant.

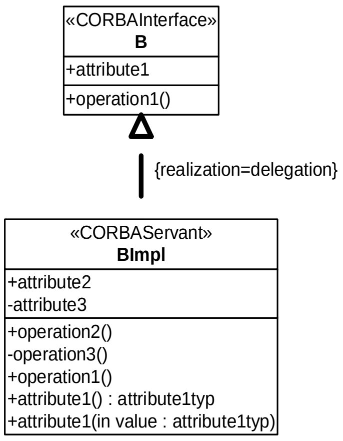
Therefore a way is required that allows, simply to specify the pattern, and avoid modeling as close grained as done for this model.

Recommendation

„ Abstract from the way, how the servant realizes the distributed object!"



Extension - `<<CORBAServantRealization>>`

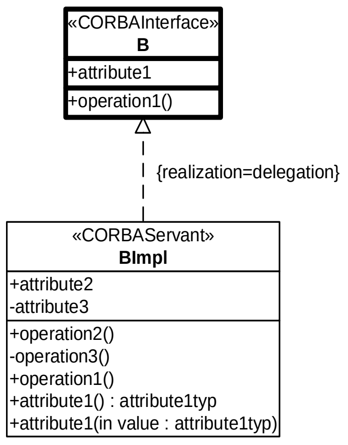


- **`<<CORBAServantRealization>>`**
  - Denotes a realization relationship between a CORBA interface and its servant.
  - C: The supplier is a `<<CORBAInterface>>`-stereotyped class.
  - C: The client is a `<<CORBAServant>>`-stereotyped class.

We realize the UML extensions using stereotypes ([Bern99a]) in order to specify specialized model elements, and using tagged values in order to specify properties, which are required for code generation functions.

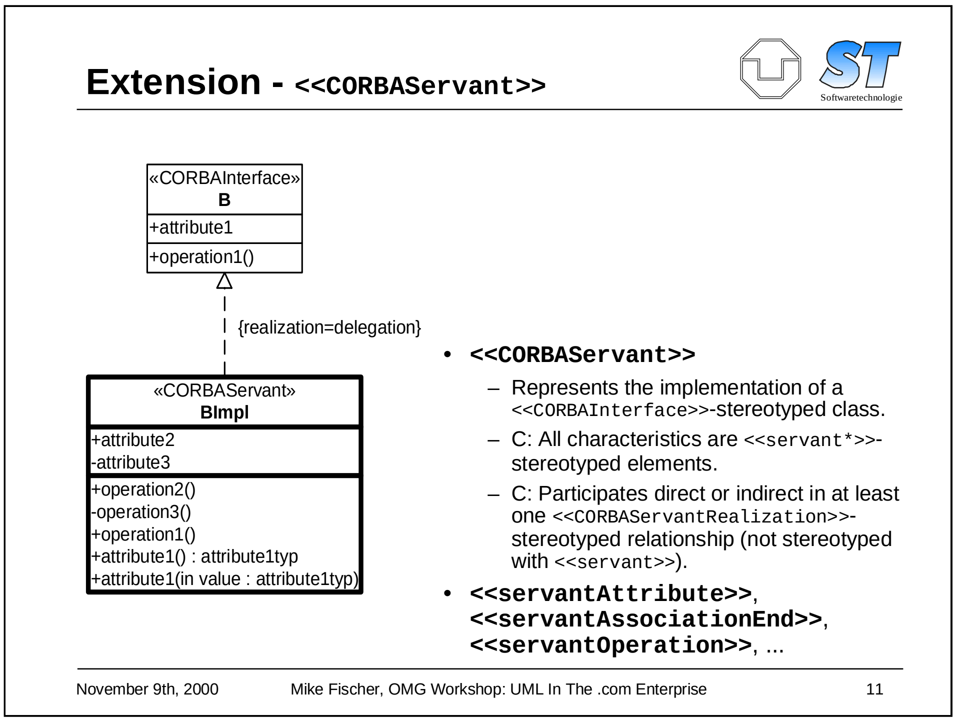
The left figure shows a shortcut for the previous model. Let us consider the relationship between a CORBA interface and its servant class. We can recognize that such a relationship is semantically a realization relationship in which the servant realizes an interface as specified by the CORBA interface. However, this is not done as directly as it is assumed for an usual `Core::Realization` metamodel element. E.g., within the generalization approach, the servant class is a subclass of a skeleton class (acting as object adapter) that implements an interface. This interface represents a language dependent representation of the CORBA interface, which is realized by the servant class. Therefore, describing this complex realization relationship, a stereotype `<<CORBAServantRealization>>` is introduced, which specializes the `Core::Realization` metamodel element. A tagged value called `realization` is applied to each element of this stereotype. This tag specifies the approach used to realize the servant, and thus, it shortcuts the model and realizes a more coarse grained view. Possible values of the tag are `generalization`, `delegation` and `tbd` (to be determine), whereas the first one is the default value.

Extension - `<<CORBAInterface>>`

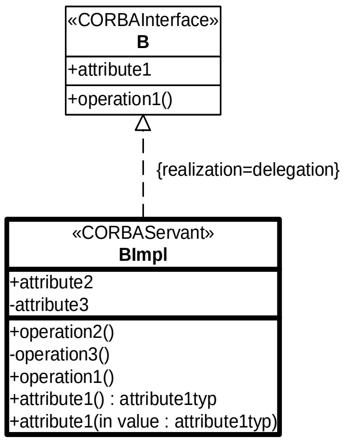


- **`<<CORBAInterface>>`**
  - Used with the same semantic as introduced in the “UML Profile for CORBA”, but with some additional constraints.
  - C: Each realization relationship in which an `<<CORBAInterface>>`-stereotyped class participates must be stereotyped with `<<CORBAServantRealization>>`

The <<CORBAInterface>> has additional constraints in order to join the CORBA profile with the extensions proposed here.



Extension - `<<CORBAServant>>`



- **`<<CORBAServant>>`**
  - Represents the implementation of a `<<CORBAInterface>>`-stereotyped class.
  - C: All characteristics are `<<servant*>>`-stereotyped elements.
  - C: Participates direct or indirect in at least one `<<CORBAServantRealization>>`-stereotyped relationship (not stereotyped with `<<servant>>`).
- **`<<servantAttribute>>`**, **`<<servantAssociationEnd>>`**, **`<<servantOperation>>`**, ...

If considering a servant, it must be (direct or indirect) a client of a `<<CORBAServant-Realization>>`-stereotyped realization relationship, and we can observe that the servant class type is a specialized `Core::Class` metamodel element. It has the same features as a class, with some additional constraints. These features are summarized by the stereotype `<<CORBAServant>>`.

