Modeling and metamodeling in Model Driven Development

Metamodeling directed relationships in UML

Warsaw, May 14-15th 2009

Gonzalo Génova

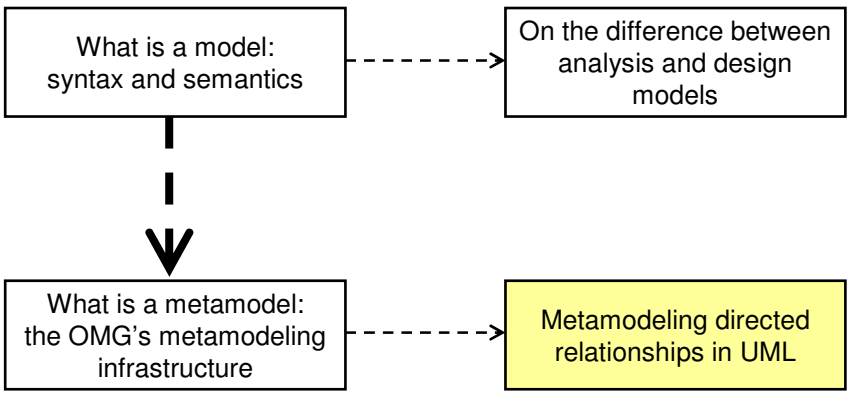
ggenova@inf.uc3m.es

http://www.kr.inf.uc3m.es/ggenova/

Knowledge Reuse Group

Universidad Carlos III de Madrid

Structure of the seminar



Sources

- My own ideas and elaboration.
  - Thanks to Kyriakos Anastasakis and Dan Chiorean for help with OCL.
  - In preparation for Journal of Visual Languages and Computing.

We hope from a scholar to tell something others have not seen, and he himself does not see very well.
(Daniel Innerarity)

Table of contents

Introduction

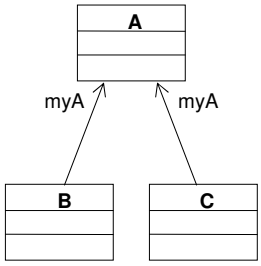
Purpose

- The original problem:
  - How to find the ***children classifiers*** of a given classifier?
  - Does UML provide a way to do this?
- The problem is found in any tool that manipulates models.
  - ***CASE tools***: if I modify a class, which subclasses will be affected?
  - ***Retrieval tools***: if I search for a class in a repository, shall I be satisfied by any answer that contains the subclasses as well?
- My purpose is to “convince” the audience that:
  - There is ***a flaw*** in the UML metamodel, and
  - The source of this flaw is ***a misunderstanding*** of metamodeling levels.
- In order to “really” convince, I expect from the audience:
  - ***Skepticism*** (maybe I am wrong), and
  - ***Openmindedness*** (maybe I am right!).

Before continuing: what is association navigability

- In ***UML 1.x:***
  - Not defined (!)
- In ***my PhD Thesis:***
  - Navigability specifies the ability of an instance of the source class to access the instances of the target class by means of the association instances (links) that connect them.
  - Navigability is closely related to the ability of sending messages, so that very often this two concepts are identified.
- In ***UML 2.x:***
  - Navigability means instances participating in links at runtime (instances of an association) can be accessed efficiently from instances participating in links at the other ends of the association. (...). Note that tools operating on UML models are not prevented from navigating associations from non-navigable ends.

```
public class A { }
// A knows nobody

public class B extends A {
      A myA;}
// B knows A

public class C extends A {
      A myA;}
// C knows A
```

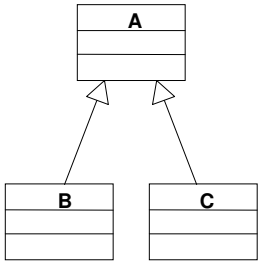


The original problem

The original problem

- How to find the ***children classifiers*** of a given classifier?
- Does UML provide a way to do this?
- At least, ***nUML*** does not. ***Eclipse?***

```
public class A { }

public class B extends A { }

public class C extends A { }

// A does not know B and C.
// B and C know A.
```



