Model Driven Development

Lecture Notes

Andrzej Wąsowski

IT University of Copenhagen. Fall 2012

Ingredients

1 Introduction **5**
- 1.1 Why Modeling? . . . 5
- 1.2 Model Driven Development . . . 6
- 1.3 Why Model Driven Development? . . . 7
- 1.4 MDD Practice . . . 8
- 1.5 Course Overview . . . 10

2 Class Modeling & Meta-modeling **12**
- 2.1 Class Modeling . . . 12
- 2.2 Class Diagrams Primer . . . 12
- 2.3 Meta-Modeling . . . 15
- 2.4 Meta-Modeling Hierarchy . . . 16

3 Exercises **21**

4 Object Constraint Language **24**
- 4.1 Introduction . . . 24
- 4.2 OCL Syntax and Semantics by Example . . . 25
- 4.3 Crash Summary of OCL . . . 27
- 4.4 Playing with EMF and OCL Practically . . . 28

5 Project Ideas **31**
- 5.1 Preliminaries . . . 31
- 5.2 Project Ideas . . . 31

6 Exercises **35**

7 Domain Specific Languages **39**
- 7.1 DSL Implementation Strategies . . . 39
- 7.2 Internal (Embedded) DSLs . . . 40
- 7.3 Sinatra: an Example from Ruby World . . . 41
- 7.4 Towards External DSLs. Metamodeling . . . 43
- 7.5 Some DSL Design Principles . . . 44

8 Writing a Project Proposal **45**
- 8.1 Why ? . . . 45
- 8.2 Problem Definition . . . 45
- 8.3 Method . . . 47
- 8.4 A Checklist . . . 47
- 8.5 When? . . . 48

8.6 Administrivia . . . 48

9 Evidence in a SE Project **49**
9.1 Introduction . . . 49
9.2 Matching Evidence to Claims . . . 50
9.3 Scope of Evidence . . . 53
9.4 Assorted Advice . . . 54

10 Exercises **55**

11 Textual Concrete Syntax, XText **57**
11.1 Introduction . . . 57
11.2 External DSLs with Xtext (Demo) . . . 57
11.3 Concrete Syntax Definitions in Xtext . . . 61
11.4 DSL Design Guidelines (Concrete Syntax) . . . 64

12 Exercises **66**

13 Model-To-Text TX (M2T) **69**
13.1 Model Transformations: Applications & Classification . . . 69
13.2 Model-To-Text Transformations (M2T) . . . 70
13.3 Gluing Things with MWE2 . . . 71

14 Model-To-Model TX (M2M) **76**
14.1 Implementing M2M Tx in Java . . . 76
14.2 Xtend at a Glance . . . 77
14.3 Implementing M2M Tx in Xtend . . . 79
14.4 Odds and Ends . . . 82

15 Exercises (Model Transformations) **84**

16 Another Take on M2M TX **86**
16.1 A Glimpse at QVT Relations . . . 86
16.2 Rule-based M2M Transformation with ATL . . . 87
16.3 Example Exam Questions . . . 89

17 Software Product Lines **91**
17.1 Software Product Lines . . . 91
17.2 Domain Modeling Spectrum . . . 92
17.3 Domain Modeling with Feature Models . . . 93
17.4 Domain Modeling with DSLs . . . 94
17.5 Domain Implementation . . . 95
17.6 Other Advice on Realization . . . 96

18 Reading and Writing Research Papers **98**
18.1 Introduction . . . 98
18.2 Scoping . . . 99
18.3 Telling A Story . . . 99
18.4 Abstract . . . 100
18.5 Introduction . . . 101

18.6 Evaluation . . . 101
18.7 Threats to Validity . . . 102
18.7.1 Internal Validity . . . 102
18.7.2 Examples of Threats to Internal Validity . . . 103
18.7.3 Threats to External Validity . . . 103
18.7.4 Examples of Threats to External Validity . . . 104
18.8 Related Work . . . 104
18.8.1 How to Find Related Papers . . . 104
18.8.2 How To Select Papers To Read . . . 105
18.8.3 Reading . . . 105
18.8.4 Citing . . . 106
18.8.5 Writing The Related Work . . . 106
18.9 Conclusion . . . 106
18.10 Title and Author List . . . 107
18.11 Format . . . 107
18.12 Homework due latest on November 7th . . . 107
18.13 Homework due latest on November 14th . . . 108
18.14 Later This Semester . . . 108

19 Theses & Projects in MDE **109**

20 Exam Pensum & Format **110**
20.1 Pensum (Exam Reading) . . . 110
20.2 Exam Format . . . 110
20.3 Example Exam Questions . . . 110
20.4 Example Assessment of The Report . . . 113

21 Writing, Reviewing **114**
21.1 Writing . . . 114
21.2 Reviewing . . . 115

1 Introduction

Reading: This is a project course. It does not have a fixed reading list. The main reading material are lecture notes and slides, along with documents referenced from them. The important references are marked "*please read*". Besides these I reference additional papers, that you may want to read depending on your interests, or relevance to your project. You will expand the reading list with papers relevant for your report. **The exam pensum is only the lecture notes and literature relevant for your report.** Also note that the project report, and the literature referenced therein (within the limit of relevance) is part of the exam syllabus.

Please read [21], 20 minutes in total.

You may want to read [8], if you are interested in how MDD is used in practice today. A good overview of what MDD offers to software developers can be found in the first two chapters of [24].

Protected articles can be downloaded from ITU's premises, through ITU's license.

1.1 Why Modeling?

Using models to design complex systems is commonly applied in traditional engineering, be it architecture (buildings), civil engineering (roads and bridges), avionics and automotive (aircraft and car).

Engineers build a variety of models to assess various properties. They build different models to design the chasis of a car, and other ones for designing its electrical system. Different models are used in production, and yet different are used in servicing systems. Clearly, models are ubiquitous in engineering. Some examples of models include:

- Computer visualizations and paper and foam mock ups of building designs.
- Mathematical models of robustness of constructions, air flow, fuel consumption.
- Economical models to assess design and production cost.

In many ways software systems are just as complex (or even more complex) than other achievements of engineering. A typical commercial software system has more lines of code than the mechanical parts of Boeing 747. Jumbo Jet has only 6 milion parts, half of them being super simple fasteners, many of them identical.[1]. Today Linux kernel has about 15 mio. lines of code.[2] Open Office has about 9 mio. lines of code.[3] Microsoft Windows is reported to have exceeded 50 million lines in 2003.

[1] http://www.boeing.com/commercial/747family/pf/pf_facts.html retrieved 2012/08/28
[2] http://www.h-online.com/open/features/What-s-new-in-Linux-3-2-1400680.html?page=3, retrieved 2012/08/28
[3] http://www.openoffice.org/FAQs/build_faq.html, retrieved 2012/08/28

*Models help us **understand** a complex problem and its potential solutions through abstraction. Therefore it seems obvious that software systems, which are often among the most complex engineering systems, can benefit greatly from using models and modeling techniques. [21]*

In computing the use of models is growing, but clearly software engineering is an immature discipline in this respect.

The last great steps in abstraction was introduction of compilers for high level languages in the 50s (move from assembly programming to high level programming) and introduction of relational databases in the 70s (nowadays almost nobody builds a database system without constructing, even an implicit, relational data model).

Note that introduction of object oriented programming was a very small advance in comparison to the two — the abstractions of object oriented programming are much closer to the abstractions of earlier programming languages than to the concepts of systems we build. A Java loop is like a Fortran loop (Selic).

So what is a model?

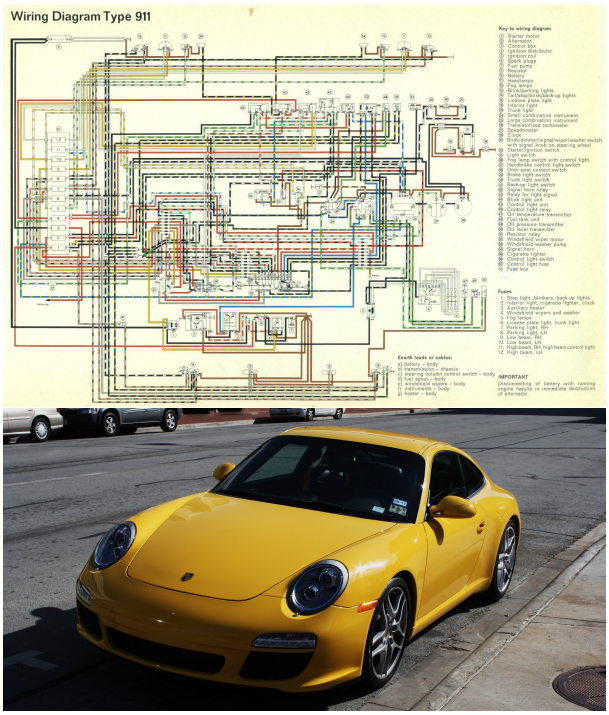


Figure 1.1: The electrical scheme of Porsche Carrera 911 (above) and the actual system (below)

⚠ **Definition 1.** A model is an abstraction of reality, of a system, made with a given purpose in mind.

A model does not contain all information, but it preserves the information necessary to perform the intended application for the model. *All models are wrong, but some are useful* [George Box, a recognized statistician]

1.2 Model Driven Development

Software development is particularly well suited for use of models. Note that when building a car, there is a huge abstraction jump between a computer model and a real physical construction. In computing both models and systems are virtual objects, so this jump is incomparably smaller. It is possible to refine models into system in a continuous manner; with much less effort than when designing cars or buildings. This is why for software it is possible to make modeling the central paradigm in development — this is the main idea of Model Driven Development:

> *Model-driven software development is a software development methodology which focuses on creating and exploiting domain models (that is, abstract representations of the knowledge and activities that govern a particular application domain).*
>
> [Wikipedia, MDE, 2011/08/27]

MDD's defining characteristics is that software developments primary focus and products are models rather than computer programs. [...] A key characteristic of these methods is their fundamental reliance on automation and the benefits that it brings.

[21].

Remark: MDD (Model-Driven Development), MDE (Model-Driven Engineering), MDSD (Model-Driven Software Development) and MDA (Model-Driven Architecture) are very close terms. MDE emphasizes broader use of models (not only for development). MDSD is simply more precise than MDD, and MDA is a trademarked version of MDSD developed by OMG and based on UML.

1.3 Why Model Driven Development?

⚠ What can we gain by using modeling in software development ?

- Ability to express concepts that are closer to the problem domain, than to the implementation technology. So we can stop talking about classes and loops, but talk about business entities, cash flow processes, and customers.
- Software defined in domain terms is easier to maintain (models are documentation). In extreme cases can be adapted by domain experts, or more technical users, due to *Domain Specific Languages*. A good example here are customizable enterprise systems which are implemented by highly skilled low level engineers, but customized by business domain consultants. Also many computer games allow end-user extensions through various mod packages implemented as domain specific programs (models).[4]
- Software (or large self-contained parts of it) can be automatically generated from models. This is well visible in the Eclipse Modeling Project, which is able to generate implementations of models, (de)serializers of models, instance generators for languages, tests and editors for models, well-formedness checkers for models.
- System can be simulated. Simulation of models is one of the most powerful technique used by hardware engineers, and by engineers of embedded systems. Construction of an executable model allows simulating the behaviour of the system before the system is actually constructed. This is a powerful technique for finding many obvious early design mistakes.

  Most certainly, you have experienced multiple times that after implementing of a program, even a small one, it usually does not run as expected. So a few first runs of a procedure are used to uncover obvious mistakes. Now if you had a model of the program in a language with a simulator, you would be able to uncover the mistakes much earlier.

  For static models (for example data models), generation of instances plays similar role to simulation for behavioral models. For instance generation of instances for a database schema model can often show problems in the integrity constraints of the schema.
- System models can be verified (for example for safety properties). Since we use code generation we are highly confident that properties of the model are also properties of the final system.

[4] In the context of DSLs, it is essentially equivalent to talk about models and programs. Anekke Kleppe uses the terms *mograms* to emphasize this [16].

- System monitoring and debugging: models can be linked to a running system implementation, program state can be visualised as models, or problematic data can be visualized as models. This is often done by embedded system's developers connecting Simulink models to running hardware.

  Model can be used to monitor a system, and flag errors if the actual execution diverges from the specification.

- If systems rely primarily on models, then data exchange and integration of systems can be done via models. This is particularly convenient, since models can be translated to models in other languages by *model transformations*, which are small programs implemented in DSLs specialized for model transformation; much easier to implement than regular data migration code.

1.4 MDD Practice

Hutchinson and coauthors [8] attempt to provide solid scientific evidence that MDD is commercially beneficial (or not). To understand this, they have asked 250 individuals in diverse organizations to respond to a survey on this topic, and subsequently they interviewed in depth 22 MDD professionals from 17 different companies.

Table 1.1 shows their summary of positive and negative influences of MDD on the development process. The Table shows a more balanced view on MDD — indeed, there is no clear picture whether MDD would turn beneficial or not.

Table 1.2 quantifies the above influences by looking, which MDD application areas lead to increase in productivity and maintainability and to what extend. High number in the "Increased" column means that most respondents apply the practice, and that they find it beneficial (it increases productivity or maintainability). Observations:

- Clearly all practices (but tesitng) are beneficial if they are used. For testing the beenfit is less clear cut.
- Used of models for communication, design, documenting, and code generation are most wide spread.

Other findings of the survery include:

- 83% of respondents think that MDE is a good thing and 5
- The majority of respondents considered the use of MDE on their projects to be beneficial in terms of personal and team productivity, maintainability and portability (58-66%). However a significant number disagreed (17-22%). This suggests that there is some challenge in implementing MDE successfully.
- MDE users employ multiple modeling languages. 40% employ DSLs
- Huge productivity gains are quoted in the qualitative study (at least 2-fold, up to 8-fold), which are sometimes hidden or downplayed
- 50+ tools are used by the respondents, suggesting a lack of maturity—definitive market leaders are yet to emerge. Tools are immature, complaints about prices are common.

Remark: This empirical study is a good example of assessment in software engineering. When you build a tool, that you claim is a technical improvement, often such study is a suitable assessment method.

Table 1.1: Illustrative influences of MDE as presented in [8].

Impact Factor	**Illustrative Examples of MDE Influences**	
	Positive Influences	**Negative Influences**
Productivity		
• *Time to develop code*	*Reduced* by: automatic code generation.	*Increased* by: time to develop computer-readable models; implement model transformations, etc..
• *Time to test code*	*Reduced* by: fewer silly mistakes in generated code; model-based testing methods, etc.	*Increased* by: effort needed to test model transformations and validate models, etc.
• *ROI on modeling effort*	*Positive* influences of modeling*:* more creative solutions; developers see the "bigger picture".	*Negative* influences of modeling*:* "model paralysis"; distracting influence of models.
Portability		
• *Time to migrate to a new platform*	*Reduced* by: simply applying a new set of transformations.	*Increased* by: effort required to develop new transformations or customize existing ones.
Maintenance		
• *Time for stakeholders to understand each other*	*Reduced* since: easier for new staff to understand existing systems; code is "self-documenting".	*Increased* since: generated code may be difficult to understand.
• *Time needed to maintain software*	*Reduced* since: maintenance done at the modeling level; traceability links automatically generated.	*Increased* since: need to keep models/code in sync, etc.

Table 1.2: The impact of MDE activities on productivity and maintainability according to the study presented in [8].

Activity	Productivity		Maintainability	
	Increased	Not Used	Increased	Not Used
Use of models for team communication	73.7%	7.0%	66.7%	6.7%
Use of models for understanding a problem at an abstract level	73.4%	4.8%	72.2%	6.1%
Use of models to capture and document designs	65.0%	9.3%	59.9%	10.7%
Use of domain-specific languages (DSLs)	47.5%	32.6%	44.0%	33.7%
Use of model-to-model transformations	50.8%	24.6%	42.6%	28.4%
Use of models in testing	37.8%	33.9%	35.2%	32.4%
Code generation	67.8%	12.0%	56.9%	12.6%
Model simulation/ Executable models	41.7%	38.3%	39.4%	35.9%

1.5 Course Overview

In this course we focus on MDD principles. We want to learn some of the basic techniques, but we also want to learn how to extend the techniques, how to critically evaluate them.

This course is not primarily about processes. It is about technology, and architecture. The conceptual underpinnings of MDD are quite simple, but the technical space is huge and inhomogeneous these days.

We will cover the following subjects approximately:

Core MDD Content in Lectures

- Introduction, motivation, overview
- Metamodeling and Domain Specific Languages
- Object Constraint Language
- Support for Concrete Syntax (Xtext)
- Model transformation (Xpand)
- Component based development (with OSGi)
- Multimodeling and model management
- Product Line Architecture and Variability Modeling

To support the projects we will also cover a number of methodological subjects. We will meet for the lectures on the following subjects:

Methodological Content

- Defining a research problem
- Evaluation: evidence for your claims, quality of solutions
- How to read a research paper? How to study related work?
- Methodological correctness; threats to validity
- Structure and content of a research paper; writing advise
- Possible future thesis work in this area
- How to write a research paper?
- How to review a research paper?

These are meant to support your thesis writing.

The lectures are meant as means of efficient supervision. If you believe that you are well familiar with the topic of a given lecture, it is definitely better to skip it, and work on the project in the time you save instead.

Exam. The exam is oral, and takes a starting point in your project. You present your project briefly and then we discuss. The discussion very quickly converges on exploring concepts explained in the lectures. So the lectures *are* part of the pensum. We can ask about anything in the lectures, but are most likely to ask about topics that you used, or should have used, in your project. For your benefit, I mark some places in the notes that are particularly easy to derive exam questions from (look for the ⚠ sign).

We are going to have an episode devoted to the exam. Then I will present example questions, and address all your worries.

Details of syntax of various technologies we discuss are not part of the syntax, but some rough idea of how to use them is needed, including ability to write examples on a whiteboard (perhaps using approximate syntax). Ask if in doubt, if you should study something or not.

2 Class Modeling & Meta-modeling

Reading: This talk is devoted mainly to a quick refresh of class modeling. Please use any sources you are aware of to read up about this.

The basic introduction to EMF is available in [3].

EMF tutorial for the current release of Eclipse can be found at http://www.vogella.de/articles/EclipseEMF/article.html

It is worth trying to look at the MOF specification from OMG. It is available at http://www.omg.org/mof/; follow the link at the bottom of the page to the current spec. Get an overview, and read enough fragments, as to get an impression of what is entailed by a standardisation of a modeling language.

Otherwise I recommend poking online (wikipedia, omg, emf websites) to get a brief understanding of what is EMF, ecore, MOF, XMI and relations between these concepts.

Meta-modeling hierarchy is described in section 6.2 of [24]. This hierarchy can be found also in *UML 2.0 Infrastructure Specification* from OMG, however that spec is not for those of faint heart.

2.1 Class Modeling

Unified Modeling Language (UML) provides a complete set of notations for modeling software systems. This includes requirements, architecture, types, structure and behaviours.

In this course we use only (or primarily) UML *class diagrams* and only for structural modeling. Our view of UML is thus clearly restricted. I also mix-in some non-standard EMF extensions.

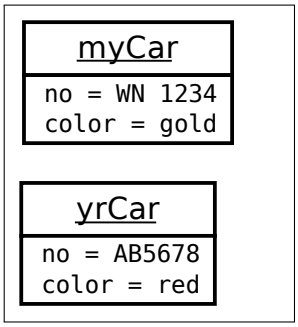
⚠ **Definition 2.** A *class* is an abstraction that specifyies attributes of a set of concept instances (objects) and their relations to other concepts (objects).

