Polymorphism

This chapter explains how to use polymorphism in the E2E Bridge. However, it is no substitute for a basic education about polymorphism as found in many textbooks of object-oriented programming.

The way polymorphism is implemented follows more the tradition of C++ and C# than Java or Ruby. The reason is efficiency and documentation. In our approach only operations that are overridable are really looked up at runtime and - maybe even more important – operations that are being overridden or override other operations must explicitly marked by stereotypes.

On this Page:

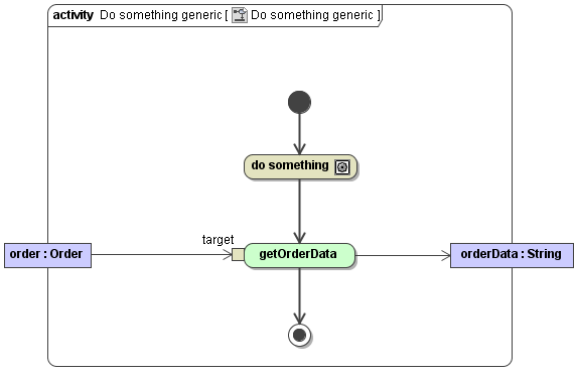
Abstract Operations

Example File (Builder project Basic Modeling/ClassOperation):

<your example path>\Basic Modeling\ClassOperation\uml\polymAbstractOperations.xml

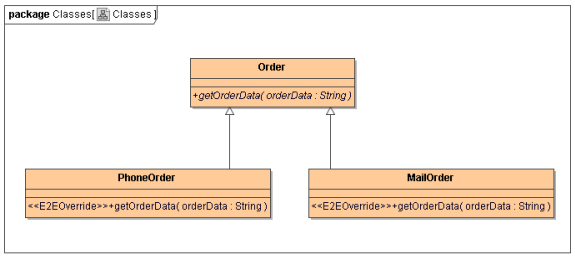
We start with a simple example. Say, you have different order backend systems to be accessed. Before getting the order data, you must do some generic calculations. This might be represented in an activity diagram as follows:

Figure: Generic Activity



Let us further assume the operation **getOrderData** that fetches the desired in-formation is member of the class **Order**. However, you also know that the implementation of **getOrderData** depends on the accessed backend system. Therefore, we specify that the **getOrderData** operation of class **Order** is abstract (indicated by an italic font in Figure below).

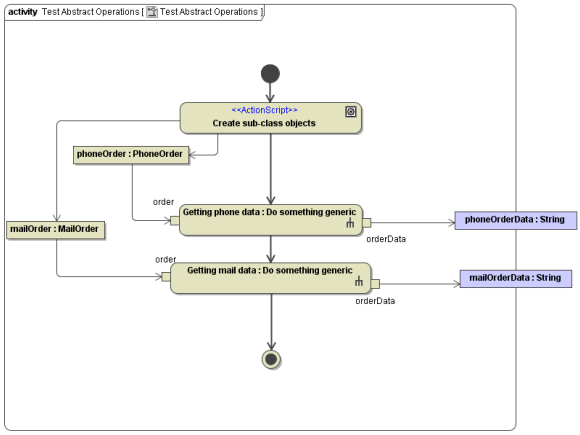
Figure: Declaring Abstract Operations



This means, only its interface is defined. Additionally, we specify sub-classes of Order: **PhoneOrder** and **MailOrder**. They may represent systems containing orders that arrived by phone respectively by mail. In our example, we want to get the same order data independent of the backend system. Thus, these classes contain an operation having the same interface (means same operation name and parameter types) as the abstract **getOrderData** operation. The stereotype <<E2EOverride>> indicates that these operations override the behavior of a base class operation, namely the **getOrderData** operation of the class Order.

What is the meaning of all this? Assume you call the activity diagram defined in Figure Generic_Activity using first a **MailOrder** object and then a **PhoneOrder** object as input:

Figure: Calling the Overriden Operations



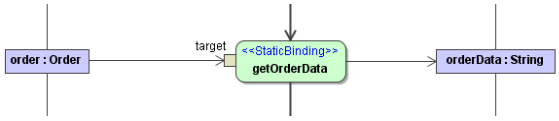
When executing the action **Getting phone data** the input object **phoneOrder** will bring its own implementation of **getOrderData**. This means, the operation being called in the activity **Do something generic** is the operation implemented in the **PhoneOrder** class.
Overriding abstract operations is actually mandatory. The reason is that abstract operations do not have an implementation by their own. If an abstract operation is not overridden by a sub-class the compiler will report an error.