The answer in the UML metamodel (1)

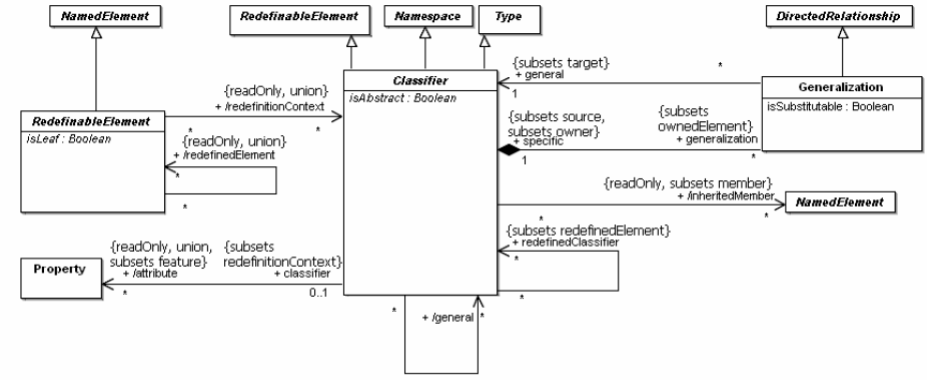
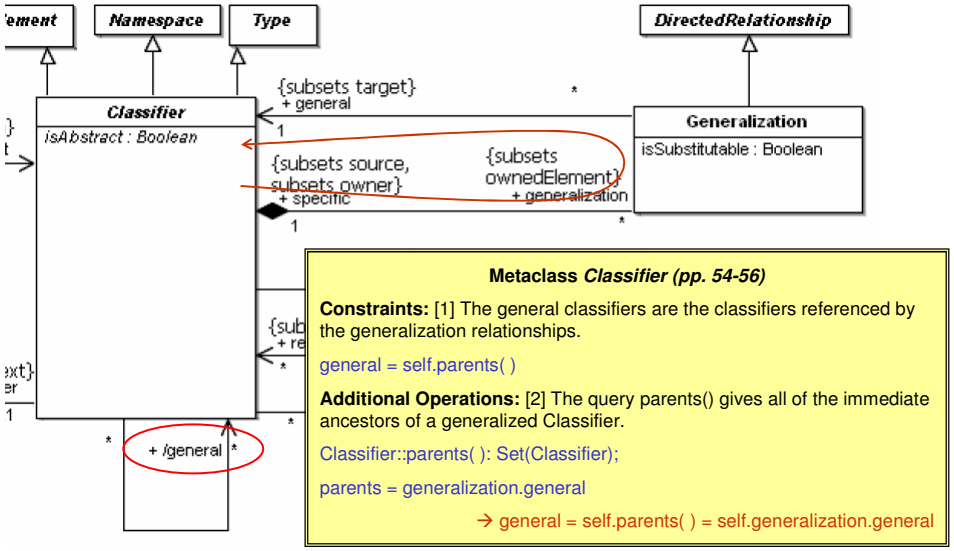


Figure 7.9 - Classifiers diagram of the Kernel package

Unified Modeling Language: Superstructure, version 2.1.1, February 2007

Knowledge Reuse

The answer in the UML metamodel (2)



Metaclass *Classifier (pp. 54-56)*

Constraints: [1] The general classifiers are the classifiers referenced by the generalization relationships.

general = self.parents( )

Additional Operations: [2] The query parents() gives all of the immediate ancestors of a generalized Classifier.

