4. Exercises on Concept Modeling

Objectives

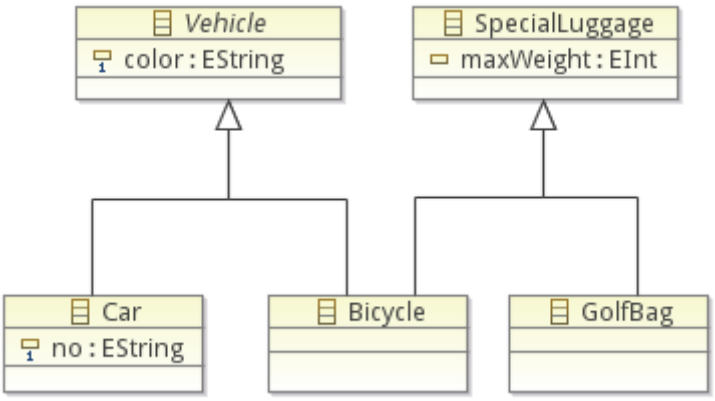
- To practice basic modeling tools of EMF
- To develop precise understanding of core class diagramming notation
- To use conceptual modeling as knowledge representation mechanism

The first two exercises are preparatory. The last task is the main objective of this week's exercise session. This task starts a mini-project that continues through several exercises ahead.

Always read the entire task description before starting to work on it.

I estimate that you have about 2 hours in class to complete this task + about 8 hours of self study time at home.

Task 1. [max. 15 minutes] Recall the inheritance hierarchy of vehicles and luggage from the lecture (recall that you have seen the logical interpretation of class diagrams in the lecture note for this week):



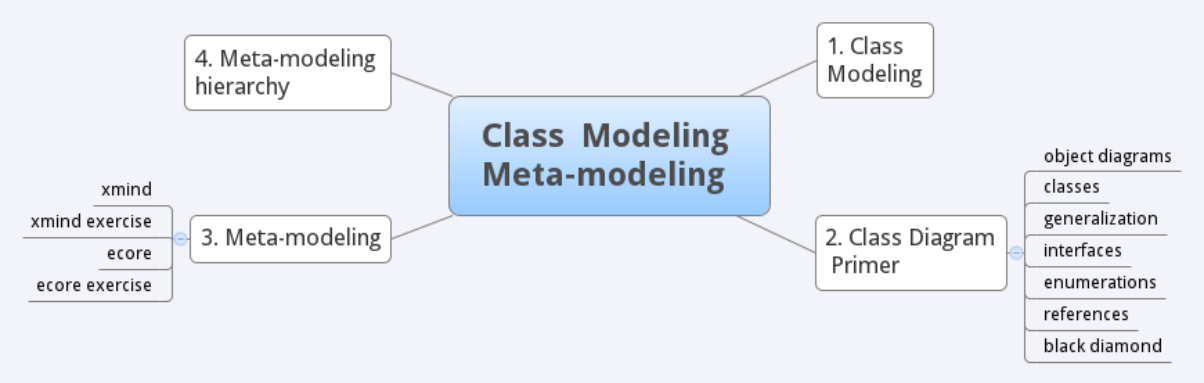
Which of the following first order sentences hold?

a) $\forall x.\,\mathsf{Bicycle}(x) \rightarrow \mathsf{Vehicle}(x) \wedge \mathsf{SpecialLuggage}(x)$?

b) $\forall x.\,\mathsf{Car}(x) \rightarrow \mathsf{SpecialLuggage}(x)$?

c) $\forall x.\mathsf{Car}(x) \rightarrow \mathsf{Vehicle}(x) \wedge \mathsf{SpecialLuggage}(x)$?

Task 2. [max. 15 minutes] Create an empty EMF project. Load the mind-map meta-model into the project. Open the ecore file and generate a Dynamic Instance of "Document Root", as you did in the first exercise session, in the tutorial. Now, please try to create an instance representing the abstract syntax of this, or similar diagram:



For the purpose of the exercise let's agree how the concrete syntax maps to abstract syntax. Topics are represented by blue boxes in the concrete syntax. Threads are represented by white boxes with a little blue circle. Thread items are represented by lines branching out of thread's blue circle.

The ecore file containing the mind-map meta-model is available in the `episode-10/` folder of the course git repository.