Observe that in the above we do not mean that a class is a programming language concept, existing in an implementation in a programming language. In this course we first and foremost will be using classes to model real-world concepts, or more precisely *domain-level* concepts. These are often much more abstract than what classes are, for example, in Java or C#. This big leap from implementation to a conceptual application domain is necessary in order to obtain productivity gains promised by MDD.

⚠ 2.2 Class Diagrams Primer

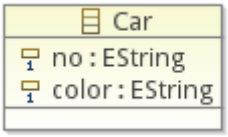
Both classes and objects are depicted by boxes with 3 compartments: names, attributes, and operations (we ignore the operations in the talk, but they are useful in some systems). When visualizing models, attributes and operations can be omitted if not essential.

The object diagram to the right shows us that there exist two cars with some attributes.



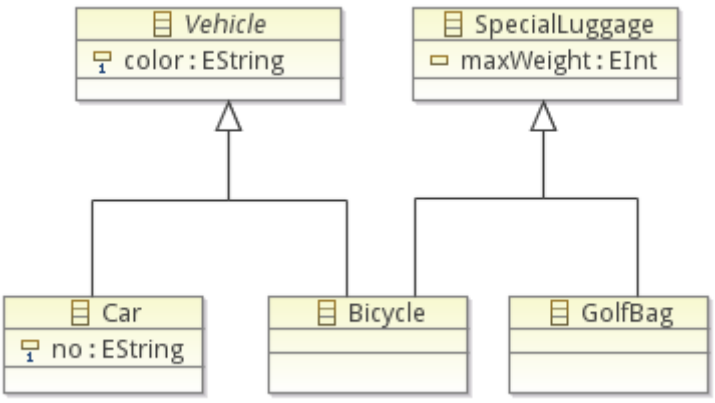
In the figure each object has two attributes. Object names (instance names) are underlined to distinguish them from classes.

In the simplest view classes are just types for objects. Our objects are of type Car:



The above block represents class Car. The name is not underlined, and attributes have types, not values. They also have a *multiplicity* constraint (here simply "1"), which says that both attributes are mandatory for any instance of this class. By convention class names are capitalized, and instance names are not.

Generalization relation (also known as inheritance relation) says that instance sets of two classes are included: if class A generalizes class B, then each instance of B is also an instance of A.



Every Car is a *Vehicle*, and so is every Bicycle. Similarly we have two kinds of special luggage. We sometimes say that a generalization relation expresses the *kind-of* relation ship ("a car is a kind of vehicle").

The above diagram shows an example of abstract class (*Vehicle*) – so class with no instances of its own (cursive title). We also show multiple inheritance: a bicycle is both a vehicle and a special luggage.

All the diagrams presented above are actually ecore class diagrams (not UML class diagrams). Ecore is an implementation of MOF made within the Eclipse project, and MOF (which stands for Meta-Object Facility) is a simple class diagramming language standardized by OMG, and used to define abstract syntax of UML languages, including the full blown UML class diagrams.

All the class diagrams above (and below) has been made using tools of the Eclipse Modeling language, and exported as models. In particular it means that EMF can process them automatically.

Already for such simple diagrams, we can generate editors of instances, or interactively derive instances and serialize them to xmi files (XMI is an OMG standard for model serialization).

NB. EMF also allows modeling of interfaces — this is done by adding an interface property to a class. The ecore diagram editor puts a little interface icon, next to the class name:

When EMF generates code interfaces are mapped to Java interfaces, while classes are mapped both to classes and interfaces. The latter is a simple pattern (a workaround, if you prefer) for Java's lack of multiple inheritance.

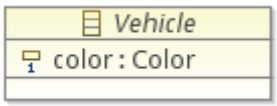
EMF provides simple types (for example **EString** used above), which are mapped to Java types during code generation.

In class diagrams an attribute declaration can be followed by default value:

```
color :EString = ‘‘red’’
```

Enumerations often come useful in modeling, when we have a finite number of discrete simple values:

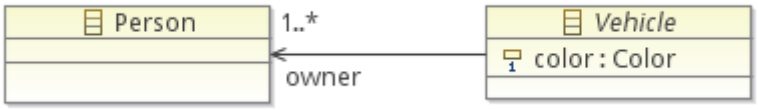
Enumerations can be used as types for attributes:



Now we only allow one of the four colors for vehicles.

It is also possible to introduce new basic types, by providing their Java implementations. Details in [3] (search for **EDataType** in the index).

An *association* represents a relation between instances of two classes. Note that association is qualitatively different than inheritance. We use associations to model all other kinds of relations between objects than *kind-of*.



In EMF associations are always binary references, so they have a direction. In UML references can be bidirectional (EMF requires simulating it with pairs of references) and n-ary. For our purpose of metamodeling, binary references are typically sufficient.

The navigable name of the association is written on the "far end" — so `myCar.owner` gives the object representing the owner of `myCar`.

In the example the reference is also decorated with a multiplicity constraint `1..*`, meaning that a vehicle must have at least one owner. More than one owner are allowed (for modeling co-ownership). This also means that technically `myCar.owner` returns a collection, and not a single instance.

Finally references can be used to denote a *part-of* relation (in constrast to the *kind-of* relation of generalization). This is denoted using a black diamond on the owner side:

In this example we state that each vehicle contains 4 wheels as its integral part. This means that a vehicle instance without 4 wheels cannot exist (such an instance is not well-formed).

(**Q. I**ncidentally, the example is silly, as it also means that every bike has 4 wheels. Do you know why?)

The black-diamond semantics also means that every object in such a relation can only have one owner — so there could not be cars sharing wheels. It also imposes a syntactic well-formedness rule: objects cannot be owners of themselves—efectively, directed subgraphs of any class diagram consisting of reference edges labelled with black diamonds have to be trees (or forests).

The black-diamond associations are interchangable called "compositions", "aggregations", and "part-of relations".

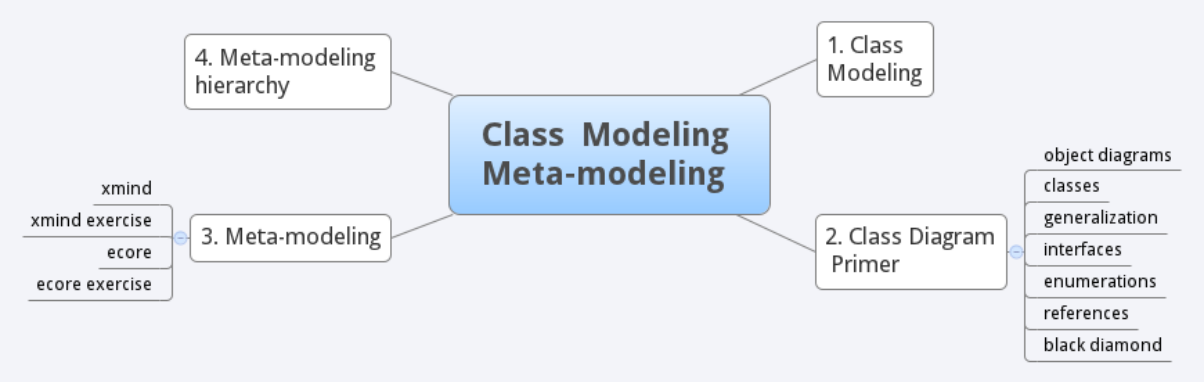
2.3 Meta-Modeling

In order to be able to process languages automatically we need to provide formal definitions of them. The most popular tools (EMF, etc) allow defining modeling languages (or at least defining their abstract syntax) using class diagrams, or very similar meta-modeling languages (like KM3). Then they can generate parsing and persistence code that treats models (mograms) in such defined languages as object diagrams.

⚠ **Definition 3.** *Meta-modeling* is the practice of modeling syntax of other languages using class diagrams. So meta-modeling is modeling of modeling languages, and a meta-model is a model of a modeling-language.

Figure 2.1 shows a meta-model of one of the classic examples—a mind-mapping language.

Small exercise: *Take a piece of paper and draw an instance diagram representing the following mindmap:*



Now that we know how to describe abstract syntax of languages using class diagrams, we can describe class diagrams themselves.

2.4 Meta-Modeling Hierarchy

⚠ It turns out that one can (retro-actively) provide an ecore model representing . . . ecore syntax (and MOF model representing MOF syntax, and a UML model representing UML syntax). Figure 2.2 shows this meta-model.

⚠ This has actually sometimes led to confusion that some languages are defined in themselves, for example 'UML defined in UML'. This is not true — circular definitions of languages are not possible.

The practice of modeling the language in itself could better be called *bootstrapping*. Indeed, it is really akin to the practice of programming language designers, who tend to implement compilers for a new language in the language itself, as the first serious maturity test. For example your favourite java compiler is most likely implemented in java.

Of course, a bootstrapped language first needs a compiler or an interpreter implemented in another language (which already has a compiler or interpreter). Similarly for modeling languages: the first definition uses an existing language, or simply natural language description. The bootstrap-like self-definition comes later.

Another exercise: *Try to take any of the tiny ecore models above and draw its abstract syntax as an instance of the ecore meta-model above. You need not to complete the exercise, but do enough, to understand the structure of the meta model. It makes sense to check out the simplified meta-model of*

ecore in chapter 2 of [3]. This model has only 4 classes, and makes it quite easy to understand the main idea behind representation of ecore in ecore.

Various levels of meta-modeling can be set at various abstraction levels. For example a meta-model of ecore is very abstract. A meta-model of UML expressed in ecore is more concrete. A model of video-rental application expressed in UML is even more concrete. An instance of that model, an actual data collection of videos is very concrete. These four levels of abstraction together constitute the
⚠ abstraction hierarchy for meta-modeling shown in Fig. 2.3.

In DSL based development, your DSL fits into level M2, replacing UML, and concrete models in the DSL are at level M1 — it depends on the concrete project whether they have further instances or not. Usually they do not.

Note that the hierarchy is a bit controversial in the sense that some (including perhaps me) would put the object diagram at M0. This issue you do not have with most DSLs, are instances of DSL models are rare. Typically a DSL model has a direct real world interpretation.

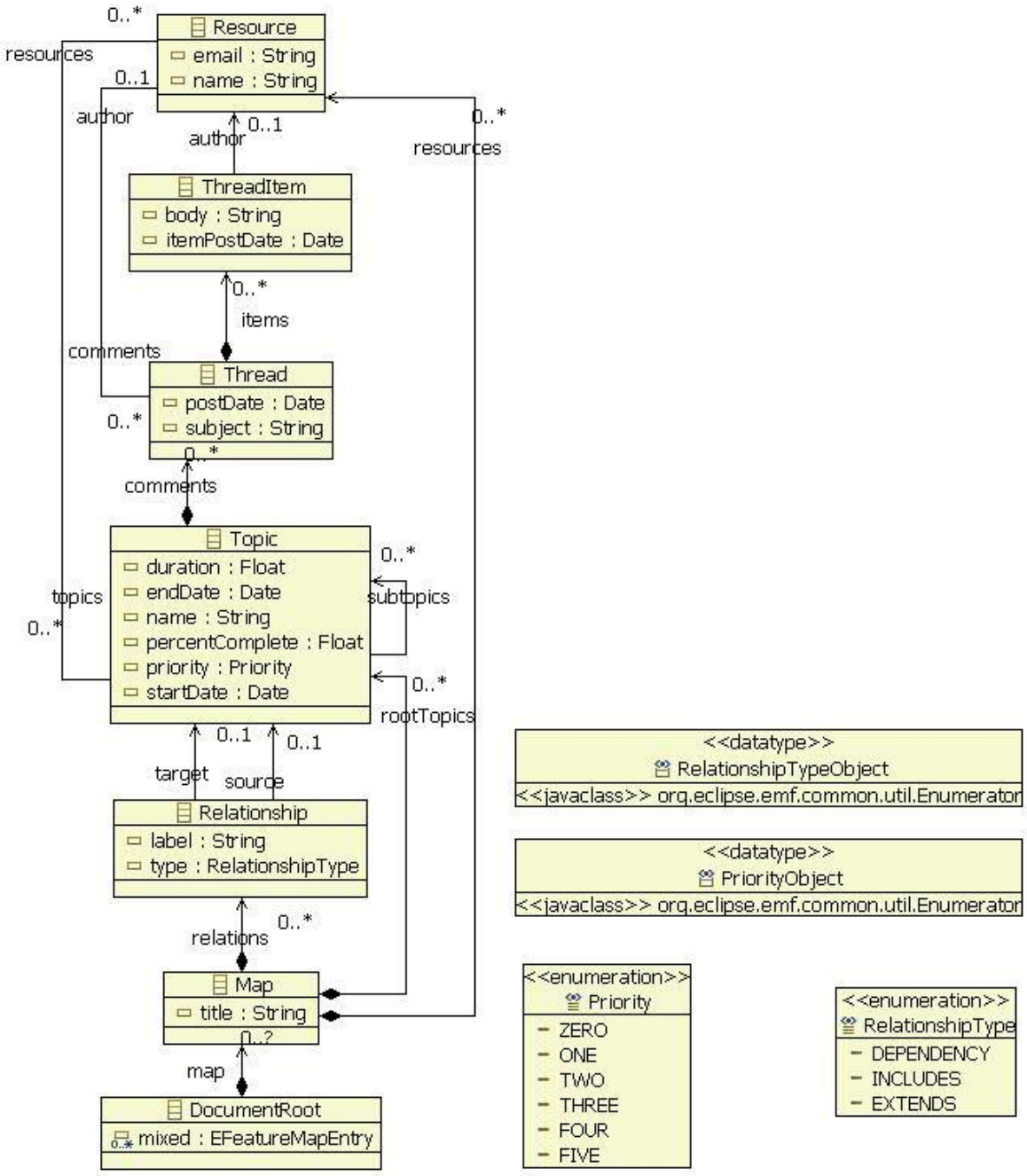


Figure 2.1: Meta-model of a mind-mapping language.

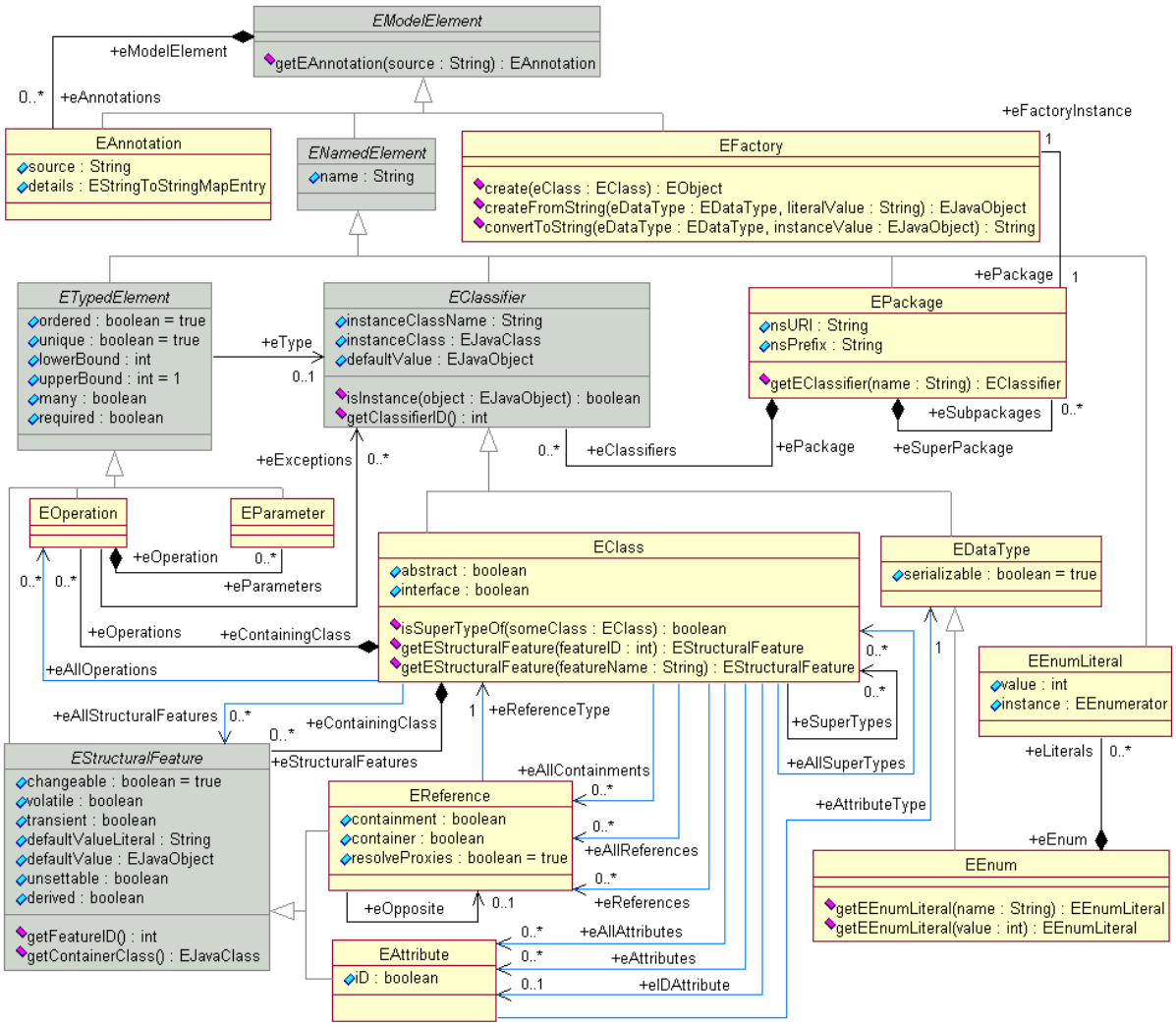


Figure 2.2: An ecore meta-model of ecore.

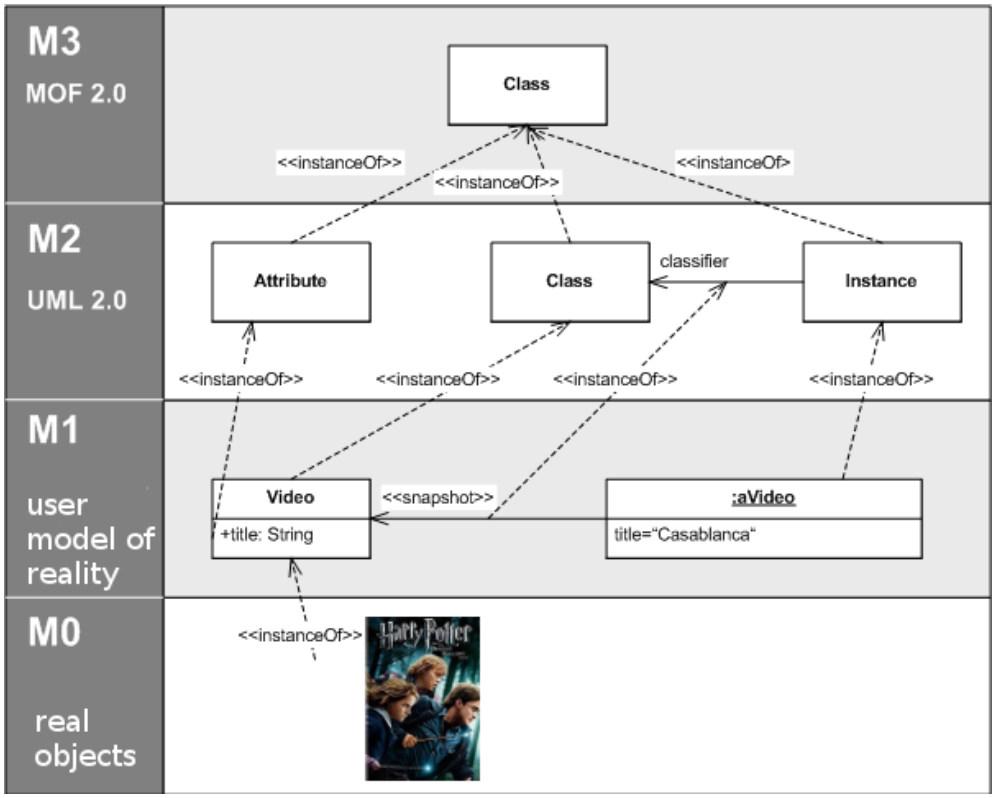


Figure 2.3: Metamodeling-hierarchy.