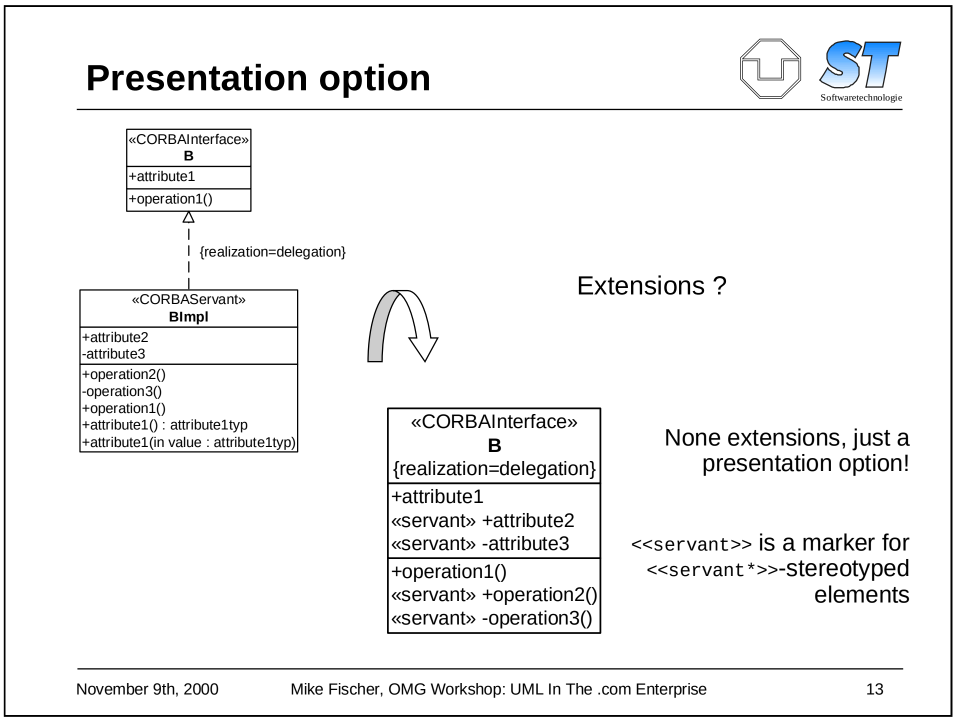
The tag `language` is applicable to `<<CORBAServant>>`-stereotyped classes, which specifies the implementation language of the class.

If we observe the servant in more detail we see that it comprises operations which are derived from the CORBA interface. In the model, these are three operations named with `operation1` and `attribute1`. As discussed previously, it is smart to keep such required information within the model. Thus, each element (e.g. attributes, operations, association ends) of a `<<CORBAServant>>`-stereotyped class is a specialized metamodel element (e.g. `<<servantAttribute>>`, `<<servant-Operation>>`, `<<servantAssociationEnd>>`), and could be shortly labeled with the stereotype marker `<<servant>>` (Generally, this marker is suppressed in the model view as shown in the model on the slide). A tagged value denoted as `derived` is applied to such an element, which indicates if the element is derived from the CORBA interface. This tag is also suppressed per default.

Recommendation

Softwaretechnologie

„Abstract from the servant"


«CORBAInterface»
B
+attribute1
+operation1()
{realization=delegation}
«CORBAServant»
BImpl
+attribute2
-attribute3
+operation2()
-operation3()
+operation1()
+attribute1() : attribute1typ
+attribute1(in value : attribute1typ)
«CORBAInterface»
B
{realization=delegation}
+attribute1
«servant» +attribute2
«servant» -attribute3
+operation1()
«servant» +operation2()
«servant» -operation3()
Softwaretechnologie

Presentation option

Extensions ?

None extensions, just a presentation option!

`<<servant>>` is a marker for `<<servant*>>`-stereotyped elements

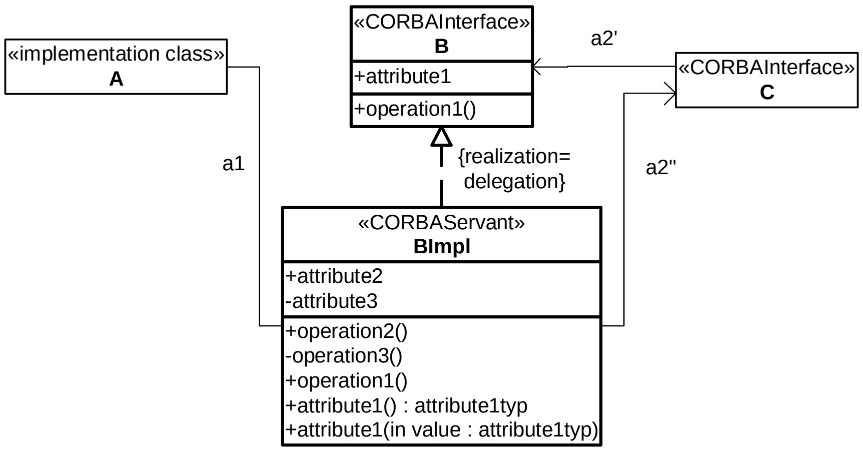
In some cases, the proposed solution in the left part on the slide could be regarded as oversized. This is true, if considering that most of the remote objects must be modeled in such a way, and thus, this solution results in a lot of additional classes shown in the entire model. Therefore we looks for an eligible way to get a more component-oriented view of the CORBA interface and its implementation in order to minimize the size of class diagrams.

To realize this requirement we considered several solutions. We must recognize that an extension of the metamodel by using the extension mechanisms of UML is not a feasible way for this case. It seems to be improper to integrate required extension elements into the existing metamodel, and if trying so the resulting stereotypes would have a complex set of constraints. Therefore, we described such a view as a presentation option within the stereotypes introduced above.

Such a presentation option may be realized by a CASE tool, however a model could be interchanged between CASE tools regardless if the presentation options are supported or not. Using a presentation option for `<<CORBAInterface>>`-stereotyped elements will allow showing `B` as displayed in the middle on the slide. Elements of the servant class (all elements with a name ending with characters ‘2’ or ‘3’ in this example), which are not denoted as derived from the CORBA interface are displayed with the marker `<<servant>>` within the CORBA interface description. Furthermore, the `realization` tag of the realization relationship is also shown within this description. Thus, the description of the servant itself can be suppressed.

Further simplifications are possible, e.g. if the tag `realization` is equal to its default value. In this case it can be suppressed in the model view. Changing between both views should be a feature of a case tool (possibly realized using techniques as described in [Rácz99a]).

Servant associations?

