

# Accessing and modifying model element properties

Element properties can be accessed with simple setters and getters. For example *NamedElement.name* property:

```
com.nomagic.uml2.ext.magicdraw.classes.mdkernel.NamedElement el = ...;
String name = el.getName();
el.setName("new name");
```

## Related pages

- [Session management](#)
- [Checking element editing permissions](#)

## Container Properties

Our modeling tools use a composite structure of the model.

Every model element is a container and contains its own children and knows about its own parent.

A model element parent can be accessed with the *com.nomagic.uml2.ext.magicdraw.classes.mdkernel.Element.getOwner()* method. Owned children can be received with the *Element.getOwnedElement()* method. Different types of children are stored in separate container properties.

You can access these container properties by names that are described in the UML specification. The *getOwnedElement()* method collects all children from all inner container properties.

The container properties modification and iteration is straightforward using the *java.util.Collection* interface. Property change events are fired automatically when container properties are modified.

Containers implement subsets and unions constraints from the UML metamodel specification. This explains how the modification of one container can affect other containers. Make sure you understand subsets and unions in the UML metamodel. If you want to add some inner Element to the union collection, you need to add it into a specific subset of *union*.

Some containers are read-only. This is true for the most of DERIVED UML metamodel properties.

Some derived properties are editable. For example, *Element.ownedElement* is editable.

It is enough to set one UML meta-association property value and an opposite property will be set too. For example, adding a *Class* into a *Package* can be done in two ways:

```
Class myClass= ...;
Package myPackage ...;
myClass.setOwner(myPackage);
```

or

```
myPackage.getOwnedElement().add(myClass);
```

## Accessing elements in container properties

The following example illustrates retrieving children of model elements:

```
Element el = ...;
for(Element element : el.getOwnedElement())
{
    // work with element
}
```

## Modifying elements in container

Use standard *java.util.Collection* or *java.util.List* methods:

```
modelElement.get<SomeContainer>().add(child);
modelElement.get<SomeContainer>().remove(child);
```

## Navigable opposite references

Standard UML metamodel defines all references between Elements in "pairs" - for each primary navigable reference there is an opposite reference. Majority of those opposite references are not navigable (are not exposed in opposite metaclass).

For example *TypedElement* references it's *Type* by reference *type*, but *Type* does not expose *TypedElements* typed by it. Such approach is valid semantically, but makes model traversing challenging.

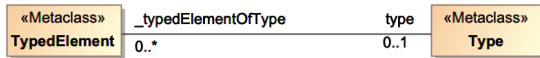
Our UML metamodel implementation does not follow this restriction and exposes all opposite references.

Some opposite references have "friendly" names defined by UML specification.

For example *NamedElement.clientDependency* opposite reference name is *Dependency.client*.

Other opposite references names are constructed using a pattern *\_**<primaryReferenceDefiningMetaclassName>**Of<primaryReferenceName>*. Such naming pattern is used even latest version of UML specification provides different names for these opposite references.

For example there is a primary reference *type* defined in *TypedElement*. Opposite reference is defined in *Type* has name *\_typedElementOfType*.



Setters and getters are provided for each opposite reference. For example

```
Primary reference Element.getAppliedStereotype(),
opposite Stereotype.get_stereotypedElement()
```

```
Primary reference TypedElement.getType(),
opposite Type.get_typedElementOfType()
```