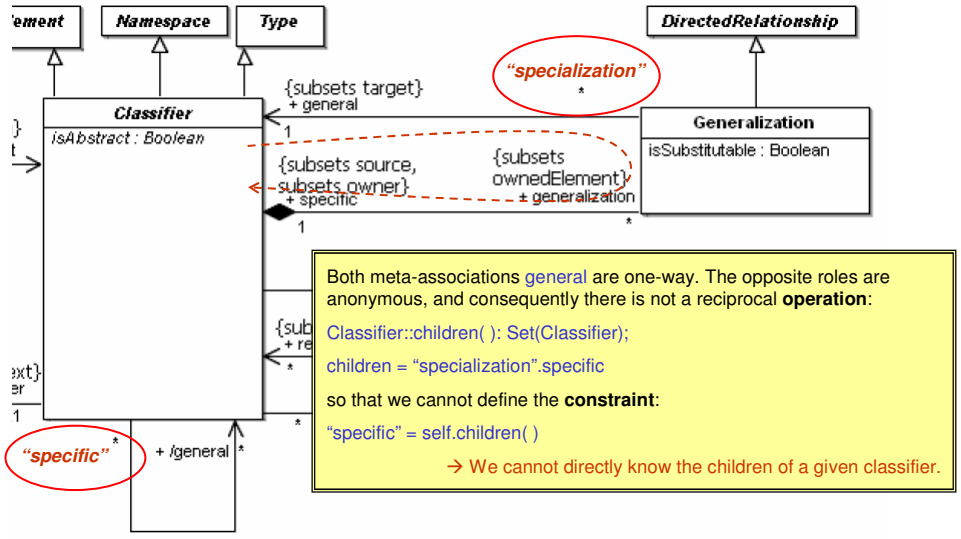
Classifier::parents( ): Set(Classifier);

parents = generalization.general

→ general = self.parents( ) = self.generalization.general

Knowledge Reuse

The answer in the UML metamodel (3)



Both meta-associations general are one-way. The opposite roles are anonymous, and consequently there is not a reciprocal **operation**:

Classifier::children( ): Set(Classifier);

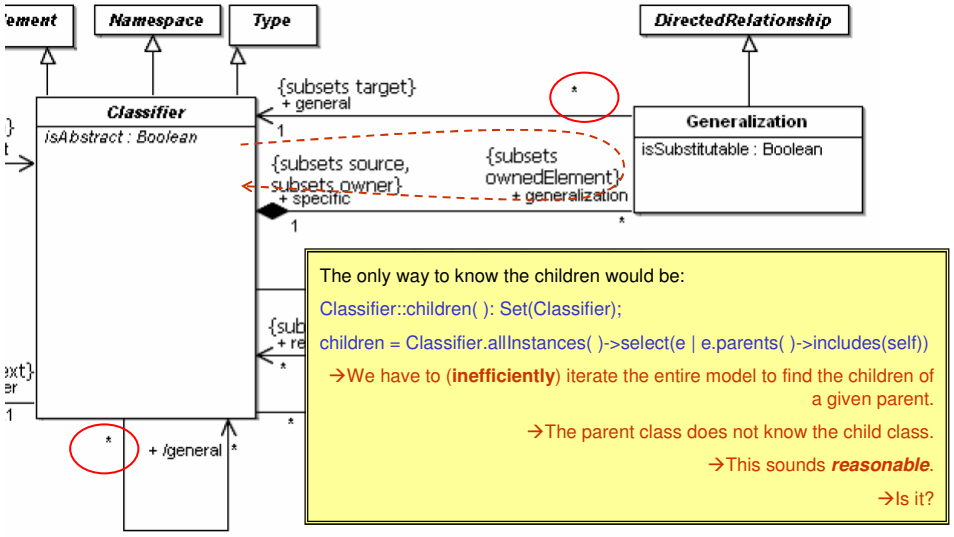
children = "specialization".specific

so that we cannot define the **constraint**:

"specific" = self.children( )

→ We cannot directly know the children of a given classifier.