> Note, if a class contains abstract operations it is not possible to create them since we would then get an object having an undefined behavior.

Static Binding

Assume, you want to call the **getOrderData** operation of class **Order** independent whether the runtime object is of type **Order**, **MailOrder** or **PhoneOrder**.
In the E2E Bridge context, this is called static binding. It can be achieved by creating an operation call action and assigning this action the stereotype <<StaticBinding>>:

Figure: Defining static binding

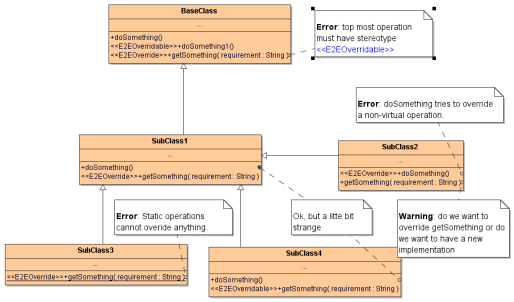


When finding this stereotype, the compiler will bind the operation call to the chosen operation at compile time.

Frequent Errors

Most frequent errors are forgetting to set the correct stereotypes or declaring overridable operations static. The following diagram shows some of these errors:

Figure: Most Frequent Errors



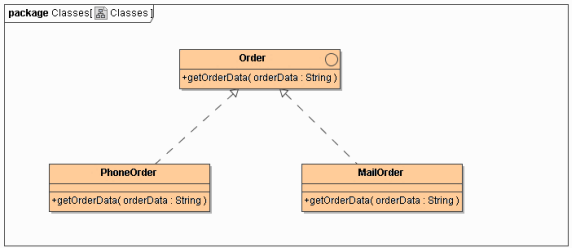
Interfaces

Example File (Builder project Basic Modeling/ClassOperation):

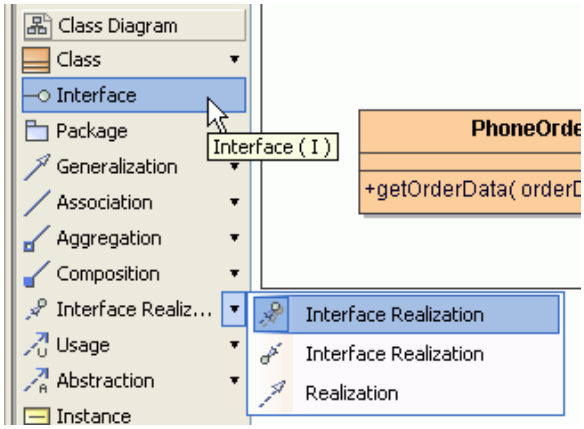
<your example path>\Basic Modeling\ClassOperation\uml\polymInterfaceOperations.xml

Related to polymorphism are interfaces. For example, if you want just define the set of operations an object must provide, it would be possible to define a class containing abstract operations only. All classes deriving from this base class must then implement these abstract operations. However, there is a more elegant way of achieving the same by using interfaces. In this case, the classes realize the interface:

Figure: Using UML interfaces to Specify Class Operations



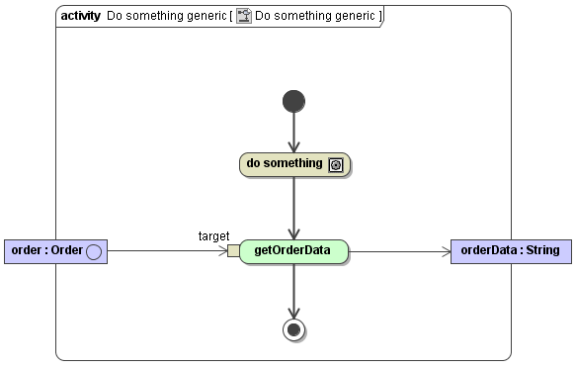
The main difference to using a set of abstract operations is that a class can realize more than one interface but it can inherit from just one ancestor class.



To draw an interface select **Interface** from the toolbar and specify it. Then, the implementation of this interface is defined by drawing an **Interface Realization** relationship between the class and the interface (see right figure).

Calling an interface operation is done exactly the same way as calling a class operation. The only difference is that the target classifier of the operation action is an interface instead of a class:

Figure: Calling an interface operation



When invoking the flow above, any object realizing the **Order** interface will be accepted as **order** object.