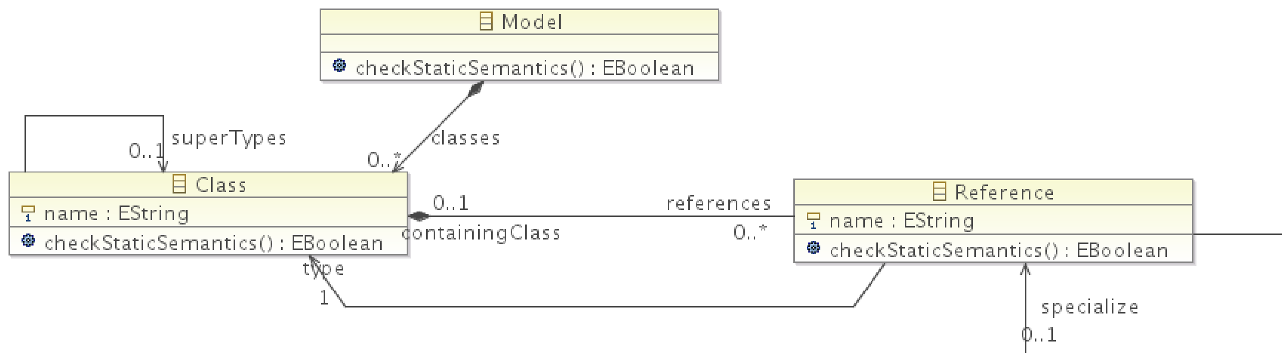
Exercise: Implementation of a code generator

Author: Martin Monperrus
Creative Commons Attribution License

Consider the following object oriented metamodel, named Oosp:



It's a subset of the Java and Ecore metamodel. However, compared to both, references can be specialized. The static semantics of specialization is as follows:

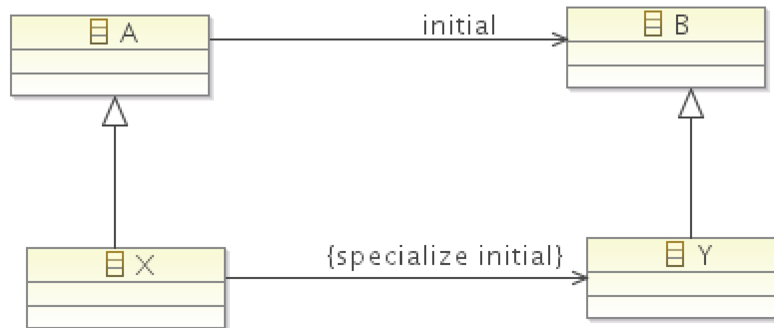
1. the type of a specializing reference must be a subtype of the type of the specialized reference.
2. the type of the containing class of a specializing reference must be a subtype of the containing class of the specialized reference.
3. the name of a specializing reference must be unsettable

The dynamic semantics is as follows. All specialized references are only accessible via setters. The setter of a specializing reference must ensure that the static semantics is preserved.

Technical details: **you should use the default "Eclipse Modeling Tools" version of Eclipse** (cf. http://www.eclipse.org/downloads/). Please read carefully the documentation of EMF, available in the Eclipse help system and, if you use one template technology, its documentation.

Tasks:

1. Create Oosp.ecore, the specification of the OOsp metamodel.
2. Create myModel.xmi that specifies the following OO classes as an instance of the Oosp metamodel (Open Oosp.Ecore >> Right click on "Model" >> Create Dynamic Instance, cf. EMF documentation).



3. Create oosp.genmodel (File >> New >> EMF Generator model, cf. EMF documentation) and generate the model code with the default generation options.
4. Implement static semantics in the method stubs of checkStaticSemantics.
5. Create a client class that loads myModel.xmi and check its static semantics (cf. documentation and lecture slides).
6. Create one model that conforms to the metamodel but violates the static semantics.
7. Implement the code generator on top of the generated model code. Choose one of the patterns presented in the lecture and explain the reasons of your choice. For example, the generated code of the example model above will look like:

```
// this code is generated
class A {
     B initial;
     void  setInitial(B  b){[write  the  code  generator  to
generate this part]}
}

class B  {}

class X extends A {
     void setInitial(B b) {idem}
     void setInitial(Y y) {idem}
}
class Y extends B{}
```

8. Refresh your project to compile the generated classes.
9. Write a test case that checks the generated code against the semantics of specialization.