The answer in the UML metamodel (4)



The only way to know the children would be:

Classifier::children( ): Set(Classifier);

children = Classifier.allInstances( )->select(e | e.parents( )->includes(self))

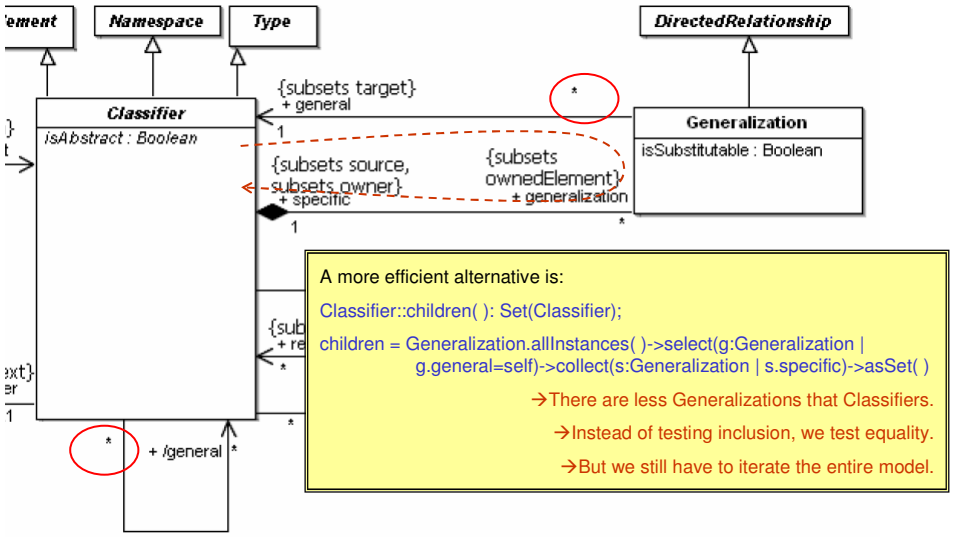
→We have to (**inefficiently**) iterate the entire model to find the children of a given parent.

→The parent class does not know the child class.

→This sounds ***reasonable***.

→Is it?

The answer in the UML metamodel (5)



A more efficient alternative is:

Classifier::children( ): Set(Classifier);

children = Generalization.allInstances( )->select(g:Generalization | g.general=self)->collect(s:Generalization | s.specific)->asSet( )

→There are less Generalizations that Classifiers.

→Instead of testing inclusion, we test equality.

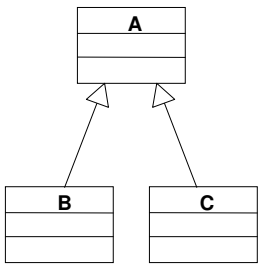
→But we still have to iterate the entire model.

Semantics of generalization

Generalization and dependency / directionality

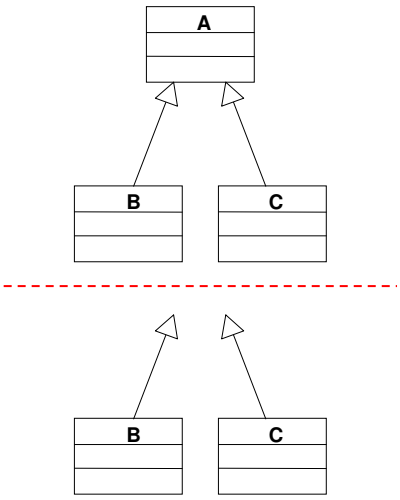
- In the code:
  - The child class knows the parent, but not viceversa.
  - Expressed in the model as a directed relationship (a generalization).
- Every generalization ***induces*** a dependency subclass → superclass.
  - A generalization ***is not*** a dependency, but induces it.
- In every dependency:
  - The dependent element requires the presence of the independent element.
  - Changes in the independent element may affect the dependent element.

```
public class A { }
// A knows nobody

public class B extends A { }
// B knows A

public class C extends A { }
// C knows A
```