3 Exercises

Objectives

- To install the modeling environment (Eclipse Modeling Distribution)
- To learn basic modeling tools of EMF
- To operationally recall modeling using class diagrams
- To use conceptual modeling as knowledge representation mechanism

The first three exercises are preparatory and let you famillliarize yourself with tools. The last task (4) is the creative objective of this week's exercise session. We will continue with the last task next week.

I estimate that you have about 2 hours in class to complete this task + about 8-10 hours of self study time at home.

Task 1. Install Eclipse modeling distirbution following the tutorial on the course website. The tools should be available in the lab, but we strongly recommend using your own laptop for the task. If you already have Eclipse installed, we recommend making a clean install on the side. Several installations of Eclipse easily co-exist on the same PC.

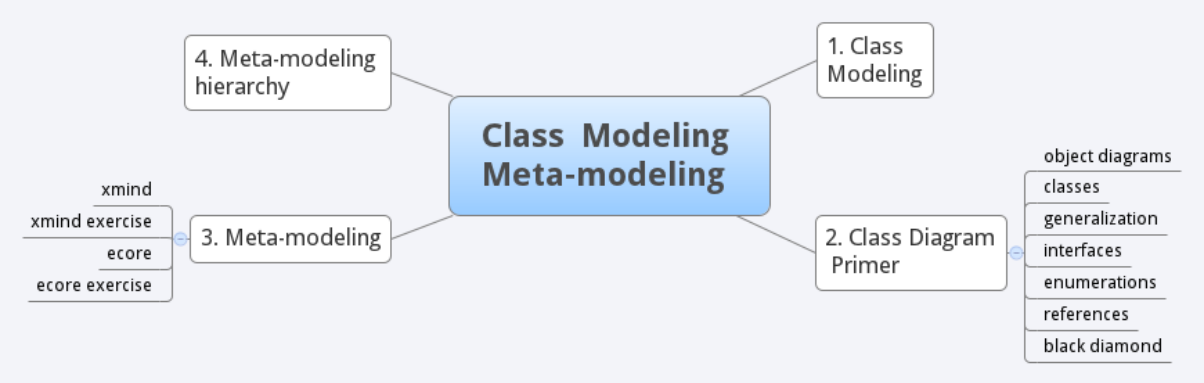
We recommend that you do this at home, before the exercise session.

Task 2. Create simple class diagrams from Episode 2 using the EMF tree editor, and EMF diagram editor. Get familiarized with both. The tree editor is much more robust and solid. Create at the very list a diagram containing multiple inheritance, associations, and containment.

A decent tutorial on using ecore is http://www.vogella.com/articles/EclipseEMF/article.html[1] In section 3.2 it uses the diagram editor, and shows it in detail. Section 3.3 shows the same diagram in a tree editor. There is not much explanation on how to use it, but it should be fairly intuitive.

Task 3. Load the mindmap meta-model into Eclipse and generate the editor plugins for it. Generate the model editor plugin for this meta-model and spawn the Eclipse instance that does this. This can be done in the same way as in sections 4 and 5 of the tutorial linked from the previous exercise. Now in the spawned Eclipse instance, you can edit instances of the mindmap. Please try to create an instance representing the abstract syntax of this, or similar diagram:

[1] It is not part of the exercise to follow this tutorial, but you are welcome, if you find this interesting.



For the purpose of the exercise let's agree how the concrete syntax maps to abstract syntax. Topics are represented by boxes in the concrete syntax. And only root topics are blue. Threads are represented by branches with a little blue circle and lines with labels over them. Thread items are represented by lines branching out of thread blue circles.

The ecore file containing the mind-map meta-model is uploaded to the class materials in learnIT.

Task 4. I assume that you know class modeling, so the above tasks were meant mostly to get you up to speed with tools, and refresh the notations. In this task we want to use class modeling as a method for system comprehension. We need a very simple case study system to do that. We will use the implementation of Junit 4 framework, since I assume that you are reasonably familiar with it, which will make the exercises easier. We will also use JUnit in later exercise sessions, and possibly also in some projects.

4.1 Start with reading the user oriented documentation of JUnit: http://junit.sourceforge.net/doc/cookbook/cookbook.htm. 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.

4.2 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://kentbeck.github.com/junit/javadoc/latest/. Start with places that seem to be already connect to elements in your model. When you study it, refine the model continuously.

Finally, you need to delve into the code, and reading JUnit code should be relatively easier at this point (after the first steps). It is a small and well implemented framework. By orders of magnitude a better experience than what you will get in your first proper programming job.

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

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

```
git clone https://github.com/KentBeck/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. 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 the implementation.

While studying code you should record new information you learn in the class model, and in your list of constraints.

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.

[3] do keep it for next exercises, though

4 Object Constraint Language

Reading: A standard reading on OCL is [25]. Chapter 3 has guidelines about writing constraints (see esp. Section 3.10 on tips and tricks).

Current OCL specification can be found at http://www.omg.org/spec/OCL/Current/. As usual do not use it for studying OCL. Learn the language from other materials, but then try to skim and read fragments of OCL spec, to get a feeling of its flavor. Chapter 7 (The OCL Language Description) is certainly worth looking into.

Jackson contrasts OCL with his specification language Alloy, which is more relational, than first-order in flavour. His book [10] and the journal paper on Alloy [9] contain short and interesting critiques of OCL. Short (few pages only, to be found in the final sections), but very useful if your project is about constraints.

4.1 Introduction

> The Object Constraint Language (OCL) [is] a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Note that when the OCL expressions are evaluated, they do not have side effects (i.e., their evaluation cannot alter the state of the corresponding executing system).[OCL specification]

⚠ We are going to describe OCL referring to the knowledge you already have about programming languages. :

OCL is a **declarative** programming language, with well defined syntax and semantics. OCL is **first order predicate logics** given a programmer friendly syntax (no **quantifiers**, or quantifiers are hidden as **collection iterators**).

OCL is declarative (no variables, no state, but no higher order - so this is not your new Haskell). OCL is **strongly typed** (like for instance Java).

Constraints express **invariants** over classes.

They can also be used to express **pre-** and **post- conditions** for operations, but we are not concerned with them in this lecture.

OCL can also be used to specify body of operations independently of programming language; again we are not concerned with that here; and for defining and deriving new associations and attributes, initial values. Standard reference on all aspects of OCL is [25].

In the DSL design context, we are primarily interested in using OCL to constrain class diagrams further, with some well-formedness conditions. Typically the diagram itself is not sufficiently expressive to do

this. In principle, one could write the well-formedness rules in natural language, but then we are not able to leverage automatic constraint checks provided by EMF.

In EMF, OCL can be used to write integrity constraints (similar to data integrity constraints, that you know from databases) and derived value expressions. The latter are used to specify default values of attributes, etc.

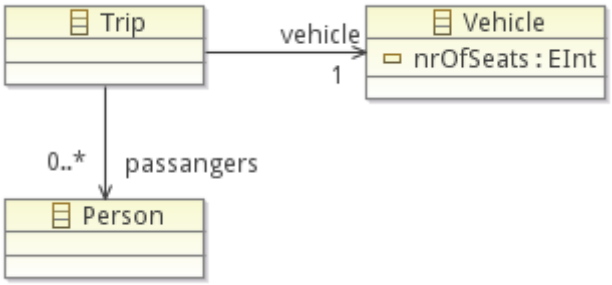
4.2 OCL Syntax and Semantics by Example

A usual invariant constraint has the following structure:

```
context ClassName

inv: OCL-expression
```

Such an invariant will apply to all objects of the given class. The expression must be a boolean expression (a predicate).

For example in the following model we are interested in describing trips that combine a number of persons (passengers) in a vehicle:

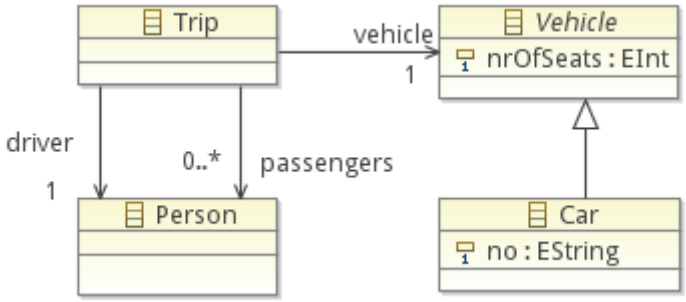


A natural integrity constraint is that the vehicle associated with the trip needs to be large enough to accommodate all the involved passengers. This constraint cannot be expressed directly in the diagrammatic language of class diagrams. It can be stated in OCL as follows:

```
context Trip
inv: passengers->size() <= vehicle.nrOfSeats
```

The type of the entire constraint expression is Boolean. It is written in the context of a Trip, so it applies to all instances of Trip. It contains numeric expressions, and a collection expression (`passengers->size()`). OCL contains a rich expression language that is syntactically similar to expression languages of modern object-oriented programming languages.

We want to add cars, and drivers to our model:



Further, we would like to make sure that a driver is listed on the passenger list. This is enforced using the following constraint:

```
context Trip
inv: passengers->includes(driver)
```

Like in Java, names of attributes and references are resolved locally in the context. So the above constraint is equivalent to:

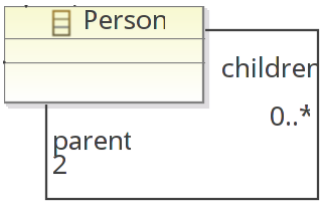
```
context Trip
inv: self.passengers->includes(self.driver)
```

We can also require that every Car is identified by its registration plate:

```
Context Car
inv: Car::allInstances()->isUnique(no)
```

The above is an example of getting a collection of all instances of a meta-class.

A common constraint pattern involves a kind of cyclic commutativity constraint. These are sometimes hard to understand, so we have a look at a special example here. A person can have zero or more children, and a child has exactly two parents:



It is expected that for a given parent, the parent is included in the set of parents of each of its children:

```
context Person
inv: self.children->forAll(c | c.parent->includes(self))
```

and dually each person is included in the set of children of its parents:

```
context Person
inv: self.parent->forAll(p | p.children->includes(self))
```

⚠ **Q. Create an instance of the class diagram that violates one of these constraints.**

Remark: In this particular case this integrity constraint can be maintained automatically by EMF, if the two references (parents and children) are related with the **EOpposite** tag. So OCL is not strictly required. This however, only works for simple cases. In more complex examples, you would need to write constraints like that in OCL.

4.3 Crash Summary of OCL

- In OCL navigation works like in Java:

  `ClassName.attribute.relation.operation().attribute ...`

  Operation calls are allowed, but then you should be careful that the operations have no side effects.

- Invariants can be named, to allow easier references:

  ```
  context Trip

  inv driverIsPassenger:  passengers.include(driver)
  ```

- Referring to enumerations

  `EnumerationClass::Literal`; for instance `Color::red`

- Supported operators include: `implies and or xor not`, `if then else` (this is the same as `? :` in Java); `>= <= > < = <> + - / * a.mod(b) a.div(b) a.abs() a.max(b) a.min(b) a.round() a.floor() string.concat(string) string.size() string.toLower() string.toUpper string.substring(int,int)`

- If navigation arrives at more than one object (via a link with multiplicity exceeding 1), we obtain a collection as the value. Use collection operators on this:

  - `Course.students->size ()` — size of the collection class
  - `Course.students->select (isGuest())-> size()`

    select the guest students, and count them
  - `Course.students->select(isGuest())->isEmpty()`

    no guest students are allowed in the course.
  - `Course.students->forAll(age >= 18)`

    this course is only for adult students

- `Course.students->forAll ( s | s.age >= 18)`

  equivalent to the above, but sometimes it is convenient, with a name for the iterated objects

- `Course.students->collect(age)`

  a collection of ages on this course.

- Notice that collection operators are introduced with an arrow (unlike in Java!)
- Other collection operators:
  `notEmpty`, `includes(object)`, `includesAll(collection)`, `union(collection)`
- Four types of collections: sets, bags, ordered sets, sequences (an ordered bag)
- When you navigate through more than one association with multiplicity greater than 1 you end up with a bag.
- When you navigate just one such association you get a set
- Now you get an ordered-set or a sequence if any of these associations was marked as `ordered`
- Let expressions:

```
let guests = Course.students-select(isGuest())
in guests->forAll( age >=18 )
                    and guests->forAll( age <= 20 )
```

- Single-line comments begin with two hyphens (like in Haskell):
  `-- this is a comment`
- Multi-line comments look `/* like in java */`

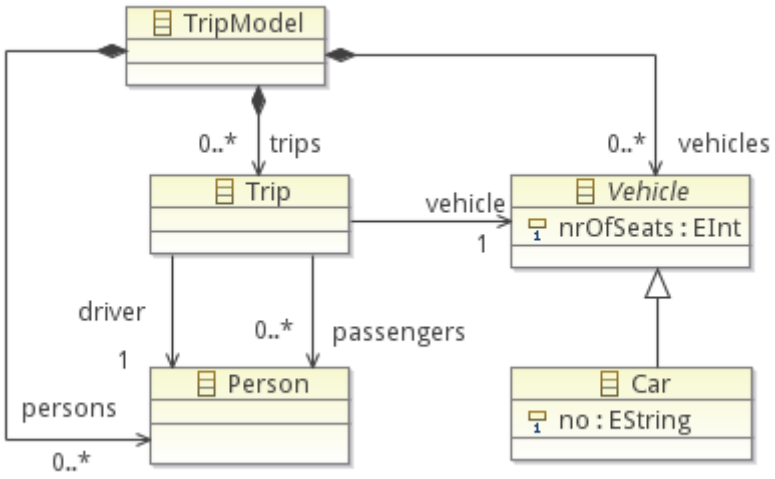
4.4 Playing with EMF and OCL Practically

The easiest way to play with OCL is to create an ecore Meta-model, derive an instance of this model dynamically, and then open interactive OCL console in Eclipse, which allows you to type in constraints and have them evaluated immediately. [1]

To derive a dynamic instance create an ecore meta-model, and right click on one of its classes. Choose "create dynamic instance". You will be asked to give a name of the xmi file, in which the instance will be stored. Now the editor opens and you can add children to the object created. Attributes and references can be specified in the properties view (right click on an object and choose "Show properties view").

In EMF all objects in a model need to be owned by some model class through composition (black diamond). You will not be able to create them otherwise. Thus it is usual practice to create one more meta-class representing the model itself, which owns all the other model elements. So our example looks more or less like below (note the TripModel class):

[1] In installation Eclipse (Indigo), for some reason I need to install "OCL examples and editors" component from the mirror "Indigo - http://download.eclipse.org/releases/indigo", before the console was available



Default editors for your models only provide for editing what was specified in our metamodel. So for example three instances of Person are visually indistinguishable. If you want objects to have visible identity, you need to give them names or identifiers. Often this is done by creating one abstract class *NamedElement* and making this class a generalization of all other classes in the model (then sometimes it is also convenient to make this class owned by the model class, instead of drawing compositions to all individual concrete classes like in the example above). If the name is unique, then it makes sense to mark that it has a property ID, in the advanced properties of the **EAttribute**. Then the tree editor will display it next to the object.

Before writing any constraints, you can check if your model satisfies the multiplicity constraints of the meta-model. You do this by selecting "Validate" in the context-menu of the instance model.

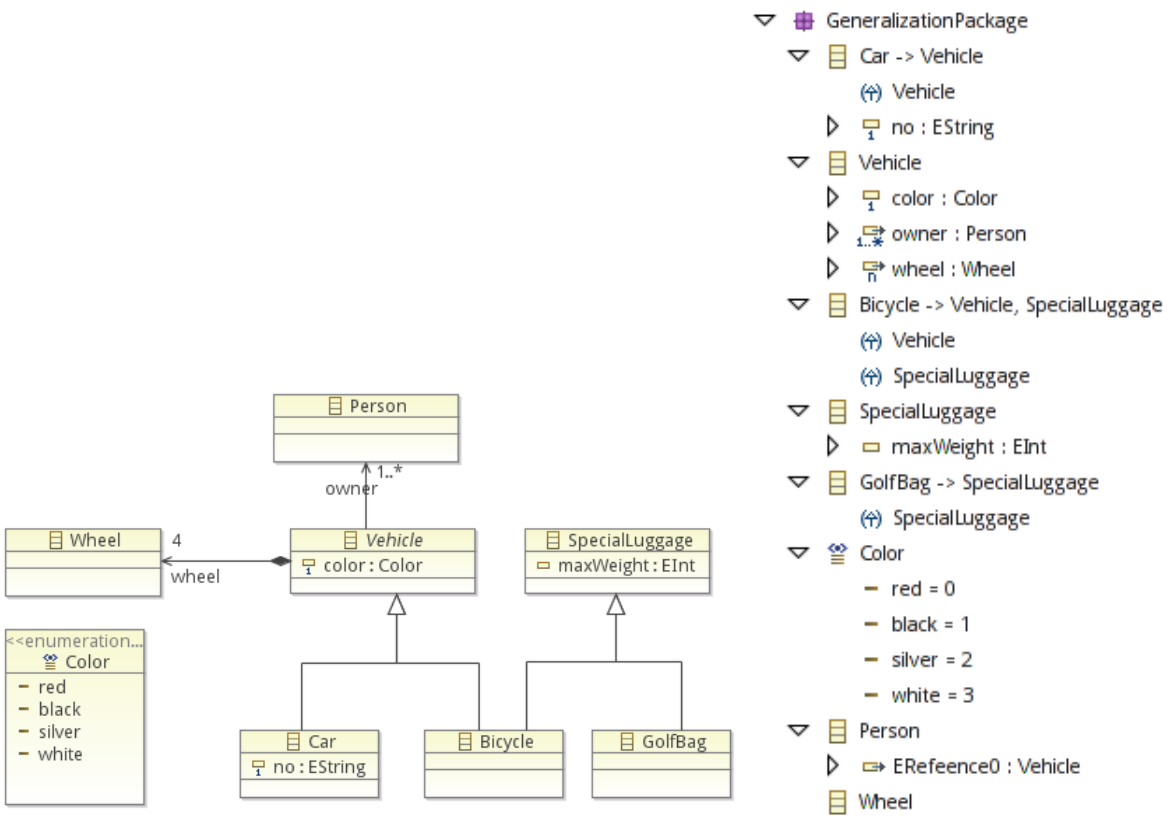
Once you created the instance dynamically, right-click on one of its elements, and open OCL console. In the console you can directly write invariant constraints. Omit the context and the `inv` keyword — instead make sure that the context element is selected in the model editor.

Note that constraints can be evaluated on M1 and M2 level (see console toolbar). At the M1 level, you can select a meta-class as context, and check if the constraint parses correctly. At the M2 level, you can select a model element (such as a concrete trip), and check if the constraint parses and evaluates to true.

This is very convenient to develop constraints (simple errors can be found immediately) — after that remember to store your constraint in a suitable file, or in the model, depending how you are going to use this.

Note: Code completion works in the OCL console (ctrl-space).

Finally, note that EMF by default supports tree editors, not diagram editors like above. This is a small syntactic difference, that is easy to get used to. Below you find a class diagram (one of the above examples) and a corresponding Tree View of the same diagram. Elements nested by composition are nested in the tree - so for instance attributes are nested under classes.



Such an editor can be automatically generated for your languages (or it can even run reflectively, as an interpreter for your meta-model). In my experience, being the main editor, the tree editor is much more stable than the diagram editor. The diagram editor stores the layout information in a diagram file, and occasionally the two files (models and diagrams) get out of sync, leading to complex errors. In such situation try to edit the ecore file in the tree editor, and if this does not help, simply delete the diagram file and work always in the tree editor (or reinitialize the diagram).

In the tree editor "-1" is used instead of "*" for multiplicity constraints.

5 Project Ideas

5.1 Preliminaries

Below you will find several stem ideas for the MDD project proposals. Please think personally, which topic is most inspiring to you. We will try to help you connect in groups with similar interests. Project proposals below are non-exclusive. Several groups can run the same project.

These are just project ideas. They serve as a starting points for discussion. The project is detailed in a dialog within your group, and with your supervisor, to make it much more specific. We will create a specific and binding project definition in the project agreement, filed in the ITU project base. There will be a lecture on how to write a good project agreement on 12/09.

I expect that we form groups some time between 12 and 19 September, and finalize the project proposals by 26 September (deadline).

Group size will be 4–7 persons.

Each project work includes writing a 15 page paper, in Springer LLNCS format. The 15 pages is a strict limit. No more pages will be allowed. Class files for LaTeX are here ftp://ftp.springer.de/pub/tex/latex/llncs/latex2e/llncs2e.zip, and the Word template is here: http://www.springer.com/cda/content/document/cda_downloaddocument/CSProceedings_AuthorTools_Word_2003.zip?SGWID=0-0-45-1124637-0. It usually takes quite an effort to fit all the material you want into 15 pages, so please take that into account when planning your available time for this project.

5.2 Project Ideas

Proposal 0. *Model Driven Re-engineering Case Study.* Study an existing open source project and understand its architecture and implementation. Design a domain specific language, that would help to raise some aspect of the implementation of this open source project to a higher abstraction level. Implement an editor for the language, and a code generator (or an interpreter) that can replace part of existing low level code, with generated (interpreted) DSL code. Evaluate how useful your DSL and implementation are.

This project proposal, along with proposal 4, are the most practical ideas in the entire collection. You should follow a process similar to our JUnit exercises, but perhaps on a larger scale. The main challenge here is selecting the right project, for which something interesting can be done in limited time period.

Proposal 1. *Textual Syntax for CVL.* Common Variability Language (CVL) is an upcoming OMG standard for imposing variability on any model implemented in any MOF-based modeling language. In simple words: for any model in a language with an ecore metamodel, it provides a way to parameterize it with some variability (for example optional elements). It links the variation points to variability

specifications which are very similar to feature models.CVL presently exists in an early specification form: a non-public document draft and a about a dozen of metamodels (abstract syntax) stored in the Rational Software Modeller XMI. The document also describes a draft visual syntax.The objective of the project is to propose a concise textual syntax for CVL and prototype it using TMF. We need a suitable editor, and a model transformation that maps it back and force to the original CVL metamodels.It would be an additional advantage if the syntax would be hooked to the INRIA prototype of CVL.

The CVL standard proposal is available at http://www.omgwiki.org/variability/doku.php (under news).

The project can be scoped by only giving syntax to a fragment of CVL.

Proposal 2. *Feasibility Study for CVL Modeling.* Common Variability Language (CVL) is an upcoming OMG standard for imposing variability on any model implemented in any MOF-based modeling language. In simple words: for any model in a language with an ecore metamodel, it provides a way to parameterize it with some variability (for example optional elements). It links the variation points to variability specifications which are very similar to feature models.

The objective of this project is to design a number of small and medium case study CVL models, based on published case studies, examples available online, and possibly own creativity, in order to assess ability of CVL to express common variability patterns, and to provide a number of standard test cases for CVL tools. The cases could be, for instance, used in a parallel project implementing a prototype of CVL.

There is no editors for CVL, so creating models has to happen via creating abstract syntax using emf generated editors, based on ecore meta-model of CVL. We could also consider using tools that are similar to CVL (SINTEF CVL, Clafer exporter).

The CVL standard proposal is available at http://www.omgwiki.org/variability/doku.php (under news). This project has good chance of influencing industrial practice, by influencing real modeling languages.

Proposal 3. *Prototype Implementation of CVL.* Common Variability Language (CVL) is an upcoming OMG standard for imposing variability on any model implemented in any MOF-based modeling language. In simple words: for any model in a language with an ecore metamodel, it provides a way to parameterize it with some variability (for example optional elements). It links the variation points to variability specifications which are very similar to feature models.CVL is in fact a small, domain specific, model transformation language, which only allows applying certain kinds of transformations, that are understood as typical in variability modeling. The objective of the project is to implement a model transformation that takes as input a base model, a CVL specification, and a configuration resolution, and generates a specialized model, as per definition of CVLs semantics.The implementation should probably be done using EMF + Xtext + Xpand (but other strategies are negotiable).

There is a report from last year, that could be used as a starting port. Possibly we can also get the code. This project has good chance of influencing industrial practice, by influencing real modeling languages.

Proposal 4. *Industrial Case Study Project.* The objective of the project is to prototype a DSL and an implementation of a code generator or interpreter for a particular software system. The project should not only perform domain analysis, and architecture re-engineering towards a model driven one, but also it should come up with strong arguments for such a change and an assessment of success of the

prototype. This may include assessment of how many lines of code could be saved in implementation; an assessment of execution performance improvements/loss; a proposal of how the new way of development can fit into organizational processes. It is a particular challenge in this project to develop an evaluation methodology that can in an unbiased way assess the success of your prototype and produce a recommendation for the software organization in question.

It should be an objective of the project to select a suitable technology. The choice of implementation technology (EMF, TMF, Microsoft DSL tools, MetaEdit+, dynamic programming languages etc) is often, but not always, dependent on the technological space used in the legacy system.

This is a generic project for students that have a strong link with a software development organization, in particular aimed at part-time students. Full-time students can join a group dominated by parttime students.

Proposal 5. *Case study in control modeling in ECDAR.* ECDAR is a modeling language (and a tool) for modeling real time control systems. It's models are timed state machines, so the they have similar flavour to statecharts, or to simulink models. ECDAR is a prototype tool, with all its issues. It needs case studies in order to develop understanding where it should improve, and how well it plays into industrial practice.

ECDAR is described in *Compositional verification of real-time systems using Ecdar* by Alexandre David, Kim. G. Larsen, Axel Legay, Mikael H. Møller, Ulrik Nyman, Anders P. Ravn, Arne Skou and Andrzej Wąsowski. Available at: http://www.springerlink.com/content/w207521301878462/. Should be accessible from ITU.

I propose to study the following publicly available PowerWindow problem from Mathworks. See http://www.mathworks.se/products/simulink/examples.html?file=/products/demos/simulink/PowerWindow/html/PowerWindow1.html. This is an example from automotive domain. One would have to model again the same behaviour in ECDAR and try to verify the requirements statement. In the interest of time, a fragment of the case study could be used.

This is particularly well suited for students with mechatronics background, or embedded systems interests. It is rather weakly linked to Eclipse technology, as ECDAR is a completely Eclipse-independent tool.

Proposal 6. *Code generation from Ecdar.* See information about the ECDAR tool above. This project is concerned with real time, concurrent automaton model expressed in ECDAR and generating Java/C#/Real-time Java code from it. What you generate should be an implementation of the composed machines. To make the project feasible, we could restrict just to parallel composition, as the only operator of ECDAR (ignoring other composition operators). For each machine in a parallel composition we need a separate thread. Timer's might need to be used to decide when actions happen.

We could use the ECDAR textual editor, and the ECDAR metamodel developed by Bastian Müller.

Proposal 7. *Simple DSL for BDD in Java.* Behaviour-driven design is a recent hype mode to link requirements, tests and code in a simplelightweight manner. The most popular implementation is for Ruby. You will find lots of inspiration at http://cukes.info/ (google for the Cucumber project). In this project you design a DSL and supporting framework that allows doing BDD in Java. If success-full you can consider refactoring your project so that it uses BDD itself (boot-strapping). Your claims can for example be about suitability of Java for BDD, or about difficulties of using BDD in a MDD project.

See also http://behaviour-driven.org/.

Proposal 8. *Building Information Modeling Pilot Study.* Building information modeling (BIM) is a process involving the generation and management of digital representations of physical and functional characteristics of a facility. See http://en.wikipedia.org/wiki/Building_information_modeling. Green Building XML schema http://en.wikipedia.org/wiki/Green_Building_XML was developed to facilitate a common interoperability model integrating a myriad of design and development tools used in the building industry. Study the field and design a DSL (concrete syntax) that can represent models in the schema, in human readable form. Perhaps consider transformations to some visualization tools, such as Graphviz, etc.

This project, could serve as a preparation for thesis in the Energy Futures area.

Proposal 9. *EMF-based Models of Natural Language Documents.*

Software systems contain not only source code but usually also artifacts written in natural language, for example LibreOffice or Word documents. Such documents may contain requirement specifications, architecture descriptions, etc. Obviously, natural language artifacts are inter-related with computer language artifacts. For example, architecture descriptions describe relations of source code artifacts to each other and requirement specifications are refined into models, source code, etc.

Natural language documents are hybrid structures in the sense that they contain a document's structure as well as the structure and content of the actual sentences themselves. Both of these facets can be modeled in a domain-specific language to allow for easy integration with other domain-specific languages and source code.

In this project you will create an EMF model to capture the structure of natural language documents returned by either the Apache POI project (http://poi.apache.org/) or by the Rich Text Format, you will define a textual DSL integrating the output of the The Stanford Parser (http://nlp.stanford.edu) into the EMF technical space, you will formulate relations between the models for natural language documents and source code artifacts for an exemplary software system.

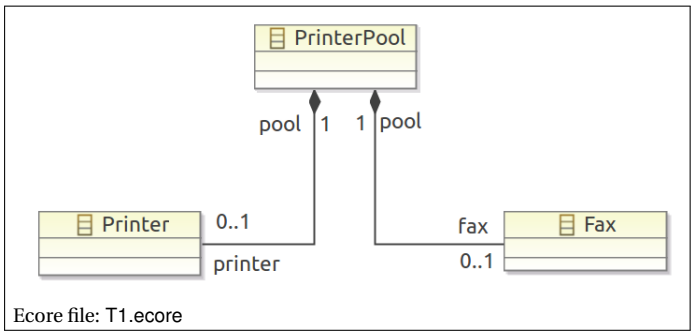
6 Exercises

Objectives

- To experience limitations of expressiveness of class diagrams
- To practice precise expression of constraints over structural models
- To identify constraints in a realistic project (JUNIT) and express them
- To recognize the differente between the isntance level and model level in practice (by interpreting constraints at the two levels)

I estimate that these exercises can be done within 4-5 hours in total.

Let us first start with an example (ecore files available from LearnIT):



Ecore file: T1.ecore

OCL constraints can be evaluated in the OCL console. Start the OCL console, by first creating a model instance (context menu in the ecore editor of the PrinterPool class), and then select the OCL Console context menu entry on the PrinterPool instance class instance. The OCL console automatically sets the context of your constraint to the highlighted ecore element.

The following OCL constraint, written for the above diagram, will not work in the console:

```
context PrinterPool
inv printer->notEmpty()
```

Instead write:

```
printer->notEmpty()
```

and make sure that PrinterPool is highlighted in the ecore editor. This constraint says that every PrinterPool contains at least one printer.

Experiment with the M2 and M1 modes of the console.

⚠ **Task 1.** Write each of the following constraints in OCL, or in Java, as requested. You are encouraged to perform these exercises on a PC, using Eclipse or some UML modeling tool.

1. *Every printer pool that has a fax, also has a printer.* Write the constraint in the context of the PrinterPool class, of the diagram T1 (above).

   Create an instance of the above model that satisfies the constraint and verify in the OCL console that this is indeed the case. Create an instance of the above model that violates this constraint and verify in the OCL console that this is indeed the case. Repeat these steps for every constraint below, to sanity check (test) your constraints.

2. Write the constraint from the previous point in the context of (an instance of) class Fax.

3. Write the above constraint in a form of a Java (or C#) assertion written in the context of the PrinterPool class.

   In order to be able to write Java assertions, you need to generate Java code from the provided EMF models. To do that first create a generator model (File > New > Other > EMF generator model). Name it the same as the ecore file, but with .genmodel as an extension. Use the ecore importer, when asked, and indicate T1.ecore file as the source. EMF generators run not directly of ecore models, but of generator models, which contain a number of properties that can customize the generator.

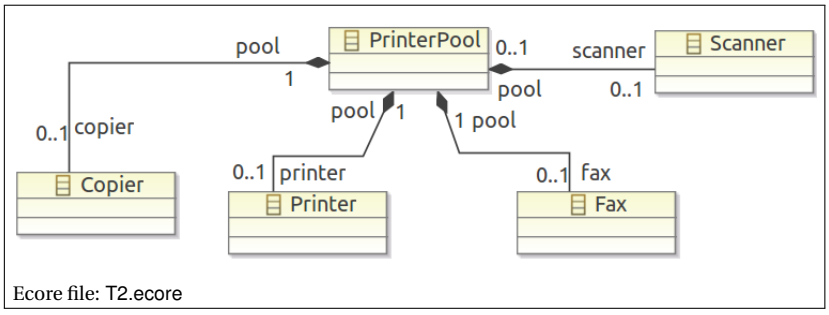
   After the .genmodel is created, open it (double click) and select "Generate Model Code" from the context menu of its root node. This will create an implementation of the model in Java. You wil find it in the src/ directory of the same Eclipse project/package.

   Add a **void** method `validate()` to PrinterPoolImpl.java and write your constraint in this method. Note how EMF generators split the implementation of PrinterPool into two files: the interface in PrinterPool.java, and the class definition in PrinterPoolImpl.java. The Java compiler will check whether your method is type correct.

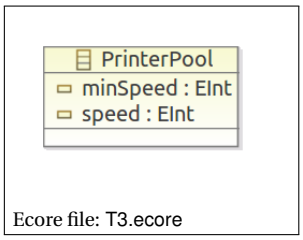
   Observe that your method will not be overridden when you run the generator again. This only happens for methods with annotated as `@generated` (See other methods in the file).

   To save time, we will not run our assertion. We just want it to typecheck and compile. Discuss with a fellow student what is more readable: your Java assertion, or the OCL constraint. Why is that?

4. *Each Printer pool with a fax, must have a printer, and each printer pool with a copier must have a scanner and a printer.* Write this constraint in OCL in the context of the printer pool.

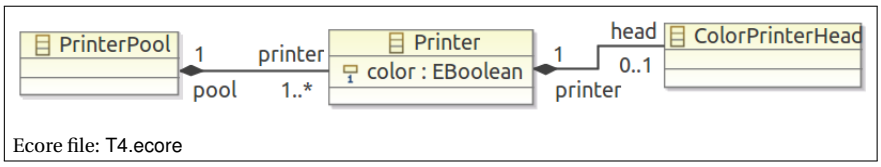


5. *PrinterPool's minimum speed must be 300 lower than its regular speed.* Write the constraint in the context of the PrinterPool.



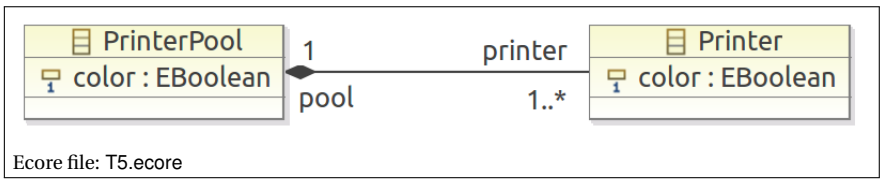
Ecore file: T3.ecore

6. *Every color printer has a colorPrinterHead.* Write the constraint in the context of the class Printer.



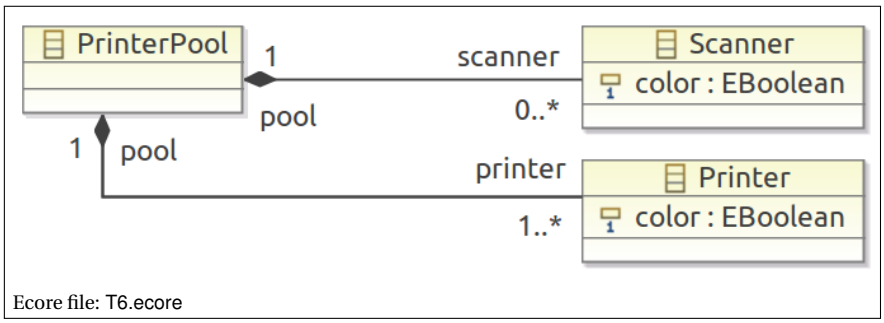
Ecore file: T4.ecore

7. *A color capable printer pool contains at least one color capable printer.*



Ecore file: T5.ecore

8. *If a Printer pool contains a color scanner, then it must contain a color printer.*



Ecore file: T6.ecore

9. *If a printer pool contains a color scanner, then all its printers must be color printers.* Use the same diagram as above.

10. Assert the above property in Java (just like in point 3 above, requires generating code from T5).

    Hint. Before generating code within the same project as in point 3, you may want to change the Base Package property in the .genmodel to some nonempty value, say "T6". This will place the generated code in a new package, and avoid clashes with the code generated earlier for T1. Note that these models (T1.ecore and T6.ecore) are incompatible. Why?

    Whic version of the constraint is more readable? OCL or Java? Why?

11. Use the same diagram to assert in OCL that *there is at most one color printer in any pool*. You may want to use the `select` iterator.

Task 2. Now analyze your notes from last week's exercise, and identify constraints that cannot be expressed in the diagrams. Alternatively, if you luck such, you need to further study the documentation and implementation artifacts in order to identify such constraints.

Once you have the constraints, write them in OCL, and verify on example model instances.

For example, in my model any Before method must have @Before annotation. Annotations and fixtures are modeled separately in my model. I needed to add a constraint that the *method which takes the role of Before method with relation to a Fixture class, must alo have an annotation, and that annotation is Before*.

Similarly, in my mode, an After method should have an after annotation, which led to the following constraint in my model: `Fixture: after.annotation.isOclType(After)`.

You should be able to identify a small handfull of similar constraints in your model. It is unlikely that there will be many, because class diagrams already allow to express most typical patterns. So only atypical, usually few, aspects of the model require use of OCL.

Side remark: If you wanted to include constraints on a more permanent basis, they can, for example, be stored in the model as annotations, and integrated in model validation. Here is a tutorial on how this is done: http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html

7 Domain Specific Languages

Reading: Concepts like abstract and concrete syntax, parsing, grammars, left-recursion are explained in classical compiler text books. If you forgot them then check out one of the standard compiler texts, such as [1].

Fowler gives a good coverage of design issues, implementation techniques and example languages in [7]. If you follow a re-engineering project, or any other project that requires DSL design, I recommend reading these: [15, 14, 19, 26, 18].

Recall that our intention is to automate development of software applications in a given domain, by using models to describe essential characteristics of an application, and using code generation to produce the application automatically. In order to make this approach work, we need to make a model in a language that is suitable to describe systems in this domain. Such a language is called a *domain specific language*.

So before creating models, and before creating code generators interpreting these models, we need to design the language in, which these models are written. What does it take to design and implement a language?

1. Concrete syntax
2. Abstract syntax
3. Editing/Modeling environment for models
4. Serialization/Deserialization for models
5. Static semantics (mostly type checking)
6. Dynamic semantics (interpreter, code generator)

Is this is a task that a typical software house can undertake? The answer is yes, mostly due to an amazing progress of DSL technologies.

7.1 DSL Implementation Strategies

There are essentially four approaches of implementing domain specific languages:

1. *Parser generators* such as ANTLR, or bison. This is the heaviest method.[1] It requires specification of the grammar of your DSL, and then usually a separate implementation of a syntactic analyzer that simplifiers the parse tree into an abstract syntax tree, and performs type analysis. This is the method applied typically to general purpose programming languages. It gives an efficient

[1] The only harder way would be to manually implement a parser — nobody sane does that today.

front-end for the compiler, which is important if the models need to be parsed a lot, and/or if they are large. It does not give you anything more than that, though.

2. *Language workbenches* with textual syntax support — Xtext[2] is the most mature tool of this kind. A language work bench includes a parser generator, but circumvents creation of complex parse trees, since it performs a model transformation in the process aiming at a concrete metamodel. Also it can validate parsed models using constraints specified declaratively (for instance in OCL or in some model querying languages). Finally it generates other elements, like full-blown editor with text completion and syntax highlighting, well integrated into Eclipse IDE. Xtext integrates directly with EMF (an EMF metamodel can be used as an abstract syntax specification), so it allows to benefit from all the advantages of EMF discussed before.

3. *A visual language workbench.* Such workbench performs for visual (diagrammatic) syntax what Xtext attempts to do for textual syntax: it would generate a diagram editor including serialization and de-serialization code for the syntax. So it is a suitable solution if diagrammatic syntax is a must in your application. Here GMF (the graphical modeling framework) seems to be still the most popular solution, but also *Microsoft DSL Tools* for Visual Studio falls into this category.

4. *Embedding the DSL* into a concrete syntax of an easily extensible language such as ruby.

7.2 Internal (Embedded) DSLs

⚠ A DSL is called *internal* (or embedded) if it is implemented within a programming language, typically as a library. This requires a language that has a reasonably flexible syntax. Many dynamic scripting languages have that, but also C++, Scala, Haskell, and Standard ML are known for being usable as *host languages*. Here is an example of a parser implementation (fragment) in Scala:

```
val ID = """[a-zA-Z]([a-zA-Z0-9]|_[a-zA-Z0-9])*"""r
val NUM = """[1-9][0-9]*"""r
def program = clazz*

def classPrefix = "class" ~ ID ~ "(" ~ formals ~ ")"
def classExt = "extends" ~ ID ~ "(" ~ actuals ~ ")"
def clazz = classPrefix ~ opt(classExt) ~ "{" ~ (member*) ~ "}"
def formals = repsep(ID ~ ":" ~ ID, ",")
def actuals = expr*

 def member = (
      "val" ~ ID ~ ":" ~ ID ~ "=" ~ expr
    | "var" ~ ID ~ ":" ~ ID ~ "=" ~ expr
    | "def" ~ ID ~ "(" ~ formals ~ ")" ~ ":" ~ ID ~ "=" ~ expr
    | "def" ~ ID ~ ":" ~ ID ~ "=" ~ expr
    | "type" ~ ID ~ "=" ~ ID
   )
```

[2] Xtext is part of Eclipse TMF, which stands for *Textual Modeling Framework*.

In this example Scala is the host language, and the parser combinator language is the embedded DSL (also an internal DSL, and the hosted language).

You can probably easily interpret this syntax, as it is very close to standard EBNF like notations. Crucially, however, this is not a specification, not an input for a parser generator or transformer. This *is* the parser implementation. This code is directly executed using the compiler and execution platform of Scala (here, incidentally, this includes JVM). No code generation or model transformation is involved.

Experienced programmers in any of the above languages can construct such embedded DSLs quite easily and quickly — not harder than implementing a library. So solution (4) above is probably the cheapest (fastest) possible. Another advantage is that your DSL can be more or less freely be mixed with the host language — so you can cheaply get a rich expression sublanguage.

Its disadvantages include that error reporting is typically done using the types of the hosting language (so you get very complex type errors for example). Also, This approach gives you a quick way to get an interpreter for a language, but it does not provide all the other integration. For example it does not give you an editing environment, unless you implement it yourself. On the other hand the editing and testing environment of the host language can be used, and is often sufficiently convenient for simple DSLs. Finally, should a more 'proper' editor be developed, it can always be done later, independently, using a language workbench.

If you are interested in Embedded DSLs, [7] has a chapter about them, which is a good gentle starting point.

7.3 Sinatra: an Example from Ruby World

Sinatra An Example from Ruby World

- A popular DSL in the Ruby community
- Used to write web applications
- Built around the main HTTP actions: GET, POST and PUT

Sinatra (http://www.sinatrarb.com/intro) is a popular DSLs in the Ruby world. It is used to write web applications. We will have just a quick look at it, to get a hint on how it is implemented.[3] Sinatra's syntax is build around the basic verbs in HTTP protocol: GET, POST and PUT. Here is a small piece of Ruby code using the sinatra library (yes, in Ruby, like in Scala, internal DSLs are implemented as libraries).

```
require 'sinatra'

get '/hello' do
~~'Hello world!'
end
```

[3] This part of the class is based on http://www.sinatrarb.com/intro and http://rubylearning.com/blog/2010/11/30/how-do-i-build-dsls-with-yield-and-instance_eval/, as seen 2012/09/11

This code outputs the web page containing 'Hello world!' whenever a user accesses the web app with a GET request for the '/hello' location on the webserver running the application. So in a standard Ruby this corresponds to something like the following code:

```
app = NoDSL::Application.new

app.on_request(:get, :path_info => '/hello') do |response|
~~response.body = "Hello world."
end
```

Similarly, Sinatra allows writing `post` and `put` *routes* (route in sinatra terminology is a block linking request to executable code, like above). Routes are matched in the order they are defined. The first route that matches the request is invoked.

Route patterns can be parameterized:

```
get '/hello/:name' do
~~# matches "GET /hello/foo" and "GET /hello/bar"
~~# params[:name] is 'foo' or 'bar'
~~"Hello #{params[:name]}!"
end
```

With the above code, if you access `/hello/Andrzej`, the app will respond with *Hello Andrzej*.

How is this implemented? One important construct in Ruby that facilitates meta-programming (so implementation of DSLs), is `yield`. It stops evaluation of current method and evaluates the block passed into the method, calling it with any arguments supplied in the yield statement itself.