Task 3.* (time consuming) In this exercise we want to use class modeling as a method for system comprehension. Recall that improving domain understanding has been pointed out by practitioners as one of the main advantages of using models to begin with. Let's see whether class modeling can help you to understand a moderately complex system.

We will use the implementation of JUnit 4 framework as a case study. I assume that you are familiar with unit testing using JUnit, which will make the exercises easier.

3.1 [1 hour] Start with reading the user oriented documentation of JUnit: http://junit.sourceforge.net/doc/cookbook/cookbook.htm or https://github.com/junit-team/junit/wiki, but do ignore the javadoc for the time being.

High level documentation will primarily give you some class names and (few) relations between them. They do not necessarily correspond to low level implementation classes.

Identify key concepts, objects, subsystems and record them as classes, associations, generalizations, and aggregations. For example when you find the concept of Test, create the corresponding class. Then you encounter a concept of a Suite that aggregates multiple tests. You can create a Suite class, and make it own one or more tests using composition (black diamond).

Continue like that. Be precise to record cardinalities. If you, at any point, encounter constraints, dependencies between concepts, which cannot be expressed using class diagrams, then note them down in English, either in a separate file, or in an annotation. They will be input for our exercise next week.

All modeling should be done using a modeling tool (not on paper, not using a drawing tool). We want you to become fluent with tools.

My solution for this model has 12 classes. So if you have significantly more, you are either off track, or trying to do much more than I expect.

3.2 [1-2 hours?] The next step is to do a cursory pass over developer oriented documentation to refine your model. Developer documentation for Junit is essentially only javadoc, available at: http://junit.org/javadoc/latest/. Start with places that seem to be already connected to elements in your model. When you study it, refine the model continuously. By refinement we do not mean converting your model to an implementation level model, but just adding further abstract concepts and relations to your model.

You should not grow your model too much. You should focus on understanding whether the selection of classes, associations and generalizations is correct (so whether the lower level documentation confirms your initial sketch from the previous point). Also try to understand and record any constraints (including cardinalities) that you might have spotted.

The objective is not to create a diagram of implementation classes. So there does not have to be (and should not be!) a one-to-one mapping between your model and implementation classes. We only look at lower level artifacts to understand details of the system that were too hard to understand from user documentation.

3.3 [2-3 hours?][1] Finally, you need to delve into the code, and reading JUnit code should be (only) relatively easier at this point (after the first two steps). JUnit is a small and well implemented framework, done by some of the best programmers in the universe. Bear it mind that it will be by orders of magnitude a better experience to read than any alien code you might read in your job.

In order to get code, it is probably easiest to [2]

```
git clone https://github.com/junit-team/junit.git
```

In my Eclipse, I needed to switch the Java compiler to 1.6 in order to make JUnit compile with no errors. It is easier if you work with a stable release, than with a snapshot code. In Eclipse you can switch to another branch in the project's context menu, under Team, switch to. Stable releases are under tags.

It is good for the project to be set up, so that you can compile it. Then you can effectively use Eclipse searching, navigation support, tooltips, etc, to orientate yourself much faster in

[1] The exercise can take quite a bit more time, if never used git, never used JUnit, and never used Eclipse. If you lack two or more of these, then try to team up with someone who has more experience than you.

[2] See also http://www.vogella.com/articles/EGit/article.html

the implementation. You can get a sanity check, if the build environment is reasonable by running JUnit's own unit tests.

While studying code you should record new information you learn in the class model, and in your list of constraints. Again it is not our point to reflect implementation classes one-to-one in your high-level model, but just add information, or correct what was misunderstood.

Task size guide: When I was solving this task, my final model had 28 classes. So I added 16 during steps 2 and 3, mostly specializations of classes already created in step 1. But I occasionally needed to refactor/reorganize something, which I did not understand correctly in step 1.

By now you should have a class model, which ... **we are going to throw away!!!** Yes, throw-away-modeling is an established practice. There is no point to maintain this model[3]. The main point of the model was to facilitate your learning of how JUnit is implemented. This ability will be useful later in the course, when we will try to change JUnit.

Hand-in: A screen shot of your **final** JUnit domain model (before you throw it away ...).

[3] do keep it for next exercises, though