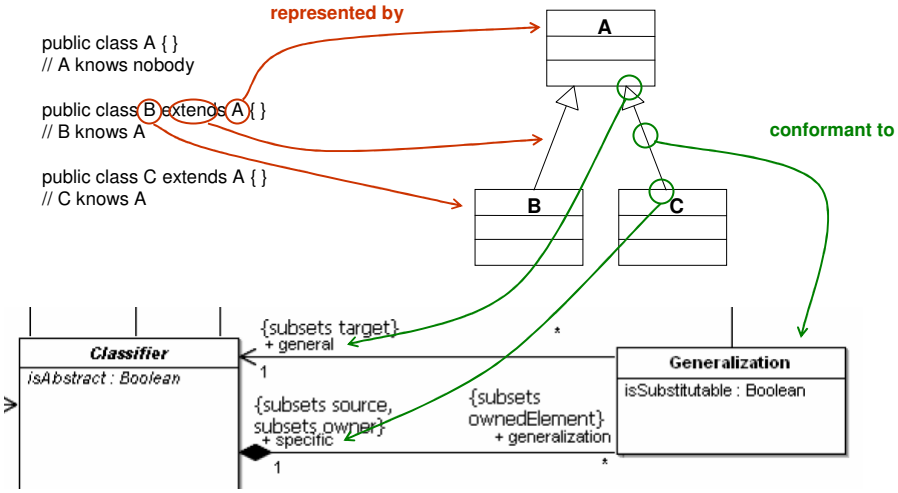


Represented reality, model, and metamodel

- Now forget the code (modeled reality) and concentrate on the model itself.
  - Do not look at the classes represented by rectangles, ***look at the rectangles***.
  - Is it true that rectangle A does not know rectangles B and C?
- What happens if rectangle A is deleted?
  - The arrows are deleted, too.
  - Thus, ***rectangle A knew the arrows***.
- We should be very careful to distinguish:
  - The code (text) that is ***represented by*** the model.
  - The model (graph) that is ***conformant to*** the rules expressed in the metamodel.

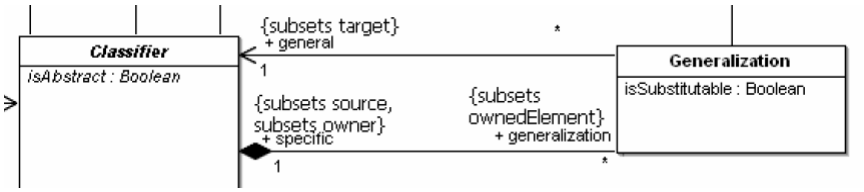


The two MM relationships: represented-by / conformant-to



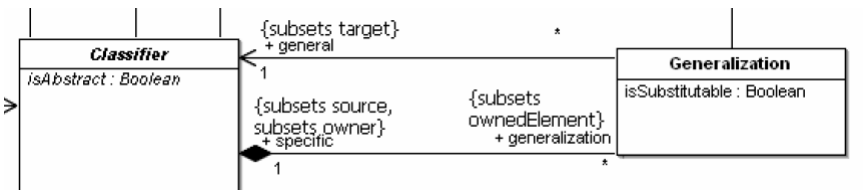
Metamodel of generalization (1)

- The metamodel defines the rules that models must conform to.
  - Models are considered ***linguistic expressions*** (in a graphical language).
  - The abstract syntax specifies ***legal combinations*** of model elements.
- The generalization arrow is metamodeled as follows:
  - A metaclass that represents the ***generalization itself*** (Generalization).
  - A meta-association that represents its ***tail*** (specific).
  - A meta-association that represents its ***head*** (general).



Metamodel of generalization (2)

- The ***directionality*** of the generalization arrow is expressed in the metamodeled by two meta-associations.
  - Why should one of these be one-way? Why not both of them?
  - Because generalization is one-way? Well, this is only a hypothesis…
- The metamodel is ***unduly*** trying to represent a feature of the semantic domain, instead of the features of the language.