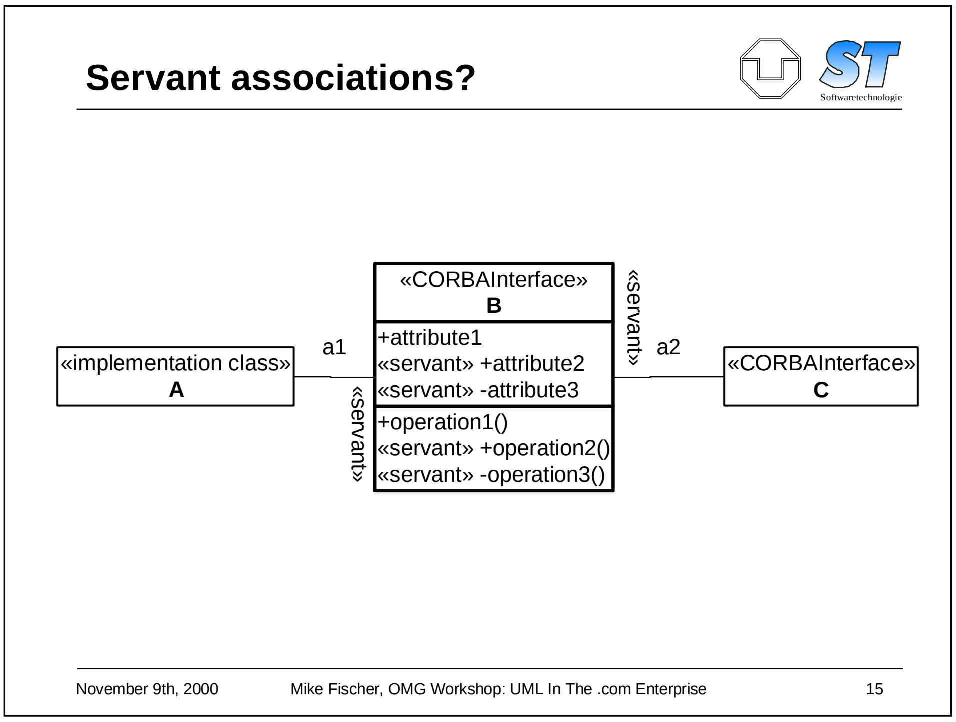


What is happening to the associations which are related to the servant?

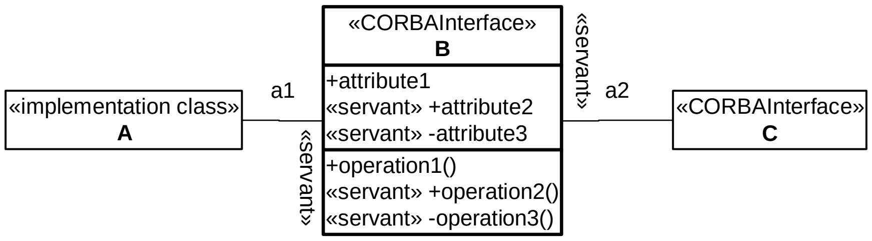
The proposed presentation option is also useful, if the servant (`BImpl`) takes part into relationships with other elements. This case is shown on the slide. Let us assume that the classes `B` and `C` are `<<CORBAInterface>>`-stereotyped classes, whereas `A` is a local, language dependant class required by `BImpl`.

Thus, in this example the association end of `a1` that is near to the servant is a `<<servant>>`-marked element (stereotype of `<<servantAssoziationEnd>>`).

The association end of `a2″` near to the servant is also denoted with this stereotype. Both associations belong to the servant.



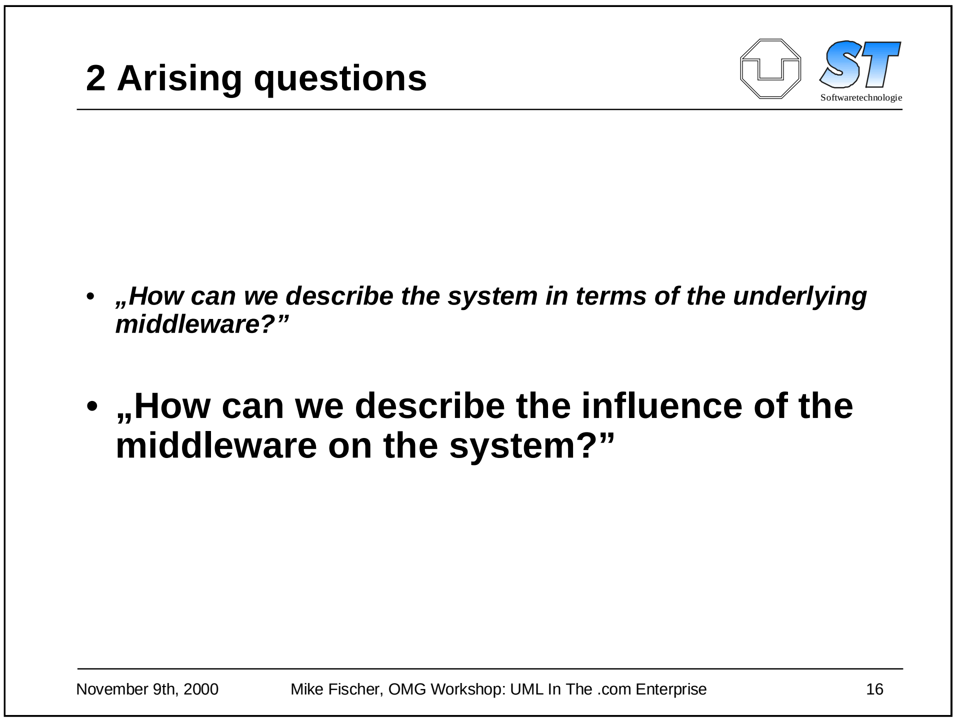
Servant associations?



Applying the proposed presentation option to the class diagram shown on the previous slide results in the diagram shown here.

The servant class `BImpl` is suppressed, and the association ends applied to the servant are shown assigned to the CORBA interface, however labeled with the marker `<<servant>>`.

The both unidirectional associations (`a2′` and `a2″`), are summarized as one bidirectional association, since they are related to the same classes.

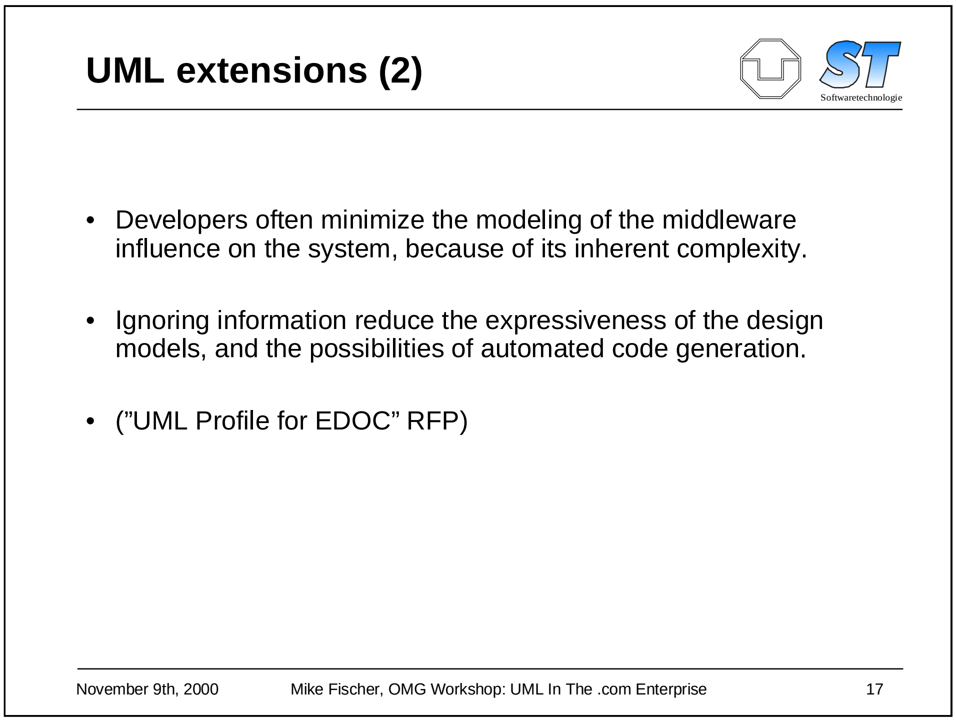


2 Arising questions

- ***„How can we describe the system in terms of the underlying middleware?”***
- **„How can we describe the influence of the middleware on the system?”**

November 9th, 2000 Mike Fischer, OMG Workshop: UML In The .com Enterprise 16

After the presentation of a solution for the first question, we will discuss about a solution for the second question.