```
def map(l)
   result = []
   l.each do |item|
     result << yield(item)
   end
   result
end
```

With the above definiton we get `[2,3,4]` as a result of calling `map([1,2,3]) {|i|i+1}`.

Now the GET route in Sinatra, basically is a function (method) that matches its argument pattern to an incoming request, binds parameters to values during this matching, and yields to the block provided within `do ... end` after the call. This is very common pattern in internal DSLs in Ruby. You can see the same technique applied in the step definitions of the Cucumber project (http://cukes.info/).

We have seen that yield and code blocks (default continuations, if you wish) are used in internal DSL implementation. Ruby provides a number of other constructs that support meta-programming. For example you can overload the dynamic method dispatch mechanism, for example, to translate all called methods names from Danish to English, using Google Translate, before you actually attempt to call them. For more serious applications you may use this to modify the list of parameters, etc.

Facilities to use variable names in strings, and to parse regular expressions are also used very often in DSL design.

Most ruby libraries these days evolve towards DSLs. Internal DSL development attracts a lot of attention, and if you are interested in this, feel free to propose/select projects in this area.

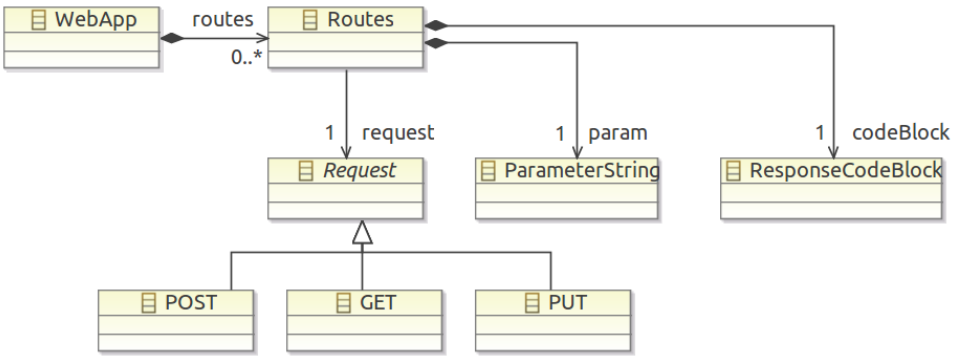
Exercise: *Next week we will develop a textual syntax for the Trip language. If you are interested in embedded DSLs, you could try to quickly develop an embedded DSL with similar syntax for the same kind of information in the language of your choice.*

This exercise turns into a larger project, if you also wanted to integrate this with EMF, so that models are interchangeble with EMF through files. An easy way is to use textual concrete syntax, and make it precisely the same as in the XText example in the following section. If you would like to go via XMI serialization, then you should probably consider a host language in which there already exists an implementation of EMF. Rumours are that there is such for Python; supposedly available online.

7.4 Towards External DSLs. Metamodeling

One problem with internal DSLs is the lack of separation of the host language from the DSL, so programs/models easily become messy. Another might be that the design of internal DSLs often resembles programming way to much, whereas a design of a good DSL for your system requires capturing essential knowledge in the domain. This is best captured by building a domain model — a class diagram capturing key concepts and relations between them.

For example, for the Sinatra DSL, a good design process would be to ask yourself the following questions: what are they key actions that any web app needs to handle? what kind of parameters these actions takes? What reaction is expected? How they are prioritized. This could lead to a meta-model similar to the following:



Creating such a model is well done in a brainstorm, when multiple stakeholders and subject matter experts can share knowledge, correct each other, and add details. A bit like we tried last week with the JUnit example. This is easier done using a conceptual model, than a Ruby code that mixes both domain concepts and the implementation details.

Of course, the above model is just a simple example. A deep analysis of the domain would create a much more complex model, possibly with structure of parameter patterns, and constraints between patterns and the code block.

Fortunately, due to availability of language workbenches, there is no need to throw away these models, before attempting an implementation. Modern tools like Xtext, EMFText, Monticore and Spoofax can take metamodels and generate (large parts of) language implementations from them automatically. We will see how this works next week.

⚠ 7.5 Some DSL Design Principles

Begin with identifying the **purpose of the language**. What it will be used for? What are the stakeholders and usecases? This will strongly influence the concepts in your language. Most popular applications are knowledge represetnation and code generation, but they have very different requirements. A language admitting underspecification cannot normally be used for code generation [14].

The language must be simple. Implementing a complex language will use a lot of resources. Keep the number of concepts as small as possible, and avoid redundancy (ability to express same things in many ways). Also accept that your language will be incomplete. It is dangerous to create a language that covers or possible general cases. It is more important to create a language that covers cases appearing in practice, and the plan for language evolution [15, 14].

It is usually a bad sign if your language becomes dominated by typical programming constructs like lops, branching, functions, classes. That is often a sign that your abstraction is not close enough to the domain. It is better to stick to the problem domain as close as possible [15, 24]. However, if your language is meant to describe large complex systems, consider adding modularity constructs to it. Large models need to be broken in smaller pieces [14].

It is dangerous to just work with metamodels. **Test the language during design**, by creating several examples of models in your language. This can be done even without concrete syntax. You can use dynamic instance creation in EMF, or invent ad hoc concrete syntax (on paper). Creating instances uncovers design errors easily [14]. We have successfully used this method in the CVL design methodology.

It is better to **use the problem domain as inspiration**, than the solution space, *even* if an implementation exists, for example in re-engineering scenarios. Of course, one should be realistic, and still design a language that can be realized on top of an existing framework. Solution space constraints should not dominate the design though. [15]

Make sure that your langauge does not overemphasize one domain feature, at the cost of other important features. Also make sure that the language allows to build many useful models (not just two...). [15]

We will come back to more design guidelines when talking about concrete syntax, and about product line architectures in later lectures.

8 Writing a Project Proposal

8.1 Why ?

It is a task of a student group to write a project agreement, with some advise from the prospective supervisor. The agreement is officially registered in the project base. To access the project base log into http://my.itu.dk.

When accepted by both sides (the students and the advisor), and confirmed by the study board, the project agreement becomes the legal basis for the final exam. It is made available to the examiners together with a copy of your report. Report results are evaluated against the project agreement.

A messy project agreement can seriously impact your grade. A good definition will improve your exam experience and helps to start working on your project more smoothly.

8.2 Problem Definition

A thesis is an *academic project*. A typical academic project has the following components: a problem statement, a problem analysis, choice of a solution method, application of the solution method, and a reflection on the outcomes. Thus in order to define a project you *have to define the problem* that you want to solve. I suggest three simple steps to formulate a problem definition, approximately one paragraph per step.

1. *Introduction: What is the subject area?* This paragraph provides a very basic background on the problem. It serves as introduction for readers of the project agreement. Some examples of introductions from actual projects agreements:

   > Code duplicates (clones) are a common problem in large code bases. They occur when a programmer reuses code through copy-and-paste instead of referencing the existing code. The reasons for doing so can be numerous. We just acknowledge that clones are likely to occur in large codebases.
   >
   > From a project titled *Refactoring of Dynamics NAV*

   > Some Open Source communities consist of large numbers of individuals collaborating on the development of Open Source software products. These individuals are distributed, in every sense of the word. They collaborate across geographical locations, often without any geographical organizational center. They do not necessarily share educational backgrounds, professional experience level, personal maturity, working culture, work ethics or cultural values. Still many of these communities manage to build and maintain a collaborative modus and organizational culture that enables them to efficiently achieve the goals they set out for the project.
   >
   > Tor B. Sørensen. *Efficient open source communities. What drives them?*

> Feature modeling is the activity of modeling common and variable properties of products. A feature diagram consists of a set of nodes, a set of directed edges and a set of edge decorations. It is known that such feature models can be translated into propositional formulas.
>
> Yuan Gao. *Flexible Interactive Derivation of Feature Models*

2. *State of the Art.* This paragraph aims at narrowing the problem, by listing typical existing solutions, or research results in the field. For example:

   > Within his thesis Michael Schou Christensen has developed an efficient way of finding duplicate source code matches in large codebases. Upon his framework, Tijs Slaats, Till Blume and Roland Schlosser have developed a specific implementation for the C/AL language.
   >
   > Tijs Slaats. *Visualization of code duplication matches in large codebases*

   > In 2007 an algorithm for computing reduced feature models has been presented by Czarnecki and Wąsowski. This algorithm needs to be extended in order to construct a regular feature model.
   >
   > Victoria Kuzina. *Interactive Derivation of Feature Diagrams*

3. What is the *specific* problem *you* are solving? What is specific about your envisioned solution? Often this is easiest described as an improvement over existing work, described in step 2. Sometimes it is more convenient to swap these two points—define your task first, and only afterwards argue that this is an improvement. You can choose either order.

   This is the most important paragraph. Make sure that you use *verbs* that clearly state what you will do. Examples:

   > During their project they [Slaats et al] noted a number of possible improvements and enhancements to Michael's framework, one of these being the visualization of the code duplication matches that were found. The problem they found with the current representation of matches is that it can be hard to get an overview of how the matches relate to each other, especially for people not already familiar with the original base.
   >
   > The focus of this project will be to look into this problem and find more robust visualization techniques that can be used to represent these code duplication matches.
   >
   > Tijs Slaats. *Visualization of code duplication matches in large codebases*

   > This thesis will outline the basic of collaborative moduls and organizational culture and attempt to identify the main factors contributing to the establishment and maintenance of these tenets. In other words the research question of this thesis paper will be: "What are the main contributing factors that build and maintain an efficient collaborative culture in large Open Source Communities?
   >
   > Tor B. Sørensen. *Efficient open source communities. What drives them?*

In more generic terms examples of suitable problem definitions are:

- Implement an algorithm and evaluate it experimentally on a specific class of inputs; compare with competing solutions.
- Read a book that introduces a certain method (let us say programming paradigm) and apply it to solve a specific problem. Reflect on the benefits of using this method in this particular case.

Examples of projects that do not normally qualify as research (and thus risk rejection):

- Implement an algorithm.
- Read a book.

It is important to see that the two latter project kinds can in principle have the same contents, as those specified formerly. Nevertheless the former project descriptions are acceptable, whereas the latter are not. They lack detail in description, that warrants academic level of activities. The first one lacks reflection and analysis, the second one lacks any intellectual creativity.

Agreements are often rejected not because their prospective content is not at the right level, but because the academic activities are not explicit in the description.

8.3 Method

Describe how you are going to address the problem defined above. In here you should include both *general* and *specific* methodological plans.

General methodology: What are the basic tools of your work? User studies? Experiments? Prototyping? Mathematical proofs? Literature surveys? How are you going to collect data (if any)? How will you evaluate quality of your design or implementation?

Specific method: Apart from general methodology, indicate specific methods, if you are planning to apply them. Examples are: model driven engineering, object oriented analysis and design, proofs with logical relations, original research methods/processes outlined in a book or a paper that you plan to follow, etc.

8.4 A Checklist

- You must be able to report about your findings. If you cannot write a clear report based on your work, then this is not a good project definition.
- Make sure that you have described the objectives precisely and unambiguously, so that the examiner can assess whether you have met your goals in the end. If your definition is vague or hard to understand, you risk being graded against an interpretation of you description, not the description itself.
- Check the spelling and grammar in the title of the project. This is exceptionally important, as this title would usually end up in your transcript of records when you graduate from ITU.
- A short bibliography is welcome, especially in definition of thesis projects. In most of the projects it is easy to point out to the main reading on the problem definition, and to the main reading about the solution method.
- Spell check the entire text. Ask a friend to proofread it.
- The project agreement has to be understandable to non-experts in the field. Run it through your fellow student at ITU, who does not focus on the same narrow area as you do. Ask him whether the description is understandable.

8.5 When?

Do not plan to send your project proposals to the prospective supervisor in the very last moment before the deadline. The project proposal has to be accepted by your supervisor, and this often requires several cycles of discussion and improvements.

By sending the project for approval just before the deadline you put your advisor in an uncomfortable position of choosing between accepting a badly written proposal, which will cause trouble later, or rejecting the proposal right away, causing trouble for you immediately on the spot.

8.6 Administrivia

The roles of respective teachers in projects:

Andrzej Wąsowski: group supervisor
Helge Pfeiffer: group supervisor
Thorsten Berger: group supervisor
Bogdan Petrutescu: teaching assistant, help desk on Eclipse




Projects are written in 4-7 person groups. Size of the group is mandatory, given the objectives of the project cluster. Please form the groups and send an email with member lists to bpet@itu.dk. Please also write to Bogdan if you cannot find the group. We will create groups for those who cannot do this themselves.

Project Deliverable

- Research paper-like report
- Report maximum 15 pages, LNCS format
- Springer provides templates for LaTeX and Word

9 Evidence in a SE Project

Reading: The two sources that I suggest today [22, 23] cover somewhat broader topic than just evidence. They tell you how to construct a report that is convincing in proving its hypothesis.

9.1 Introduction

In the following lectures we will change focus from technology, to writing a perfect report, and a perfect MSc thesis. This will necessarily not only talk about writing, but also about constructing the project, prioritizing the work that brings most value, and seeing your own work in context.

Most research papers has the following structure:

1. Introduction
2. Background
3. *Real stuff comes here*
4. *Describing your technical solution*
5. *Or performing some analysis*
6. Evaluation
7. Threats to Validity / Discussion
8. Related Work
9. Conclusion

It is rare that a research paper has more than 10 sections. Note that this structure also applies to many research thesis (including MSc thesis), where sections become simply chapters. The order and list of sections may differ occasionally. It is instructive to assess the papers you are reading as related work for your project in this respect — this will give you some experience.

For instance sometimes it is convenient to place the related work section early (usually under the title "State of the Art" in such case). This more common in longer papers (journal papers) and in theses. In theoretical papers Evaluation is spread all over the contents.

The paper is preceded by an *Abstract*, which summarizes the context and the contribution. Readers select papers to read based on title and abstract, so abstract needs to be excellent. Also for your thesis.

Introduction motivates your work, explains your contribution and summarizes its potential impact. Normally in the introduction you state your *objective* or *claim*.

Background provides all the basic information necessary to proceed with reading (for a generally educated computer scientist). This could be standard definitions, setting notations, etc. This is often necessary to disambiguate from many possible presentations, assumptions etc.

Most of software engineering papers contain *Evaluation*, which as a purpose has to provide evidence for your claim, stated in the introduction. If your claim is that my heuristic procedure is faster than known approaches, then the evaluation should show measurements proving that.

Threats to validity — this section is included in papers which have an experimental evaluation section. It should indicate any potential weaknesses of your evaluation method, that you are aware of. For instance that you benchmarked your heuristic only on random input data, and not on realistic data, or that you only used a multi-core machine. In the former case, it is unclear whether the results are transferable to realistic input. In the latter case, it is unclear how well the procedure will perform on a single core machine.

"Threats to validity" are omitted in theoretical papers, or in strictly technical papers, that do not use experimental evaluation. In such case it is more common to include a *Discussion* section, which evaluates the robustness of results, and shows how it interacts with other known knowledge. For example, what assumptions can be lifted easily, or whether the work is controversial, or supports the existing knowledge about related problems.

In *Related Work* you summarize what others have written on the subject. This is treated very seriously by reviewers and committee members.

You should not forget to summarize the story of the paper in the *Conclusion*. This often allows you to present it in a different language than in the introduction.

We will come back to construction of the Abstract, Introduction, and Conclusion trio a bit later in the semester (when you will be writing these). Now, we will focus on the main axis of any project: aligning your argumentation and evidence with the claims you make.

This lecture is on critical thinking. On how to construct a strong argument, and how to see weaknesses in arguments used in papers authored by others.

9.2 Matching Evidence to Claims

Most projects in computing fall into one of the three categories:

- **Empirical** projects
- **Engineering** projects
- **Theoretical** (mathematical) projects in CS

An empirical project aims at understanding properties of some phenomenon in reality. For example whether people make less errors in Java programs, or in C# programs, on average. Or what are the main reasons for failures of IT projects. Or which model transformation technology is more efficient in which case.

An engineering project aims at developing some technology. It often needs to argue in what sense this technology is better, than known so far. Often, when the technology is completely new, one has to simply argue for its quality.

A theoretical project aims at understanding mathematical properties of computing objects. For example what is the fastest way to sort, or what are properties of semantics of a programming language.

These categories do overlap quite strongly. Empirical projects often require engineering substantial infrastructure to carry out the experiments. For instance in model transformation benchmarking, you need to design a benchmarking harness, and generate test instances. Engineering projects often have to use mathematical methods to argue for correctness of solutions, and empirical methods to evaluate quality. Theoretical projects, often support themselves with implementations, to motivate for relevance of results.

By far most of SDT projects fall into the second category, *Engineering* with excursions into empirics and theory. This, unfortunately (?), means that we need a broad perspective on methodology.

There exists an extensive literature on research methodologies. In this lecture we only want to give a few practical rules of thumb. The follow up course (Advanced Software Engineering) will give you a much more thorough and systematic insight into research methods.

So you design, engineer, a piece of technology. What is your claim ?

Types of Claims in Eng. Projects

- **Feasibility** — it was not possible, it is possible now.
- **Quantifiable Improvement** — a quantitative improvement over previous work, for example in terms of performance, memory consumption, precision etc.
- **Qualitative Improvement** — often qualities can be pointed out in a discussion, although this is the weakest kind of evaluation. For instance one could provide a qualitative discussion of advantages of an object oriented programming language over a procedural one.

Here I give some examples inspired by your projects.[1]

Traceability Example Question: What is The Claim?

> Maintaining traceability links between models and code, allows uncover many programming errors.

Clafer2Ecore Example Question: What is The Claim?

> Clafer is a new language with a similar objective to ecore. Since it is a new language, there are many advantages of translating models from other languages (in particular from ecore) to Clafer. It allows to provide a benchmark for Clafer infrastructure. The opposite translation can be used to exploit ecore infrastructure for Clafer based development...

[1] Please do not take them literally for your projects. They are not fully worked out. Likely you want to do better

MT Example Question: What is The Claim?

> Model transformation technology (MT) has a broad application range, whenever data that adheres to some modeling language, needs to be translated to some other formalisms. In MT use cases range from very small models processed occasionally, through many models processed under high throughput (for example scientific data processing), to processing of very large models (in many thousands of elements). We evaluate several mainstream model transformation technologies with respect to the various use cases, to provide recommendations for users...

CVL

> CVL is an upcoming standard for describing and implementing variability management in a model-driven development process. We implement the CVL prototype in order to demonstrate the feasibility of the standard, to uncover issues with expressiveness of the CVL language, to identify semantic unclarities and flaws.

The claims like above define what does it mean for our project to deliver *good* results. In the first case this is the usability as a benchmark and as a bridge between technological spaces. The second case, it is providing guidance for various use cases. In the third one it is debugging the CVL specification to provide feedback to its designers. Each requires a suitable evidence.

Evidence for The Examples

- **Traceability** — a tool maintaining traceability links + a controlled experiment to measure the density of model-code synchronization errors with and without the tool.
- **ecore2clafer** — a set of benchmarks; show that they are valid Clafer models (parse with Clafer's infrastructure). Integrate Clafer models into ecore infrastructure, for example, generate an ecore editor for a clafer model.
- **MT** — a benchmark extending over multiple use cases + statistically significant number of experiments. Ideally make sure that the data has realistic characteristics or is actually real (this makes your conclusion stronger).
- **CVL** — document the problems you had while implementing, including errors, unclarities, and limitations in modeling. argue for importance of a feasibility check for standards.

Simple rules of thumb:

- feasibility — it was not possible, it is possible now. Working implementation is often the core argument here. In such a project the motivation needs to be strong. What are the challenges? And why does it matter that we have such technology. Answering these questions usually leads to very strong claims that can be evaluated empirically to show that your implementation delivers on the promises.
- quantitative improvement — a quantitative quality that you measure against baseline of previous work. For example performance improvement is shown by benchmarking. Many other measures than performance are used in research: memory consumption, precision/recall, success rate, reliability, engineering effort/time, cost, ...

- qualitative improvement — often qualities can be pointed out in a discussion, although this is the weakest kind of evaluation. For instance in the Clafer paper we systematically discuss how Clafer meets its requirements, and why it does it better than other similar languages.

9.3 Scope of Evidence

It is essential to notice, that in all the above examples we are *not* discussing how to describe things. We are actually discussing how your objective, and the need for providing convincing evidence *shapes your work*.

You cannot first do the work, and then think how you will describe it. You should start planning the evidence and claims right from the beginning. Otherwise, in the end you will discover that you have done not the work you should have!

Often it happens that you discover that the ideal evidence for your claim is too difficult (or even infeasible) too obtain. Then the standard advice is to narrow your claims. For example:

Narrowing Scope Examples

- **Traceability**: if your tool can always prevent some kind of error, without human intervention, focus on proving this robustness. If this is only with high probability, run an automatic experiment to assess this probability. This way you avoid the expensive experiment with human subjects.
- **ecore2clafer**: if this is too much work, you drop one way of translation, and for example only focus on benchmarking not on integration.
- **MT**: you can limit the scope of your claim. Instead of stating that you will provide assessment across use cases, you can say our claim is that technology X is best suited for processing large quantities of small models.
- **CVL**: focus on feasibility only. Or focus on creating test models, instead of implementing transformations.

Here the challenge is to avoid limiting your claim too much, until it becomes uninteresting, irrelevant, or obvious.

Exercise. *Read one of the papers related to your project. Critically identify the claims made by this project, and compare them with the evidence provided. Is this evidence convincing ? What are its weak points ? Write this down in a paragraph of text. It will come handy when you will be writing related work for your paper. You may ask your supervisor to help with selecting a paper to read.*

Consider what could have been broader and narrower scopes for this paper?

9.4 Assorted Advice

Mary Shaw [22] has performed a survey of papers submitted in 2003 to ICSE, the most prestigious research conference on software engineering. Another source is a bit dated recommendation by Snyder available at: http://www.sigplan.org/oopsla/oopsla96/how91.html [23]. Let me summarize the main points that deal with providing evidence, and make the story of your paper stand against critique.

Your paper should present supporting evidence. Not just a conjecture [23]; *Solid evidence to support your result. It is not enough that your idea works for you, here also must be evidence that the idea will help someone else as well.* [22]

Specific advice from Mary Shaw:

- If it is supposed to work on large systems, then explain why you believe that it scales.
- If it is automatic, then explain exceptional, non automatic cases. Also cover the extent of customization/configuration needed to use the automatic technology. Try to assess how successfully automatic it is. Does it fail often ? Does the work required to use it is lower than the benefit obtained?
- If you propose a new metric: does it measure what it claims that it does ? Is it a better indicator than the other pre-existing metrics ? (for instance test coverage metrics)
- If you develop a formal model, then provide rigorous derivation and proof. Giving examples is not enough.
- She summarizes the following kinds of evidence: analytical, evaluation, experience, example, persuasion, blatant assertion. The last two are unsuitable for research papers. Example often is too weak. It is necessary, but not sufficient.

  Using a weak evidence in a conference submission, means q very swift rejection of your paper. In an exam situation, or thesis exam, this means weaker grade, since you are not convincing.
- If you claim to improve prior work, compare your result to prior work possibly objectively.
- If at all possible compare similar situations with and without your result/contribution.

I recommend studying carefully especially sections 4 and 5. In Section 5 she gives some historical data, on what kind of evidence goes with what kind of research. This should help you to verify your project.

Do not get fooled that she writes to PhD student. She writes to young research students, which in the American system means students in the first year after their bachelor.

As you have read the paper, note that it also meets structural requirements for a research paper itself. She does mention though a few weaknesses — it would not probably stand as a software engineering paper. Interesting reading!

10 Exercises

Objectives

- To perform a simple scope analysis by studying existing source code
- To design a simple DSL using meta-modeling
- To iteratively refine the meta-model through instance creation

I estimate that this exercise can be completed within 4 hours. Please read the entire description before starting to work on individual tasks.

We are going to design a simple domain specific language that in later exercises will be used to generate the assertion class of Junit. This is a small, and somewhat artifical exercise, given that the assertion class is small, and clearly not worth of replacing with generated code. The small size of the exercise comes handy though, as we want to use only limited time on this.

Task 1. Find your check out of the Junit code base from the previous exercises. The Assert class is found in `src/main/java/org.junit/Assert.java`. Open the file and read through it trying to understand the main contents.

We will be aiming at generating (almost) all methods that start with the "assert" prefix. Each of assert methods comes in two versions: with and without a message. Both versions can be generated from the same information, so we will ignore the methods without messages today. Focuse on the methods with the actual assertion.

Note that all of these methods have a similar structure: they have 1–3 parameters, and their body is essentially a call to a fail method depending on a simple expression over these parameters. Try to write down (on paper) this expression for each of the assert methods. This will give as a scope of what kind of expressions need to be supported in our DSL.

For example for

```
assertEquals(String msg, double expected, double actual, double delta)
```

You would write the following expression:

```
Math.abs(a-b) <= delta
```

Repeat this for all the public assert methods (ignore `assertThat` for simplicity)

Task 2. Now we aim at a language that looks similar to this:

```
Equals (double expected, double actual, double delta)
               asserts Math.abs(expected - actual) <= delta
```

Create a meta-model (in ecore) representing the abstract syntax for this language.

As an inspiration, my meta-model had approximately the following non-abstract classes: Assertion, Parameter, SimpleType, ArrayType, Expression, Identifier, Binary expression, unary expression, function call, Constant, Null. It took me about 30 minutes to create the first version and move to instance creation.

In your ecore model there should be a root class, say `Model`, that owns all the assertions in a model. Use containment (aggregation, black diamond) to bind elements to their parents in the syntax tree. Xtext, which we will use for code generation next week, requires that all model elements are owned directly or indirectly by the root model element (`Model` class here) via containment. A non-contained meta-class (just accessible via references) cannot be created in the standard ecore tree editor.

Task 3. Create instances of your meta-model representing the assert methods in the Assert.java file. Use the expressions you noted down in Task 1 to help you. Whenever you cannot express some instance the way you would like, please refine the meta-model.

Would be really cool, if you can create some new assertion methods that do not existing in Assert.java, but make sense. Try to model one as an instance of your meta-model.

Your meta-model is probably not final yet. We will test it and improve it further, when working on code generation next two classes.

Scoping notes. Some notes that I took, during the exercise myself. You may find them useful to scope the task down to a manageable size:

- JUnit has a kind of DSL for specifying assertions (implemented as the matchers API). It is used by `assertThat`. Let's ignore this fact for the purpose of our exercise, unless someone wants to reuse the design of matchers in her DSL. I also ignored the `assertThat` function in the meta-model design. I will not be generating it.
- Again: the DSL does not need to know about the distinction between message and non-message assertions. The code generator later on will be able to generate both from the same model instance.
- To avoid blowing up the number of meta-classes I modeled function calls using Strings (for function names). I did the same for binary and unary operators. If you want to model the operators kinds explicitly, I noted that down that these are the types used in the Assert.java file: `==, &&, !=, equals, <=, -`
- I assumed that the file header (imports, boilerplate), constructor, fail functions, format functions, and some array equality checking classes will be part of the static code included by code generator. They will not be described in the DSL.
- If the task gets out of hand, you may want to ignore all methods that involve arrays.
- Ignore deprecated methods.

Finally, a large part of the meta-model is actually a simple expression language. This could have now been borrowed from the XBase language for free, but I decided to model from scratch, to make the experience simpler.

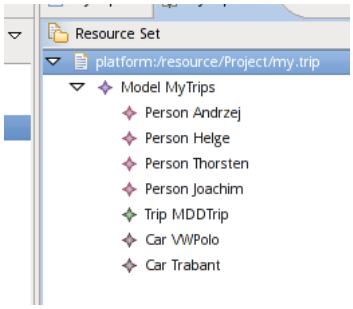
11 Textual Concrete Syntax, XText

Reading: We will talk about parsing, grammars, left-recursion, etc. If you are rusty on this then the Dragon book is your friend [1].

To get the idea of the Xtext framework you should read the first part (Getting Started) of the extensive documentation available at http://www.eclipse.org/Xtext/documentation/2.3.0/Documentation.pdf. In general familiarize yourself with http://www.eclipse.org/Xtext/, if your project involves implementing a textual DSL. Watch the videos, read the more advanced parts of the manual.

11.1 Introduction

The metamodel for a language can provide an abstract syntax. Domain experts and programs using this language will often need a readable concrete syntax, which can be manipulated in a text editor. Human readable syntax is sometimes also useful, even if models are not written by humans, but are created and processed completely automatically—human readable syntax is very helpful in debugging and monitoring in such cases.



You can think of the tree view editor, recalled in the figure above, as of an abstract syntax editor. While the diagram editor gets much closer to what a concrete syntax editor can do.

11.2 External DSLs with Xtext (Demo)

Xtext is the most popular language workbench for Eclipse. It is aims at languages with textual concrete syntax (as opposed to visual languages). We start with a simple demonstration: we will use Xtext to automatically derive concrete textual syntax for the abstract syntax of one of the examples used before in this course. The following EMF model describes the trip language, which can specify relations between trips, vehicles and passengers (persons). We used a similar example a few lectures ago:

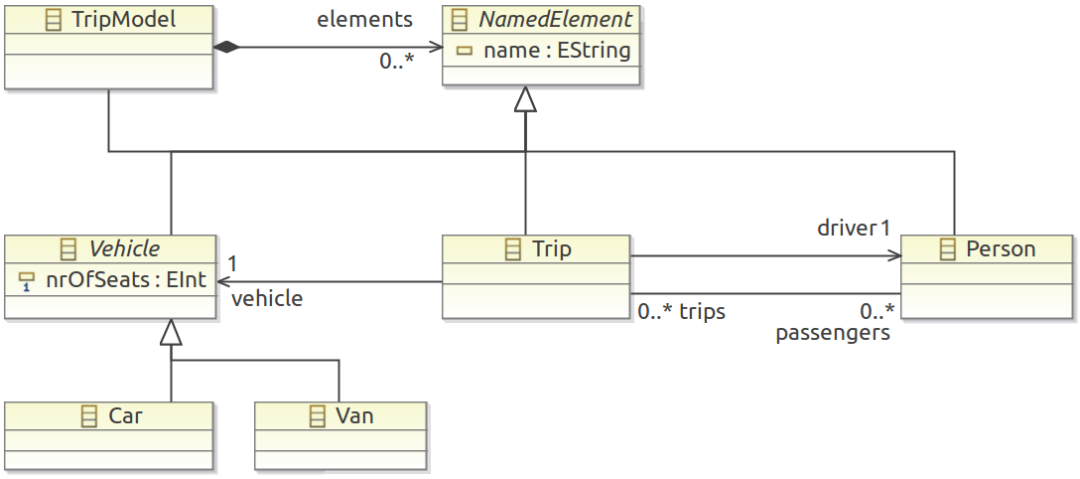


Figure 11.1: Abstract syntax (the meta-model) of the trip language

Before we proceed with generating a textual editor for this language, let me point out two properties of this model, which are necessary if you use Xtext. First, there is a root class, that represents the model (here this is TripModel). This class owns all the other elements through aggregation (containment). In this simple meta-model this ownership is direct, but it could also be indirect (via contained classes, that in turn contain other classes). Second, all important model elements are named. This often simplifies accessing them from various component frameworks. The name field is treated specially in the framework.

We follow these steps:

1. Create the .ecore file with the meta-model. Set the nsURI property of the toplevel package in the .ecore file to point to the location of the file in the Eclipse workspace (`platform:/resource/projectName/pat`
2. Create the default generator model for the trip.ecore file (File / New / Other ... / EMF Generator Model).
3. Genereate the model code from this new genmodel (context menu of the TripModel in .genmodel editor)
4. Now create an Xtext project using the *Xtext Project From Existing Ecore Models* wizard (File / New / Project ...). Choose trip.genmodel generated above when asked for the EPackage file in the wizard. Select TripModel as the root class (entry rule) in the same dialog.
5. A new project is created, and the default grammar specification for your language generated and opened in a text editor.
6. Generate Xtext artifacts (below). Approve downloading ANTLR.
7. Run the main xtext project as an Eclipse application.

The wizard generates four projects, out which the top one (without ui or test subpackage suffix) is of main interest. In the file trip.xtext you will find the syntax definition of our language (see Fig. 11.2).

Now, we generate the xtext artifacts (just a call from the *Run As...* context menu of the grammar editor). n a few seconds we can run the generated Eclipse application (plugin) and enjoy editing of

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

import "platform:/resource/episode10xtext/model/trip.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

TripModel returns TripModel:
  {TripModel}
  'TripModel' name=EString
  '{'
    ('elements' '{' elements+=NamedElement ("," elements+=NamedElement)* '}')?
  '}';

NamedElement returns NamedElement:
  Trip | Person | Car | TripModel | Van;

Vehicle returns Vehicle:
  Car | Van;

EString returns ecore::EString:
  STRING | ID;

Trip returns Trip:
  'Trip' name=EString
  '{'
    'vehicle' vehicle=[Vehicle|EString]
    ('passengers' '('passengers+=[Person|EString]("," passengers+=[Person|EString])*')')?
    'driver' driver=[Person|EString]
  '}';

Person returns Person:
  {Person}
  'Person' name=EString
  '{'
    ('trips' '(' trips+=[Trip|EString] ( "," trips+=[Trip|EString])* ')' )?
  '}';

Car returns Car:
  'Car' name=EString
  '{'
    'nrOfSeats' nrOfSeats=EInt
  '}';

Van returns Van:
  'Van' name=EString
  '{'
    'nrOfSeats' nrOfSeats=EInt
  '}';

EInt returns ecore::EInt:
  '-'? INT;
```

Figure 11.2: Automatically derived concrete syntax definition, based on the trip metamodel.

trips with text highlighting, code completion, name resolution and interactive error reporting. See Figure 11.3.