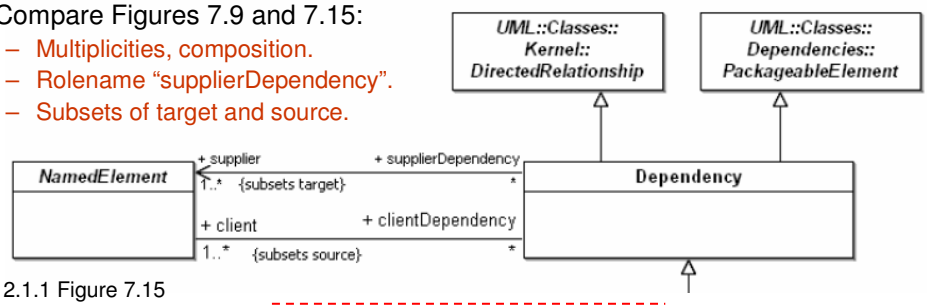
Mixing metamodeling levels

- M0 – the represented reality (the code)
  - A generalization is one-way, from the subclass to the superclass.
  - The general element must ***not*** know the specific element.
- M1 – the model representing the code
  - The directionality of generalization (***M0***) is represented by an arrow (***M1***).
  - The arrow, the linguistic element, the graphical symbol, is an object that ***expresses*** a direction but ***has not*** a direction itself.
  - ***The arrow knows the two boxes, and the two boxes know the arrow***.
- M2 – the metamodel (the rules of the modeling language)
  - The directionality of generalization is ***sufficiently represented*** by two meta-associations. This should be enough.
  - Introducing directionality in these meta-associations ***mixes M2-M1-M0***.
  - Practical tools will ***disobey the metamodel*** in this point.

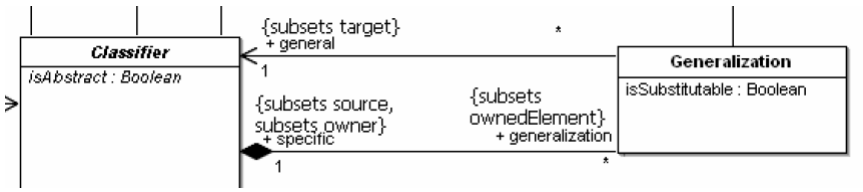
Other directed relationships

The case of dependency

- Compare Figures 7.9 and 7.15:
  - Multiplicities, composition.
  - Rolename “supplierDependency”.
  - Subsets of target and source.



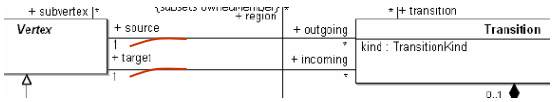
UML 2.1.1 Figure 7.15

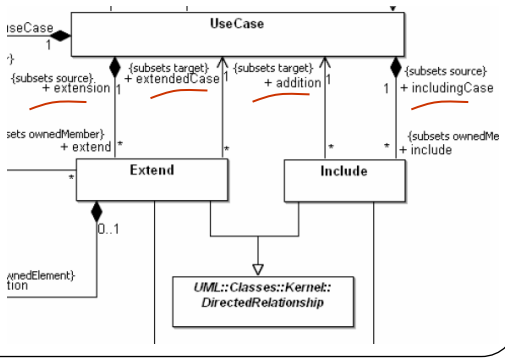


UML 2.1.1 Figure 7.9

Other directed relationships

- Chapter 7 (Classes):
  - Subtypes of dependency: Abstraction, Realization, Substitution, Usage.
  - PackageMerge, PackageImport, ElementImport.
- Chapter 8 (Components):
  - ComponentRealization.
- Chapter 15 (State Machines)
  - ProtocolConformance.
  - ***But not Transition!*** Even though it is defined as: “A transition is ***a directed relationship*** between a source vertex and a target vertex” (p. 568).
- Chapter 16 (Use Cases)
  - Include, Extend.
- Chapter 17 (Auxiliary Constructs)
  - InformationFlow, TemplateBinding.
- Chapter 18 (Profiles)
  - ProfileApplication.





The general case: DirectedRelationship

- In some cases, ***target*** and ***source*** multiplicity has been restricted from 1..* to 1.
- In most cases, navigability for the ***opposite of source*** has been added.
- Is this legal according to ***Liskov's substitution principle?*** Yes.