UML extensions (2)

- Developers often minimize the modeling of the middleware influence on the system, because of its inherent complexity.
- Ignoring information reduce the expressiveness of the design models, and the possibilities of automated code generation.
- (”UML Profile for EDOC” RFP)

November 9th, 2000 Mike Fischer, OMG Workshop: UML In The .com Enterprise 17

Answering a question can reflect opposite interests, and this is what happens for this one.

First both points will be illustrated in more detail on the next slide.

Related work:

The work done in this paper falls also under the context of the ”UML Profile for Enterprise Distributed Object Computing” sought by a Request for Proposal ([OMG99b]). The mainly objective of the RFP is to elaborate a UML profile that permits:

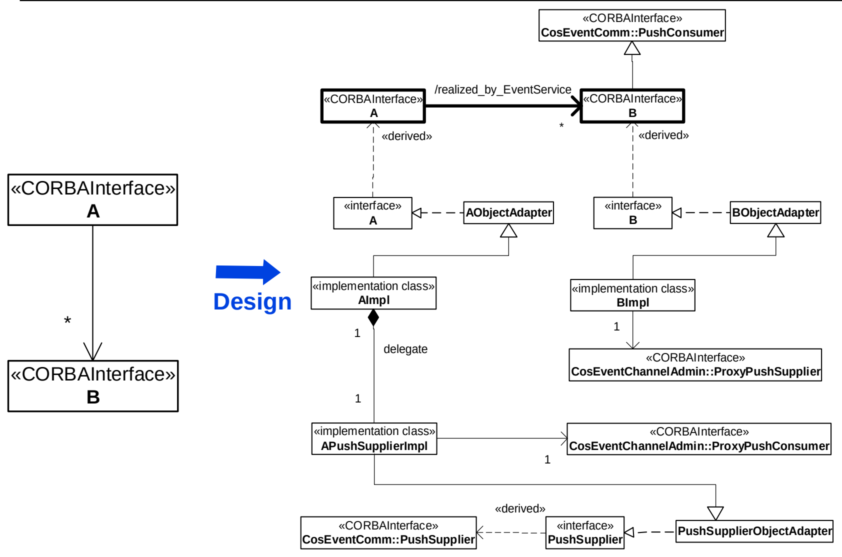
- The construction of an appropriate model providing connections back to the domain-dependant system requirements, and

- the driving of the design model into a distributed implementation.

This presentation focuses on the second aspect, whereas most of the initial submissions ([OMG99c]) refer to the first aspect. However, the presentation is not evolved in the context of the RFP, but gives a proposal for mapping design models into a CORBA-based implementation.

Considerations about a architectural modeling of the object transaction service are done in [Buss98].

Current solution



November 9th, 2000 Mike Fischer, OMG Workshop: UML In The .com Enterprise 18

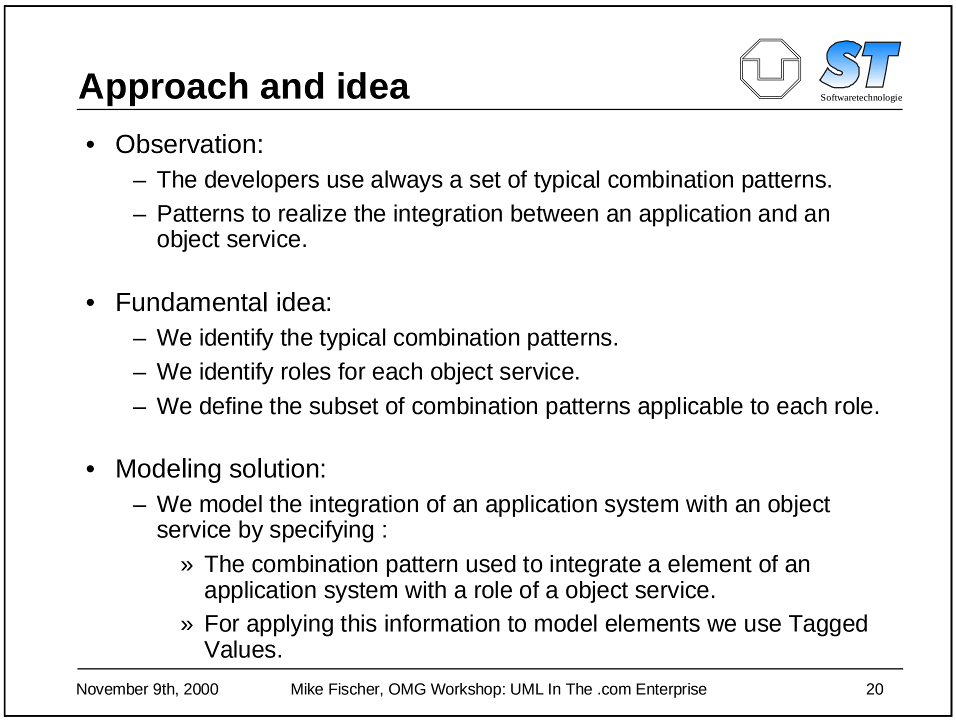
Let us give an example derived from our observation field. The class diagram on the left side shows an early design diagram where two remote objects A and B are linked by a unidirectional association, with a star multiplicity on one of the association ends. Later, a potential design decision could be to realize the star association end means the CORBA event service ([OMG98b], [Sieg96], [Sadi98a]). This achieves that the classes A and B are as independent as possible from each other. The result of a nearly complete model for this case is shown on the right. An Object of the CORBA interface A supplies events in which the objects of the CORBA interface B are interested in. By using the event service, A must realize the `PushSupplier` interface, whereas B must realize the `PushConsumer` interface of the event service.

Here, this is solved by generalization for B, and by delegation for A. The delegation as realized by A is useful, when A does not want to show the interface `PushSupplier` (that it has to realize) in its own interface. Now, consider that this not quite small model is only for one direction of the association! It becomes obvious that modeling in such detail produces bloated class diagrams. Such a solution is therefore inapplicable! This is why, at the moment, in practice the design problem is shift in the implementation phase, and the solution consists in enhancing the documentation or commenting the UML model. This solution is only informal, and the resulting model has several drawbacks:
-To understand the model, it is required to be familiar with the development style guides (of the company or the development team) in order to find the information concerning the inchoate model, and additionally be a CORBA specialist. Information are perhaps part of the documentation, however they are not available at a glance.
-Generating precise and correct code from the design model is not possible (in analogy to the discussion from slide 7).