The trip model in Fig. 11.3 is incorrect: Helge is not declared as a person. This is why the tool reports this immediately by underling the reference in the driver entry. The problem is also listed in the *Problems* view in the bottom of the screen (not shown in the figure). Technically this means that Xtext

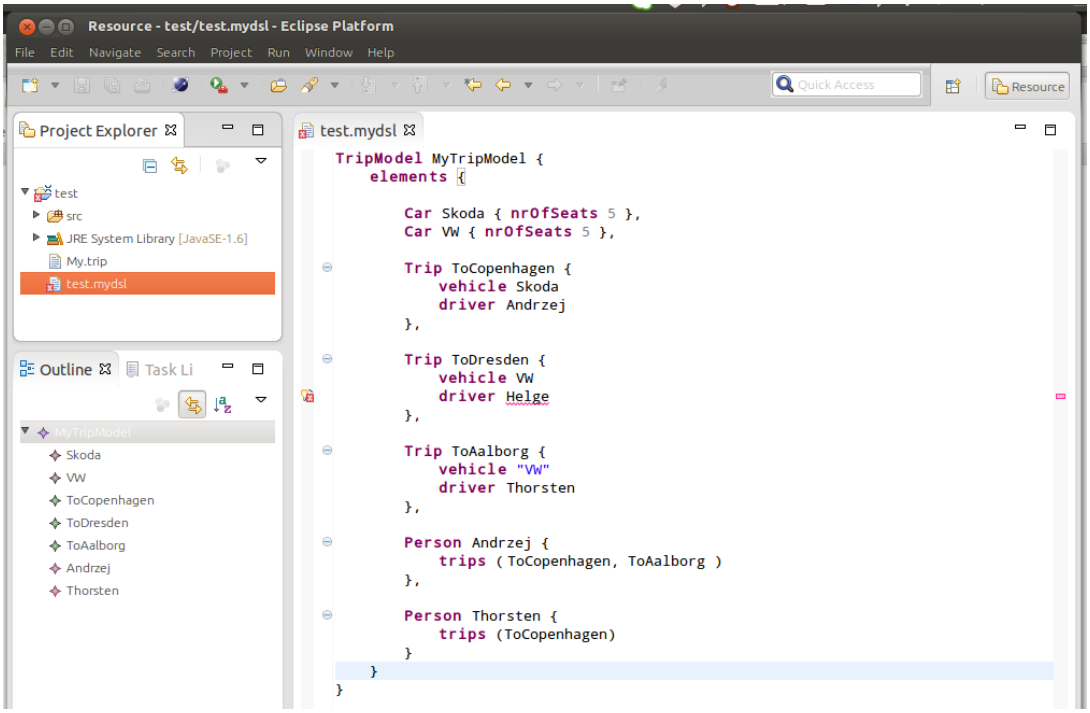


Figure 11.3: Generated default plugin for the trip language (in action)

not only parses the model, but also performs static validation of references. It also does type checking, and can be integrated with other validity constraints. The underlined token is actually reference to an undeclared Person object.

Also note that terminals of the above grammar specification have been directly turned into keywords in the editor.

Attentive reader could notice that there is another (integrity) rule that is violated by the example model in the trip editor. Person blocks list a few trips, but Trip blocks do not list the respective persons in the passenger list. This violates the constraint that the passengers reference is the inverse (*EOpposite*) of the trips reference in our meta-model. Indeed, the editor does not check these constraints presently, although suitable EMF mechanisms could be invoked (programmatically).

Let us finish this section with a quote that Xtext project uses to introduce itself:

> *A quote from the guide: Xtext provides you with a set of domain-specific languages and modern APIs to describe the different aspects of your programming language. Based on that information it gives you a full implementation of that language running on the JVM. The compiler components of your language are independent of Eclipse or OSGi and can be used in any Java environment. They include such things as the parser, the type-safe abstract syntax tree (AST), the serializer and code formatter, the scoping framework and the linking, compiler checks and static analysis aka validation and last but not least a code generator or interpreter. These runtime components integrate with and are based on the Eclipse Modeling Framework (EMF), which effectively allows you to use Xtext together with other EMF frameworks like for instance the Graphical Modeling Project GMF.*

> *In addition to this nice runtime architecture, you will get a full blown Eclipse-IDE specifically tailored for your language. It already provides great default functionality for all aspects and again comes with DSLs and APIs that allow to configure or change the most common things very easily.* (XText online documentation)

In the following section we will explain the syntax definition format of Xtext, so that you can define concrete syntax yourself. We will also rewrite the default grammar into a simpler, more human-friendly one. This will have the advantage that we will have two similar languages defined, which we will be able to use as an example for model transformations later in the lectures.

11.3 Concrete Syntax Definitions in Xtext

The XText specification language is a variation of the familiar *Extended Backus-Naur Form* (EBNF) notation for context free grammars, which is also used by most parser generators, so you remember it from your compiler course. Xtext relies on the ANTLR parser generator to produce parsers. ANTLR is a recursive descent parser so it can only handle LL(k) languages, where k is the size of lookahead. The parser is usually most efficient, if the k is small, preferably equal to one.

This restriction of ANTLR (and thus of Xtext) is not a serious one, as it is widely believed that all interesting programming languages are LL(k) languages, and can be parsed by such parsers. The only problem is that it sometimes takes some effort to put the grammar of the language in the right form. Here the main problem is to make sure that the grammar is not *left-recursive*. Recall that a grammar is left-recursive if it has a production in which the left hand side nonterminal appears as the first element of the right hand side (either directly or indirectly via another nonterminal). Left-recursive grammars are quite common, for example a typical arithmetic expression grammar would include a production similar to

$$expr \rightarrow expr\ binary\text{-}operator\ expr\ ,$$

which is left-recursive. Sometimes a serious rewrite is necessary to remove left-recursion. For example:

$$\begin{aligned} expr &\rightarrow term\ (+\ term)^* \\ term &\rightarrow factor \mid factor\ *\ factor \\ factor &\rightarrow ID \mid (\ expr\ ) \end{aligned}$$

The standard reference on parsing techniques, where you can also find left-recursion elimination is the so called *Dragon Book* by Aho and others [1], but almost any compiler construction text book would do.

We shall now discuss the main syntactic elements of the Xtext language. First the specification starts with the name of the grammar (in the same format as fully qualified Class name in Java):

```
grammar org.xtext.example.mydsl.MyDsl
```

In order to enable reuse in language definition, Xtext allows importing other grammars. In our example above, we have included the grammar that describes standard terminals in a typical programming language (the terminal `ID` used in Fig. 11.2 comes from this included grammar:

```
with org.eclipse.xtext.common.Terminals
```

Then we import the meta-models used as the abstract syntax:

```
import "platform:/resource/episode10xtext/model/trip.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

The first model imported is the model presented in Figure 14.2. We also import Ecore itself, in order to be able to use its types, for instance EString. Note that the elements of trip become available in the default name space (no *as* clause). So later in the grammar, types like Person are referred to directly. At the same time the Ecore types are prefixed by the name space `ecore`. Like in OCL, double colon is used as the name space prefix operator.

The first symbol (the left hand side of the first production) is the start symbol of the grammar. Here: `TripModel`. The `returns` construction allows to specify the type representing a nonterminal in the abstract syntax. The default type has the same name as the nonterminal, so most of the *returns* clauses in the generated grammar are redundant, but some are not.

Terminals are simply introduced as string literals. For example `'Car'` and the braces in the following rule:

```
Car: 'Car' name=EString '{' 'nrOfSeats' nrOfSeats=EInt '}';
```

A value resulting from parsing a nonterminal can be stored directly in a property of the current abstract syntax object. For example, the above production says that an object of type Car will be constructed upon its successful application. The name of the car (Car.name in Java) will be initialized with the value of a string directly following the first keyword. Similarly the number of seats will be initialized to an integer value following slightly later. In general a property of any time (including class types) can be assigned with an object constructed by invoked productions.

In the following rule we see how elements parsed can be used to populate collections.

```
Person returns Person:
  'Person' name=EString
  '{'
    ('trips' '(' trips+=[Trip|EString] ( "," trips+=[Trip|EString])* ')' )?
  '}';
```

If a property of an abstract syntax object is a collection, we can add a value to the collection (as opposed to replacing the collection) using the += operator instead of assignment. This happens for both vehicles and elements above. There is no null pointer error, since the collections are initialized to be empty upon object creation (by EMF).

The braced type name (`{TripModel}`) enforces creating an instance of a given type. This is useful if we are parsing a concept that is represented by an abstract type with several possible implementations. For example, parsing named elements, could be more concisely written without introducing nonterminals for Cars and Persons as:

```
NamedElement returns NamedElement:
      Trip
    | Car
    | TripModel
    | Van
    | {Person} 'Person' name=EString '{'
        ('trips' '(' trips+=[Trip|EString]
                  ( "," trips+=[Trip|EString])* ')' )?
'}';
```

The use of this construct in Fig. 11.2 is redundant (this is because this is automatically generated code, that needs to cater also for other complex meta-models).

Remaining syntax: alternatives (|), repetition (+,*), optionality (?) is familiar from most regular expression dialects, and the meaning is as expected.

A crucial ability of Xtext is resolving references.

The syntax for a cross-reference is `[TypeName|RuleCall]` where `RuleCall` defaults to `ID` if omitted. The parser only parses the name of the cross-referenced element using the ID rule and stores it internally. Later on, the linker establishes the cross-reference using the name, the defined cross-reference's type (Entity in this case) and the defined scoping rules. For example:

```
Person: 'Person' name=EString '{'
    ('trips' '(' trips+=[Trip|EString]
              ( "," trips+=[Trip|EString])* ')' )?
'}';
```

There is much more to Xtext, than I present, but this is sufficient to do simple languages. Among other elements, of the highest interest are probably customizable scoping semantics (what names are visible in what scopes), and fully qualified name support for references across name spaces/scopes. These are described in Xtext documentation.

To end this lecture we present another version of the grammar for the same language Trip that leads to a much simpler syntax:

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

import "platform:/resource/episode10xtext.trip/model/trip.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

TripModel: (elements+=NamedElement)*;


NamedElement returns NamedElement:
    Trip | Person | Car | Van;

EString returns ecore::EString:
    STRING | ID;

Trip: 'trip' name=EString
        'car' vehicle=[Vehicle|EString]
        ('passengers' passengers+=[Person|EString] ( "," passengers+=[Person|EString])*)?
        'driver' driver=[Person|EString];

Person: 'person' name=EString;
Car: 'car' name=EString 'with' nrOfSeats=EInt ('seats' | 'seat');
Van: 'van' name=EString 'with' nrOfSeats=EInt ('seats' | 'seat');

EInt returns ecore::EInt: '-'? INT;
```

It took me not more than 10 minutes to rewrite the default generated syntax specification to this one, including generating a new Xtext based editor for models. An example model in this syntax looks as follows:

```
person Andrzej
person Helge
person Thorsten
person Joachim

car VWPolo with 4 seats
car Trabant with 4 seats

trip MDDTrip
    car VWPolo
    passengers  Helge, Andrzej, Thorsten, Joachim
    driver Joachim
```

In fact this model violates the meta-model constraint, because with this parser it only stores the references from trips to passengers, and not the other way around (remember that we have an integrity constraint that makes the two references inverse of each other).

So loading this model into EMF infrastructure (depending on configuration) may cause validation errors. The idea is that the inverse reference should not be stored in the model, but can be recovered with a model transformation. We will use this as an example in later classes.

11.4 DSL Design Guidelines (Concrete Syntax)

It is known that 75% of population prefers visual to textual syntax. Here Xtext is not very useful [15]. However state of the art in applications of DSL is using them by engineers to write models *together*

with subject matter experts. Here textual DSLs are good enough. Use of DSLs by subject matters only is rare.

It is also known that programmers are more likely to prefer textual notations (to visual), so DSLs aimed at software developers should probably be textual. [15]

Choosing either representation purely on the basis of prejudice is bad, as is ignoring other possibilities such as matrices, tables, forms, or trees. The correct representational paradigm depends on the audience, the data's structure, and how users will work with the data. Making the wrong choice can significantly increase the cost of creating, reading, and maintaining the models. [15].

Whenever possible you should adopt existing notation sused by the intended audience of your language. It is known that we engineers are fast in (and attracted to) learning new languages. This is completely opposite with non-programmers. Introducing new notation is likely just another barrier to acceptance of technology you want to introduce (on top of the tool and process change) [14]. For the same reason avoid changing the meaning of known symbols (for example + should mean addition, and unlikely anything else). If in need for new symbols, descriptive terms (English words).

Always allow comments. These are very useful for evolving and experimenting with models [14].

It is very practical to keep the abstract and concrete syntax not too far apart. This simplifies your implementation (parsing, transformations, pretty printing). Element with different concrete syntax should have different abstract syntax. Also the concrete syntax should be context independent—concrete syntax of an element should be dependent on its abstract type, and not on the context of use [14].

Karsai and coauthors [14] give many other very practical guidance, thus I again encourage you to read their paper when designing your DSLs.

12 Exercises

Objectives

- To practice implementation and design of concrete textual syntax
- To practice grammar transformation to resolve ambiguity and remove left-recursion
- To learn using Xtext with EMF models
- To prepare test harness for the generated code

I estimate that this exercise can be completed within 10 hours. *Please read the entire description before starting to work on individual tasks.* I expect this to be one the most laborious exercise sheet this semester.

Task 1. (easy) Follow the guidelines from lecture on Xtext to implement the first textual syntax for your Assert language. Start with the meta-model developed last week.

Task 2. (easy) Test the created syntax by typing in 1-2 models representing the individual assertion methods.

Task 3. Start editing the generated grammar to improve readability and writeability of the syntax. Revise the syntax and regenerate the editor as many times as you need. This task can take a lot of time, depending on your proffiency with grammars, and/or ANTLR.

Just to inspire you, here is a fragment of my model in the my concrete syntax:

```
True (boolean x) := x;
False (boolean x) := !x;
Equals (Object expected, Object actual) :=
   (expected == null && actual == null) ||
   (expected != null && expected.equals (actual));
ArrayEquals(Object[] expecteds, Object[] actuals) :=
   internalArrayEquals(expecteds, actuals);
NotEquals (double first, double second, double delta) :=
   Math.abs(d1-d2) <= delta;
```

Task 4. Use your new editor to create an instance model representing assertion methods in **Assert.java**. This model will be the input for the code generator that we will develop last week, so make sure that it is sufficiently concrete.

I did this task iteratively alternating with 3. I used the process of creating the model (4), to improve the concrete syntax (3).

Task 5. Develop a Junit test harness for assertion methods in Assert.java. Make sure that you thoroughly test each method that you intend to replace by code generation. Make sure that all the tests pass on the original Assert class implementation. This task can take quite some time, but should be easy and familiar.

Some Hints on Xtext Issues If an exception is thrown that the ecore package cannot be loaded when you start up your new DSL editor, then check out the following:

- When generating Xtext artifacts, make sure that you open the console view, and confirm that ANTLR can be downloaded (the first time you do it).
- Have you reloaded / regenerated the generator model after changing the ecore model ?
- Have you regenerated the Xtext artifacts, after any updates to the grammar ? after updates to the ecore / genmodel?
- Is the platform URI correct in the ecore toplevel package properties set correctly? (see lecture notes, changes require reloading to genmodel)
- Check if the name of the physical directory on the disk is the same as the name of your project. Sometimes these names get out of sync, if you use version control and rename projects within Eclipse.
- Try erasing the `runtime-EclipseApplication` folder, where the workspace for spawned Eclipse is stored. Sometimes the meta-data in this workspace gets corrupted, when you change the loaded plugins frequently.
- If you are using git to manage your exercise files, you need to take extra care. Perhaps it is easier to avoid version control during exercises, to avoid any extra layer of complexity. Eclipse Git client stores the projects in a local git repository on your drive, which is usually located outside you workspace directory. If your EMF project is shared with git, the EMF generators will create the code and projects within the git repo directory (but not add them to git). However, if you create the xtext project, it will be located in the workspace directory, and not in git. In my experience it will work just fine, as long as you do not move it to the git repo (which is clearly needed in a real project!)

  I had some success in this direction by first generating the xtext project, and all xtext artifacts, and adding them to git repository afterwards. Then I changed the default value of `var runtimeProject` in the MWE2 script to point to the actual location of the main xtext project directory (the one containing the grammar), and regenarating again. It might help to uncomment the `registerGenModelFile` call in the StandaloneSetup section of the MWE2 file, too. Stale editors from previous debugging session, seem to have some adverse effect on loading packages, too. So you may want to close them, and restart the second eclipse instance with a clean desktop.

  In general, it appears that using xtext with git **is fragile**, so either use svn, or use a git set up that creates a git repository directly in your workspace, so that the workspace and the git repository are physically in the same location.

Helge reports reasonable success in using the command-line git client with eclipse, as opposed to using EGit – the built-in client. Might be worth trying, if you need version control (just version control your workspace directory calling git from the operating system level).