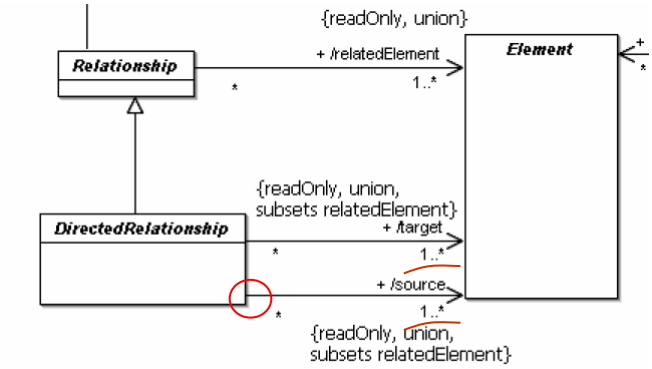
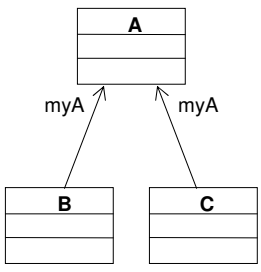


Figure 7.3 - Root diagram of the Kernel package

The case of one-way associations: dependency / directionality

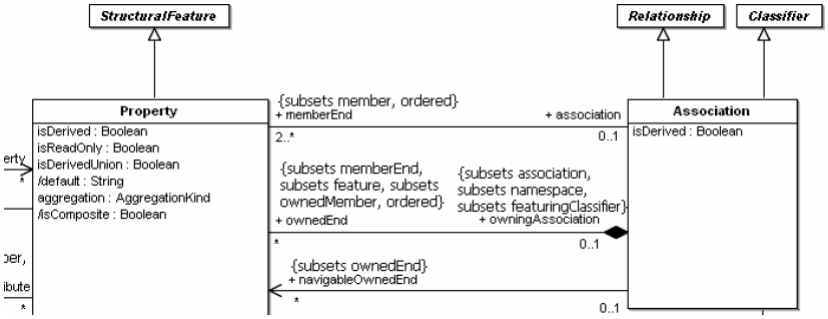
- In the code:
  - The source class knows the target class, but not viceversa.
  - Expressed in the model as a directed relationship (a one-way association).
- Every one-way association ***induces*** a dependency source → target.
  - A one-way association ***is not*** a dependency, but induces it.
- Again, what happens if rectangle A is deleted?
  - The arrows representing incoming associations are deleted, too.
  - Thus, ***rectangle A knew the arrows***.

```
public class A { }
// A knows nobody

public class B extends A {
      A myA;}
// B knows A

public class C extends A {
      A myA;}
// C knows A
```



Metamodel of associations

- Even though Association is a kind of Relationship, ***memberEnd*** is not a subset of ***relatedElement***, but of Namespace::member.
- Navigable ends are a subset of the association owned ends.
- In this case, navigability has not been added but restricted (***Liskov?*** No.)
- Non-navigable ends know the association, ***but they don't know that they cannot navigate it!***
- Directionality at M1 did not require strict navigability at M2.



Conclusions

Conclusions

- Graphical modeling languages:
  - A graphical modeling language is composed of graphical elements.
  - Graphical language elements that are connected ***know each other***.
- OMG's metamodeling levels:
  - A model (M1) expresses the properties of a certain modeled reality (M0).
  - A metamodel (M2) expresses the properties of a modeling language (M1).
  - The metamodel (***abstract syntax***) should only express the legal combination of modeling elements and the relationships among them.
  - Should the metamodel express the properties of the ***semantic domain*** where the modeling language is used? Yes, but the directionality of generalization is ***sufficiently represented*** by two meta-associations. This should be enough.
  - Introducing directionality in these meta-associations ***mixes M2-M1-M0***.
- Remember: "Note that tools operating on UML models are not prevented from navigating associations from non-navigable ends". ***Then why specify it?***
- Should we conclude that ***the metamodel must not contain one-way meta-associations?***

Questions?