Another level of abstraction to describe the application system, the (CORBA-based) middleware and its influences is required. Applying the recommended UML extensions (1) will downsize the class diagram on the right. Yet, a further minimized view could be helpful. As there does not exist (standardized) notations in order to describe this in a proper way, this presentation proposes a solution in the following.

Recommendation

„Abstract from the role that a distributed object plays in a CORBAservice interaction!"



Approach and idea

- Observation:
  - The developers use always a set of typical combination patterns.
  - Patterns to realize the integration between an application and an object service.
- Fundamental idea:
  - We identify the typical combination patterns.
  - We identify roles for each object service.
  - We define the subset of combination patterns applicable to each role.
- Modeling solution:
  - We model the integration of an application system with an object service by specifying :
    - » The combination pattern used to integrate a element of an application system with a role of a object service.
    - » For applying this information to model elements we use Tagged Values.

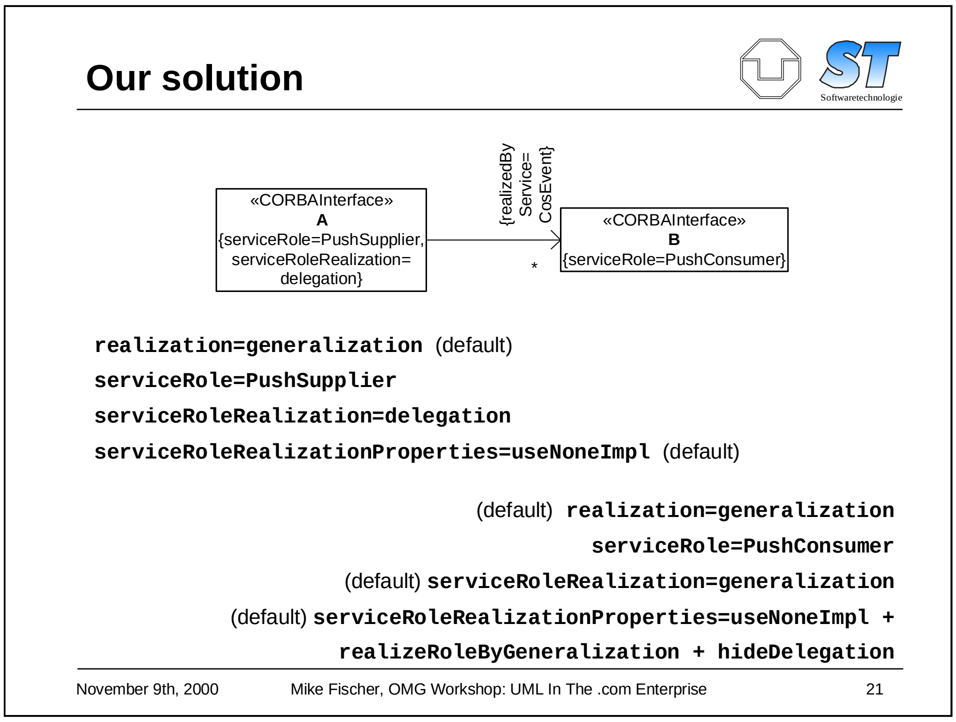
November 9th, 2000 Mike Fischer, OMG Workshop: UML In The .com Enterprise 20

We discussed how to model the relationship between a CORBA interface, and its implementation. This lays the foundation for the ongoing discussion: ”How can we describe the influence of the middleware on the system?”. We will continue this discussion by using the CORBA object services as an example. Our observations of the realization of medium-sized software projects show that it is difficult to model how a system makes use of object services. As already discussed, another level of abstraction to describe such an integration is required to prevent bloated class diagrams as shown on the previous slide. Doing so is not an easy task. The difficulties result from several facts: Firstly, object services consist of a large set of interfaces and other data structures, therefore they can not be seen at a glance! Secondly, a designer has to recognize which one of them are important for the model of the system, and therefore should be shown as part of the model. This fact is strengthened, because the CORBAservice specifications as well as the reference documents of object service implementations mainly consist of the specification or description of the interfaces of the object service. Only a very small part of these documents deals with the overall service idea, or with the question how to integrate the service with an application. A identification of roles or connection points, that describe the use of the object service, is unfortunately not included in these documents.

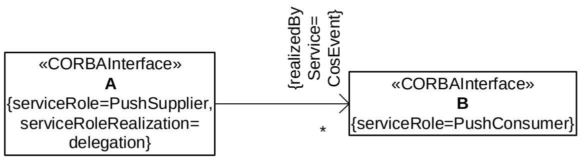
Our recommendation is to: “Abstract from...”

We use the technical term “pattern” in an analogous manner as it is used in the classical design patterns ([Gamm95]).

A role denotes a connection point toward an object service, and will mostly be an interface within an object service.



Our solution



«CORBAInterface»
A
{serviceRole=PushSupplier, serviceRoleRealization= delegation}
{realizedBy Service= CosEvent}
*
«CORBAInterface»
B
{serviceRole=PushConsumer}

`realization=generalization` (default)
`serviceRole=PushSupplier`
`serviceRoleRealization=delegation`
`serviceRoleRealizationProperties=useNoneImpl` (default)

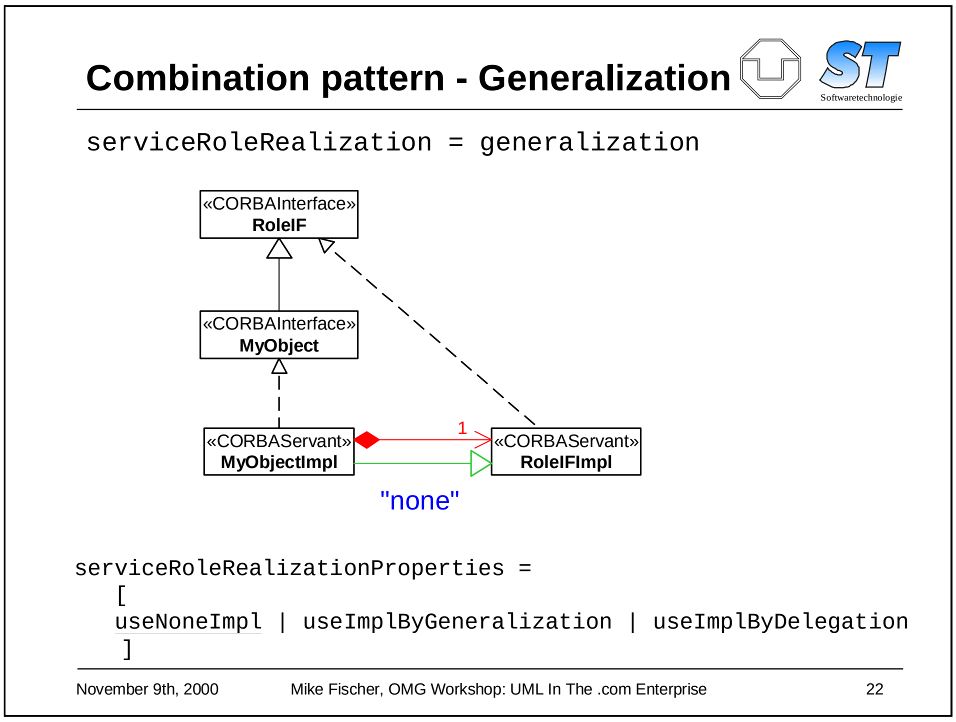
(default) **`realization=generalization`**
`serviceRole=PushConsumer`
(default) **`serviceRoleRealization=generalization`**
(default) **`serviceRoleRealizationProperties=useNoneImpl + realizeRoleByGeneralization + hideDelegation`**

The class diagram shown on this slide is the proposed shortcut for the big class diagram from slide 18. The abstractions used in the model are based on:

* the UML extensions (1), and the introduced presentation options (the servants are suppressed here). For the servants of both interfaces, the realization approach is from type `generalization`. This is the default approach, therefore the realization tag is suppressed in both CORBA interface descriptions.

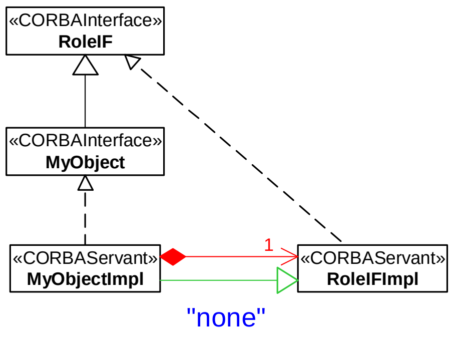
* the UML extensions (2), used to describe the integration of the CORBA interfaces `A` and `B` with the event service for this example. As specified by the tag `serviceRole`, `A` is integrated with the role `PushSupplier`, and `B` with the role `PushConsumer` of the event service. The tag `serviceRoleRealization` specifies which combination pattern is used to realize the integration. For `A` it is a pattern denoted as `delegation`, and for `B` a pattern denoted as `generalization` (both patterns are explained in more detail in the following slides). We defined the `generalization` pattern as the default value for this tag, that is why this tag is suppressed in the interface description of `B`. The tag `serviceRoleRealizationProperties` is used to configure the combination pattern applied. For `A` and `B`, the value `useNoneImpl` of the tag indicates, that none default implementation of the role is used or available within the object service implementation. The additional values for `B` specify that its role is realized using the generalization approach, and that the delegation relationship is not exposed to the interface of `B`.

The association between `A` and `B` is realized for this model using the event service. Therefore, the depicted association is a derived association that only has informal character. To emphasize this character, a tagged value could be attached to the association as done in the model (`realizedByService=CosEvent`).



Combination pattern - Generalization

`serviceRoleRealization = generalization`



```
serviceRoleRealizationProperties =
   [
   useNoneImpl | useImplByGeneralization | useImplByDelegation
    ]
```

November 9th, 2000 Mike Fischer, OMG Workshop: UML In The .com Enterprise 22

How do we achieve the suggested solution? A more detailed discussion:

*** Point one of our fundamental idea:**

We have identified 3 major combination patterns, which are explained on this slide, and both following slides. Generally, the combination patterns are independent from the object service!

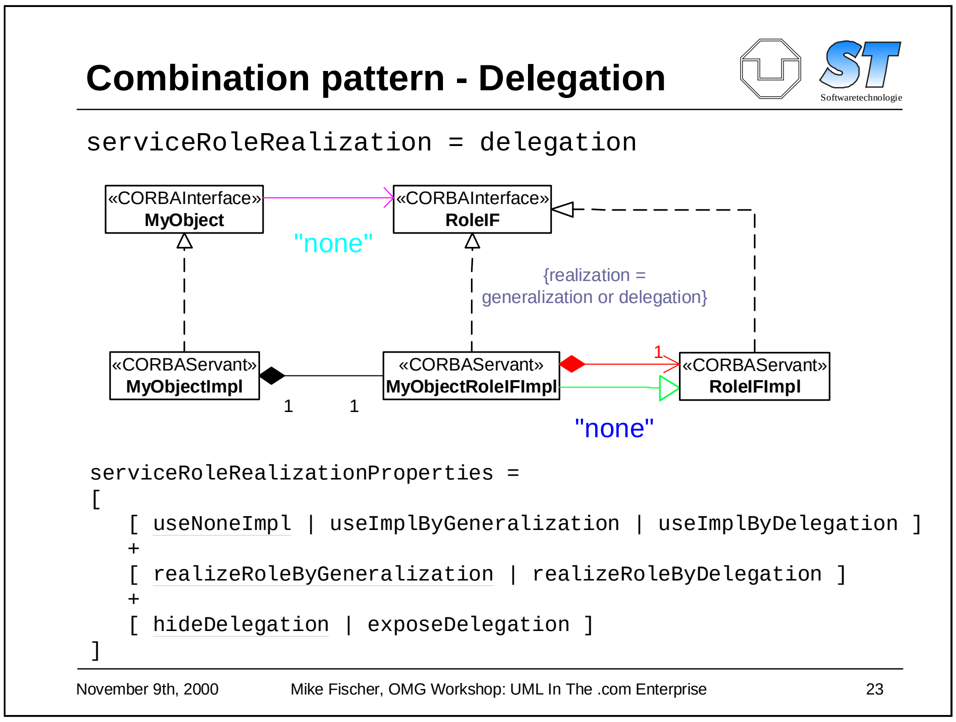
Together with the pattern we introduce tagged values used to apply the required information to UML model elements (The tagged values introduced here, can be applied to the <<`CORBAInterface`>>-stereotyped classes). Generally, regarding the patterns there are two tagged value types: `serviceRoleRealization` and `serviceRoleRealizationProperties`. The first one specifies the combination pattern applied, the second one specifies some properties (described using [DeMa97]), which allow to configure the pattern in more detail.

The first combination pattern introduced here is denoted as `generalization`. An application and an object service is integrated by using interface inheritance. In other words: An element of the application system must implement an interface specified within an object service. This is shown between `RoleIF` and `MyObject`.

The first part of the presentation has introduced an abstraction to describe the way how `MyObjectImpl` (servant) realizes the CORBA interface `MyObject`. Such issues are therefore not part of a combination pattern.

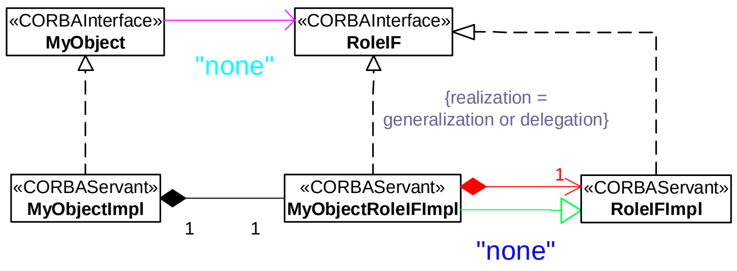
For realizing the servant, a developer could make use of a default implementation (`RoleIFImpl`) for the role interface (`RoleIF`). It depends on the concrete object service if such an implementation exists, and it depends on design decisions of the system designer, if it is used. Generally, the default implementation can be used by delegation or by class generalization. Such issues are specified within the tag `serviceRoleRealizationProperties`.

Observed by: Event Service, ...



Combination pattern - Delegation

`serviceRoleRealization = delegation`



«CORBAInterface» MyObject
«CORBAInterface» RoleIF
"none"
{realization = generalization or delegation}
«CORBAServant» MyObjectImpl
«CORBAServant» MyObjectRoleIFImpl
«CORBAServant» RoleIFImpl
1
1
1
"none"

```
serviceRoleRealizationProperties =
[
   [ useNoneImpl | useImplByGeneralization | useImplByDelegation ]
   +
   [ realizeRoleByGeneralization | realizeRoleByDelegation ]
   +
   [ hideDelegation | exposeDelegation ]
]
```

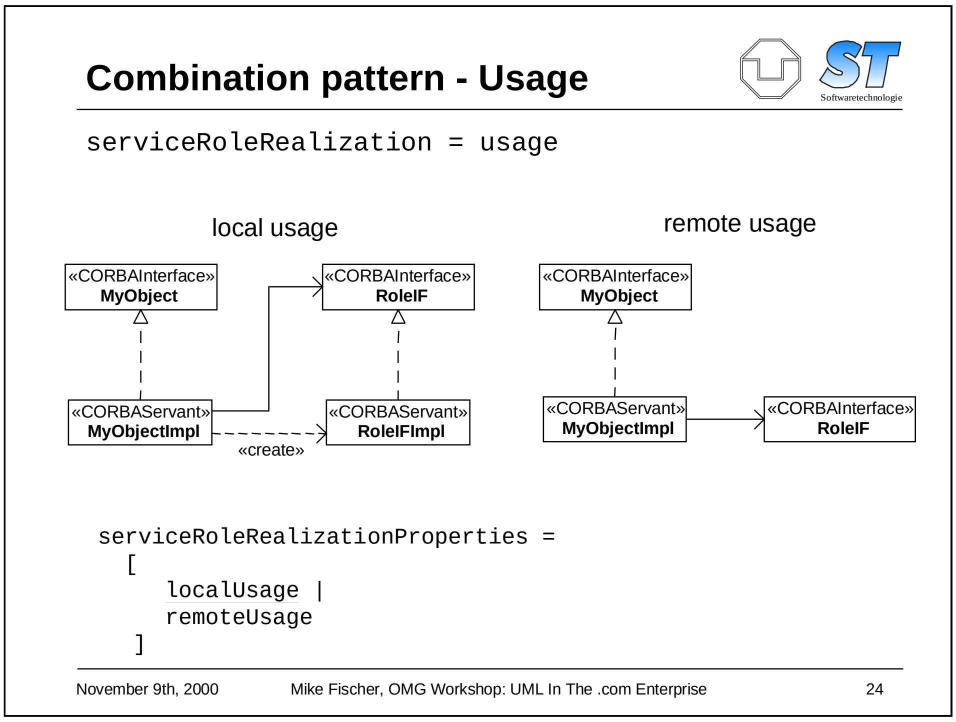
The next combination pattern is denoted as `delegation`. It describes in almost the same manner as the previous pattern type a strong integration of application system, and object service.

In the case of this pattern (similar to the previous pattern), an element (`MyObject`) of the application system must fulfil a role (`RoleIF`). The `delegation` pattern is useful, if the designer does not want to show the interface of the role together with the interface of the element. Both interfaces are decoupled by using composition on the servant level, between the servant of the element (`MyObjectImpl`) and the servant of the role interface (`MyObjectRoleIFImpl`).

To realize the servant of the role interface, a developer can use the default implementation for this servant (`RoleIFImpl`), if such an implementation exists. In this case, the developer can specify the approach used to realized the role implementation. Since our approach aims to suppress all elements apart from `MyObject`, specifying the approach is required in this case.

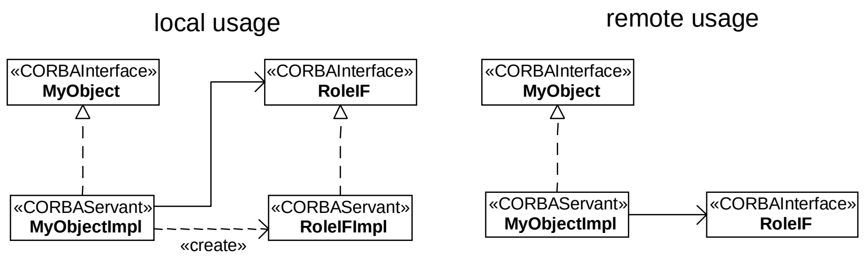
A further option of the pattern is the possibility to show the delegation at the interface level by having an attribute of the role interface type within the element of the application (unidirectional association between `MyObject` and `RoleIF`).

Observed by: Collection Service, ...



Combination pattern - Usage

`serviceRoleRealization = usage`



local usage
«CORBAInterface» MyObject
«CORBAInterface» RoleIF
«CORBAServant» MyObjectImpl
«CORBAServant» RoleIFImpl
«create»
remote usage
«CORBAInterface» MyObject
«CORBAServant» MyObjectImpl
«CORBAInterface» RoleIF

```
serviceRoleRealizationProperties =
  [
     localUsage |
     remoteUsage
   ]
```

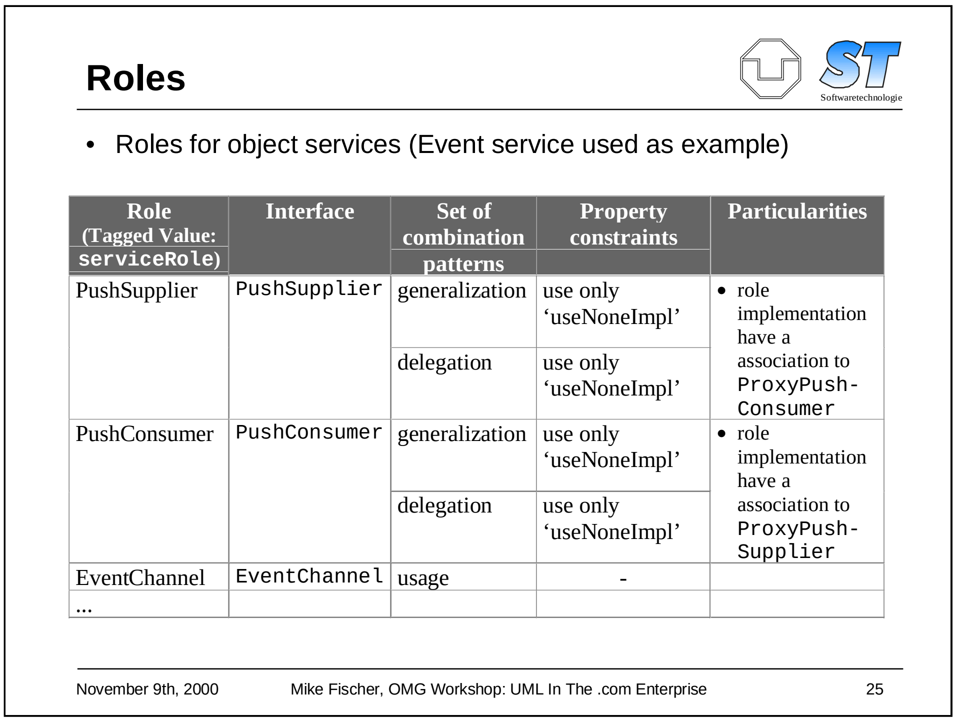
The last combination pattern is denoted as `usage`. In contrast to the both previous patterns, this pattern describes only a weak integration of application system, and object service, which could changed during the lifetime of the system elements.

This pattern describes that an element of the application system makes use of a role of an object service. It occurs in 2 different kinds. These kinds can be distinguished using the `serviceRoleRealizationProperties` tag.

We describe the first kind as local usage (shown on the left). In such a case, the servant of the application element (`MyObjectImpl`) creates in its own local space an instance of the servant (`RoleIFImpl`) of the role interface, and uses this servant.

The second kind is denoted as remote usage (shown on the right). In this case, the servant of the application element (`MyObjectImpl`) accesses remotely the role. It can get the reference to the role object using usual finding approaches or by creation via an object factory.

Observed by: Naming Service, ...



Roles

- Roles for object services (Event service used as example)

Role (Tagged Value: `serviceRole`)	**Interface**	**Set of combination patterns**	**Property constraints**	**Particularities**
PushSupplier	`PushSupplier`	generalization	use only ‘useNoneImpl’	• role implementation have a association to `ProxyPush-Consumer`
		delegation	use only ‘useNoneImpl’	
PushConsumer	`PushConsumer`	generalization	use only ‘useNoneImpl’	• role implementation have a association to `ProxyPush-Supplier`
		delegation	use only ‘useNoneImpl’	
EventChannel	`EventChannel`	usage	-	
...				

We are now going to discuss point two and tree of our fundamental idea: “Identifying roles for each object service.” and “Defining the subset of combination patterns applicable to each role.”

Our idea is, that a role denotes a connection point toward an object service. A role helps a designer to realize which elements of the object service are important for the integration with applications, and therefore should be considered in an understandable design model. Roles can be mostly mapped to an interface within the object service, as it is shown for the event service in the table above. However, in some cases more general roles are quite possible (e.g. collection service).

Let us consider the `PushConsumer` role in more detail. This role is an interface that must be implemented from CORBA objects which are interested in events submitted by one or more suppliers (via an event channel). This interface is defined within the CORBA service specification, yet it must be implemented within the application. To realize such a strong integration, we have observed two combination pattern. The most obvious pattern is the `generalization` pattern (slide 22), describing a CORBA interface generalization. However also the delegation pattern (slide 23) could be practical. The implementation of the application element and the role are separated within this pattern. The third pattern (`usage`) is not applicable in this case, because it does not allow such a strong integration as required here.

Additionally, some constraints restrict the usage of the applicable patterns. Since the `PushConsumer` interface is very small (it comprise only two operations), there does not exist any default implementation for this interface. Consequently, only the `useNoneImpl` value of the `serviceRoleRealizationProperties` tag is valid. Considering implementations of the `PushConsumer` interface leads to the observation, that they should comprise an association to a `ProxyPushSupplier`. This observation is also described within the role definition in the “Particularities” column.

The table above should build a schema for further role definitions.

Summary and Perspectives

Softwaretechnologie

- The shortly introduced elements will allow to:
  - Hide the required information about influence of middleware instead of ignore them.
  - Switch between different levels of detail if these elements are supported in CASE tools.
  - Build the basis for a code generation functionality, which produce smarter code as the current ones.
- What remains to be done?
  - Further verification of proposed combination patterns.
  - Integration into a CASE tool.
  - Considering diagram types other than the class diagram.
  - Do the same work for another middleware or componentware.
  - Abstract from the experiences, and find a more general approach.

Questions and Feedback ?

Softwaretechnologie

- **Questions and Feedback are welcome !**

- Acknowlegments:
  - Many thanks to Anne Thomas for helping to develop this presentation.

References

[Bern99a]	S. Berner, M. Glinz, S. Joos. A Classification of Stereotypes for Object-Oriented Modeling Languages. Proceedings of the Second International Conference <<UML>>'99 – The Unified Modeling Language, Fort Collins, USA, October 1999.
[Buss98]	S.Busse, S. Tai. Software Architectural Modeling of the CORBA Object Transaction Service. Proceedings of the 22nd International Computer Software and Applications Conference (COMPSAC'98) IEEE Computer Society, Vienna, Austria, August 1998.
[DeMa79]	T. DeMarco. Structured Analysis and System Specification. Yourdon Press, 1979.
[Gamm95]	E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, Inc., 1995.
[OMG00a]	OMG. UML Modeling Language Specification, Version 1.3. OMG document number: formal/00-03.01, March 2000.
[OMG00b]	OMG. UML Profile for CORBA. Joint Revised Submission. OMG document number: ad/00-05-07, June 2000.
[OMG00c]	OMG. URL: http://www.omg.org/technology/documents/recent/omg_modeling.htm, 2000.
[OMG98a]	OMG. The Common Object Request Broker: Architecture and Specification. OMG document number: formal/98-12-01, December 1998.
[OMG98b]	OMG. CORBAservices: Common Object Services Specification. OMG document number: formal/98-12-09, December 1998.
[OMG99a]	OMG. White paper on the Profile mechanism, Version 1.0. OMG document number: ad/99-04.07, April 1999.

References

[OMG99b]	OMG. UML™ Profile for Enterprise Distributed Object Computing. Request for Proposal. OMG document number: ad/99-03-10, March 1999.
[OMG99c]	OMG. UML™ Profile for Enterprise Distributed Object Computing. Initial Submissions. OMG document numbers: ad/99-10-07, ad/99-10-10, ad/99-10-09, ad/99-11-05, ad/99-10-19, ad/99-10-08, ad/99-10-17, ad/99-10-20, October /November 1999.
[OMG99d]	OMG. A UML™ Profile for CORBA. Request for Proposal. OMG document number: ad/99-03-11, March 1999.
[Prin00a]	Princeton Softech, Inc. Select Enterprise: http://www.princetonsoftech.com/products/enterprise.asp, 2000.
[Rácz99a]	F. D. Rácz, K. Koskimies. Tool-Supported Compression of UML Class Diagrams. Proceedings of the Second International Conference <<UML>>'99 – The Unified Modeling Language, Fort Collins, USA, October 1999.
[Rati00a]	Rational Software Corporation. Rational Rose: http://www.rational.com/products/rose/index.jtmpl, 2000.
[Sadi98a]	W. Sadiq, F. Cummins. Developing Business Systems with CORBA. Cambridge University Press, 1998.
[Sieg96]	J. Siegel. CORBA Fundamentals and Programming. John Wiley & Sons, Inc., 1996.
[Toge00a]	TogetherSoft Corporation. Together Control Center: http://www.togethersoft.com/together/togetherCC.html, 2000