Hints on Language Implementation. Depending on the constructs used in your meta-model, you may need parts of Xtext specification language that were not presented in the lecture. I encourage you to simplify the meta-model, if this happens. In the end I only needed one construct that was not presented—the implementation of enum rules. Here is how keywords can be mapped to enum values in xtext:

```
enum SimpleTypeEnum:
   BOOLEAN='boolean' | OBJECT='Object' | OBJECT_ARRAY='Object[]' |
   DOUBLE="double" | LONG="long" | BYTE_ARRAY="byte[]" |
   SHORT = "short" | INT ="int" | FLOAT = "float";
```

The above construction introduces a production called SimpleTypeEnum. It returns each of the listed enum values (capitalized in the example) as a result of parsing the respective keyword.

NB. You can always avoid using enumerations like this one, by turning `SimpleTypeEnum` into an abstract class, and each of the enumeration values into its concrete attribute-less subclasses.

When you will be performing left-factoring of the grammar, you might hit another problem, that the type returned by a rule should change depending on how long expression is parsed. Here is the crux of an expression rule from my example:

```
Exp returns Exp:
    Term ( {BOp.lexpr=current} operator='+' rexpr=Exp)*;
```

In this rule `current` is a special keyword, denoting the object representing the syntax of what was parsed so far (so here the object returned by `Term`, or by previous application of the Kleene-star). So whenever a new instance of the Kleene-expression is matched a `BOp` object is created and the previously parsed value is stored in its **lexpr** attribute. If the Kleene-expression is not matched at all (zero occurrences) the object created by the `Term` rule is simply returned. Of course that rule must have a type consistent with `Exp` and `BOp` also must be a subclass of `Exp` in this example. See an entire tutorial on left-factorization with Xtext at http://blog.efftinge.de/2010/08/parsing-expressions-with-xtext.html.

When debugging your grammar it is convenient to examine the outline view (usually in the left-bottom corner of the workbench space). It continuously displays the AST of the concrete syntax in the main Xtext DSL editor. This way you can easily check if the operator precedence, left/right-biding, etc, are set up correctly in your grammar rules. Note that the default generated grammar does not take precedence of operators into account.

Finally, as per the guidelines discussed in the lecture, C-like comments (both block comments, and line comments) are automatically supported in our language, courtesy of Xtext.

13 Model-To-Text TX (M2T)

Reading: Czarnecki and Helsen [6] survey model transformation technologies. This is a useful, if dated, state of the union regarding model transformation. Most of the technologies mentioned are still available, and the catalog of main characteristics of these technologies has not changed much since then.

Other than that we rely on TMF documentation: the MWE2 tutorial and the Xtend tutorial.

13.1 Model Transformations: Applications & Classification

So far we were concerned with defining the syntax of DSLs in efficient ways, including tools for editing models. Now we are going to turn our attention to implementing DSLs, or to their semantics. The standard way to give semantics to programming (modeling) languages is writing an interpreter or a compiler. We will focus mostly on compilers, i.e. we will consider translating models to other languages.

Translating of models between languages is called *model transformation*. Model transformation is a form of meta-programming, so programming that treats other programs (models) as data. In that sense it is a more advanced programming activity than usual programming.

⚠ Applications of model transformation include [6]:

- **Generating** lower-level models and eventually code, from higher-level models
- **Mapping** and **synchronizing** among models at the same level or different levels of abstraction
- Creating a **query-based views** on a system
- Model **evolution** tasks such as model **refactoring**, model versioning, diffing and patching
- **Reverse engineering** of higher-level models from lower-level ones

From all these points the first one is most certainly most important, and of interest to general software development audience. The remaining ones are more internal MDD problems, that are addressed by vendors of MDD tools.

Overview of the model transformation set-up:

rogram transformation and model transformation are very close, as models and programs are sometimes hard to distinguish. The main objective of program transformation has always been compilation (and optimization of code). Program transformations are based on mathematically-oriented concepts such as term rewriting, attribute grammars, and functional programming. Model transformations usually target object-oriented programming and adopt an object oriented approach for representing and manipulating models. Model transformations, as they relate models to each other, are often expected to preserve traceability links, which is not done in program transformation [6].

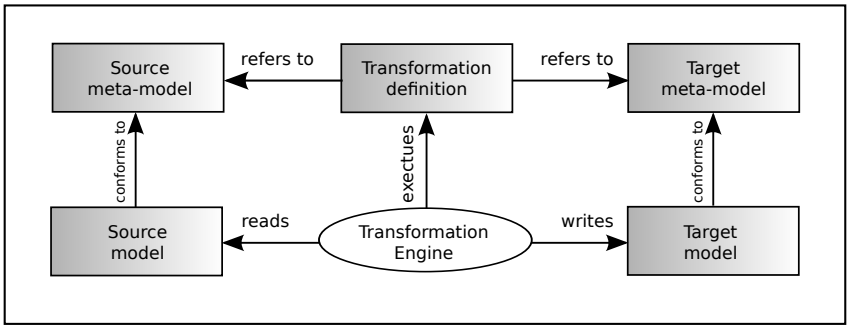


Figure 13.1: Overview of the model transformation languages and concepts. Figure from [6]

13.2 Model-To-Text Transformations (M2T)

A large class transformations translate models to text. The text can be generated code, other models in textual syntax, or other textual artefacts such as reports or documentation.

Text is typically generated using *templates*, a technique extremely well established also in web development. A template can be thought of as the target text with holes for variable parts. The holes contain metacode (so code creating code), which ir run at template instantiation time to compute the variable parts [6].

In the following we show to simple M2T transformations that generate respectively HTML and SQL code from models adhering to the trip language. In TMF templates are written using the textual templates of the Xtend language. Figure 13.2 shows the template generating a simple HTML report about our trips. Observe how concise is the template:

```
«IMPORT tripLoose»

«DEFINE main FOR TripModel»
«FILE this.name + ".html"»
<html><head><title>Report for Model "«name»"</title></head>
<body>
<h1>Trip Model Report for Model "«name»" </h1>
«FOREACH elements AS e»«EXPAND entry FOR e»«ENDFOREACH»
</body>
«ENDFILE»
«ENDDEFINE»

«DEFINE entry FOR NamedElement»
«ENDDEFINE»

«DEFINE entry FOR Car»
    <strong>Car</strong> «name»<br/>
«ENDDEFINE»
«DEFINE entry FOR Person»
    <strong>Person</strong> «name»<br/>
«ENDDEFINE»
«DEFINE entry FOR Trip»
    <strong>Trip</strong> «name»
    <ol>
    «FOREACH passengers AS p»<li>«EXPAND entry FOR p»«ENDFOREACH»</li>
    </ol>
«ENDDEFINE»
```

Figure 13.2: Xpand template generating a simple report about trips.

This template generates the following HTML (up to white space) given the same input model as used in the previous talk:

Output

```
<html><head><title>Report for Model "MyTrips"</title></head>
<body>
<h1>Trip Model Report for Model "MyTrips" </h1>

<strong>Car </strong> WVPolo<br/>
<strong>Car </strong> Trabant<br/>

<br/>

<strong>Person</strong> Andrzej<br/>
<strong>Person</strong> Helge<br/>
<strong>Person</strong> Thorsten<br/>
<strong>Person</strong> Joachim<br/>

<strong>Trip</strong> MDDTrip
<ol>
<li><strong>Person</strong> Helge<br/>
<li><strong>Person</strong> Andrzej<br/>
<li><strong>Person</strong> Thorsten<br/>
<li><strong>Person</strong> Joachim<br/>
</li>
</ol>
</body>
```

Let us briefly summarize the syntax of the template. The first line imports the meta-model of the language processed. Then we define the main template which is to be executed on elements of type TripModel. The object of this type can be referred to as `this` in the text of the transformation.

The body of the template is treated as a text file, which is verbatim output to the output file. Text in «French quotes»is the meta-code. Use code complete (ctrl-space) to switch to the meta-code mode. The `FILE` instruction opens a new file.

We do a polymorphic expansion for named elements. We bind the element processed to a variable *e* — so that we can access it in the contained `EXPAND` command. This command calls another expansion function. The call is polymorphic, so a different function will be called for objects of various concrete types. These polymorphic templates are given in the bottom of the figure. The transformation handles Cars, but silently ignores other vehicles.

The following template is very similar, but it generates an SQL code for inserting the trip data into a hypothetical relational database:

13.3 Gluing Things with MWE2

The transformations, model readers, model serializers, validators, and all other model processing components need to be wired together to perform the task you need. This can be done using a GPL

```
«IMPORT tripLoose»

«DEFINE insert FOR NamedElement»«ENDDEFINE»
«DEFINE insert FOR Person»
    INSERT INTO PERSONS VALUES ('«name»')
«ENDDEFINE»
«DEFINE insert FOR Vehicle»
    INSERT INTO CARS(regnr,seats) VALUES ('«name»','«nrOfSeats»')
«ENDDEFINE»
«DEFINE insert FOR Trip»
    INSERT INTO TRIPS (id,name) VALUES ('«name»','«name»')
    «FOREACH passengers AS p»
    INSERT INTO PASSENGERS (tripid,personid) VALUES ('«name»','«p.name»')
    «ENDFOREACH»
«ENDDEFINE»

«DEFINE main FOR TripModel»
«FILE this.name + ".sql"»
«FOREACH elements AS e»«EXPAND insert FOR e»«ENDFOREACH»
«ENDFILE»
«ENDDEFINE»
```

Figure 13.3: Xpand template to generate simple SQL.

like Java, but it is often easier in some kind of scripting language. Ant can be good for combining very coarse grained components, but since here individual components are Java classes, we need a language that is able to instantiate classes, initialize their parameters and invoke them in the right order. MWE2 is TMF's language for this purpose. It is a kind of *glue* language used to define Workflows.

MWE2 is not a overwhelmingly interesting language in itself, but you will need to know it when working with TMF. In the following I give a quick summary of MWE2, so that you can use it in your tools.

The main concept of MWE is the Workflow. Each script defines one workflow, which is like a model processing recipe, consisting of coarse grain steps.

Here is the simplest work flow (inspired by MWE2 official tutorial):

```
module episode13m2t.trip.generators.HelloWorld

Workflow {

    component = SayHello { message = "Hello!" }

}
```

The first line declares the name of the workflow module (and to which package it belongs). Then our workflow consists of creating, initializing and invoking exactly one component, called SayHello.

A component is a Java class implementing the IWorkflowComponent interface. So to run the above work flow, we need to provide an implementation of such Java class.

Here is one possibility:

```
package org.xtext.example.loosetrip.generators;
import
  org.eclipse.emf.mwe2.runtime.workflow.IWorkflowComponent;

public class SayHello implements IWorkflowComponent {

  private String message = "Hello World!";

  public void setMessage(String message)
  { this.message = message; }

  public String getMessage() { return message; }

  public void invoke(IWorkflowContext ctx)
  { System.out.println(getMessage()); }

  public void postInvoke() { }
  public void preInvoke() { }
}
```

Observe that the component implements the method invoke, and it provides an attribute message, with respective access functions. The workflow line activating this component roughly corresponds to:

```
SayHello h = new SayHello ();
h.setMessage (‘‘Hello!’’);
h.invoke(ctx);
```

where ctx is an object storing the state of execution of the workflow.

Now of course this is quite a heavy weight way to say 'Hello' to the world. The MWE2 starts to shine if you realize that the entire TMF infrastructure is based on Java. We will see that some classes can be (for example) implemented in Xtend, which is another programming language of TMF meant for transformation and model manipulation. Moreover, the framework provides many standard components, and some components are generated for your languages, thus you do not have to implement them in Java.

The following work flow reads a model in xtext syntax for trip and saves it in the xmi syntax. So this is already a kind of model transformation, implemented purely in MWE2, as it only relies on standard components:

A Larger Example: trip2xmi

```
module episode13m2t.trip.generators.M2T

Workflow {

bean = org.eclipse.emf.mwe.utils.StandaloneSetup {
platformUri = "../"
registerGeneratedEPackage = "tripLoose.TripLoosePackage"
registerEcoreFile = "platform:/resource/episode10xtext.tripLoose/model/tripLoose.ecore"
}
```

```
component = org.eclipse.xtext.mwe.Reader {
path = "input/"
register = episode10xtext.tripLoose.xtext.TTripLooseStandaloneSetup { }

load = {
slot = "tripmodel"
type = "TripModel"
firstOnly = true
}
}

component = org.eclipse.emf.mwe.utils.DirectoryCleaner {
directory = "../episode13m2t.trip/model-gen/"
}

component = org.eclipse.emf.mwe.utils.Writer {
modelSlot = "tripmodel"
uri = "platform:/resource/episode13m2t.trip/model-gen/output.xmi"
}
}
```

The first bean registers the metamodel for our language, and informs the state of the platformUri (the directory of current Eclipse's workspace). Then we call the standard Reader component. This reader reads all the models under the indicated path, and stores elements of type "TripModel" into a *slot* called "tripmodel". A slot is simply an index entry in a map from names to slots. The map of slots is the main component of the context, which is passed around to exchange information between components.

Then we use a directory cleaner, a standard component preparing space for generated artefacts. And finally we use a standard writer (with an xmi) extension, to save a file in the xmi format.

Note that in this workflow there is actually no transformation directly, but simply loading and saving the same model in a different representation.

Finally we show the workflow that is needed to call one of the two M2T transformations presented in the previous sections:

```
module org.xtext.example.loosetrip.generators.M2T

Workflow {

  bean = org.eclipse.emf.mwe.utils.StandaloneSetup {
    platformUri = "../"
    registerGeneratedEPackage = "dk.itu.example.tripLoose.TripLoosePackage"
    registerEcoreFile = "platform:/resource/dk.itu.example.trip/model/tripLoose.ecore"
  }

  component = org.eclipse.xtext.mwe.Reader {
    path = "../org.xtext.example.loosetrip/m2/txt/"
    register = org.xtext.example.loosetrip.LooseTripStandaloneSetup {}

    load = { slot = "tripmodel"
             type = "TripModel" }
```

```
  }

  component = org.eclipse.emf.mwe.utils.DirectoryCleaner {
    directory = "../org.xtext.example.loosetrip/model-gen/"
  }


  component = org.eclipse.xpand2.Generator {

    metaModel = org.eclipse.xtend.typesystem.emf.EmfMetaModel {
      metaModelPackage = "dk.itu.example.tripLoose.TripLoosePackage"
      metaModelFile = "platform:/resource/dk.itu.example.trip/model/tripLoose.ecore" }

    metaModel = org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel {}
      //expand = "templates::DoHtmlReport::main FOREACH tripmodel"
      expand = "templates::DoSQL::main FOREACH tripmodel"

      outlet = { path = "model-gen" }
  }

  component = org.eclipse.emf.mwe.utils.Writer {
    modelSlot = "tripmodel"
    // file extension is important here.
    // ".strip" (so the declared xtext syntax extension) saves in textual syntax
     // ".xmi" or ".ecore" saves in xmi syntax.
     uri = "platform:/resource/org.xtext.example.loosetrip/model-gen/capitalized.strip"
  }
}
```

The only really new thing in this workflow, is a call to the `org.eclipse.xpand2.Generator`, which executes the M2T transformation in xpand.

In Eclipse execute the MWE2 workflows by choosing “Run as” from their context menu.

14 Model-To-Model TX (M2M)

Reading: See http://www.eclipse.org/xtend.

14.1 Implementing M2M Tx in Java

One possible approach to implement M2M transformations would be implementing them in Java. Since our metamodels in EMF have Java implementations, all the persistence code is generated and readily available, we can simply load model data in memory objects, and manipulate them using java. We can also call Java components from MWE2.

Here is an example of an endo-transforation. It takes a model instance of the Trip model and capitalizes names of all named elements and all vehicles:

```
import java.util.List;

import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowContext;
import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowComponent;

import dk.itu.example.tripLoose.NamedElement;
import dk.itu.example.tripLoose.TripModel;

public class DoCapitalizeNames implements IWorkflowComponent {

    public void invoke(IWorkflowContext ctx) {

        @SuppressWarnings("unchecked")
        List<TripModel> tms = (List<TripModel>) ctx.get("model");
        TripModel tm = tms.get(0);

        for ( NamedElement ne : tm.getElements())
            ne.setName(ne.getName().toUpperCase());
    }

    public void postInvoke() {

    public void preInvoke() {

}
```

Figure 14.1: Capitalizing name attributes for an instance.

Observe that in the beginning we obtain the model from one of our slots (it must be placed there by some other workflow element). Then we simply iterate over objects implementing the ecore model, introducing the desired changes, just as if these were usual Java implementation objects.

This low-level approach quickly gets complicated. For example complex meta model graph give rise to complex, multiple passes over the models.

The transformation, as usual in TMF, is called from a work flow. Here is the work flow that I have written for this purpose.

```
module org.xtext.example.loosetrip.generators.M2M
Workflow {

  bean = org.eclipse.emf.mwe.utils.StandaloneSetup {
    platformUri = "../"
    registerGeneratedEPackage = "dk.itu.example.tripLoose.TripLoosePackage"
    registerGeneratedEPackage = "dk.itu.example.trip.TripPackage"
    registerEcoreFile = "platform:/resource/dk.itu.example.trip/model/tripLoose.ecore"
    registerEcoreFile = "platform:/resource/dk.itu.example.trip/model/trip.ecore"
  }

  component = org.eclipse.xtext.mwe.Reader {
    path = "../org.xtext.example.loosetrip/m2/txt/"
    register = org.xtext.example.loosetrip.LooseTripStandaloneSetup {}

    load = { slot = "model"
             type = "TripModel" }
  }

  component = org.eclipse.emf.mwe.utils.DirectoryCleaner
  {  directory = "../org.xtext.example.loosetrip/model-gen/" }

  component = DoCapitalizeNames { }


  //component = Loose2StrictNaive {}
  //component = Loose2Strict {}

  component = org.eclipse.emf.mwe.utils.Writer {
    modelSlot = "strictmodel"
    // file extension is important here.
    // ".trip" (so the declared xtext syntax extension) saves in textual syntax
    // ".xmi" or ".ecore" saves in xmi syntax.
    uri = "platform:/resource/org.xtext.example.loosetrip/model-gen/strict.xmi"
  }
}
```

The main differences from the workflow presented in the previous lecture is use of the M2M component (DoCapitalizeNames), instead of the xpand generator. Also in the initialization phase we load both metamodels, target and source. For this particular transformation (capitalization), one would be enough, but we need both for the later transformations in this note.

14.2 Xtend at a Glance

For us Xtend is a pragmatic language that happens to be used in TMF. So we use it. Itself it is a reasonable modern OO-language with some extra goodies to make design of DSLs and transformations easier. Here is the characterization from Xtend docs:

Xtend is a statically-typed programming language which is tightly integrated with and runs on the Java Virtual Machine. It has its roots in the Java programming language but improves on a couple of concepts:

- *Advanced Type Inference - You rarely need to write down type signatures*
- *Full support for Java Generics - Including all conformance and conversion rules*
- *Closures - concise syntax for anonymous function expressions*
- *Operator overloading - make your libraries even more expressive*
- *Powerful switch expressions - type based switching with implicit casts*
- *No statements - Everything is an expression*
- *Template expressions - with intelligent white space handling*
- *Extension methods*
- *property access syntax - shorthands for getter and setter access*
- *multiple dispatch aka polymorphic method invocation*
- *translates to Java not bytecode - understand what's going on and use your code for platforms such as Android or GWT*

It is not aiming at replacing Java all together. Therefore its library is a thin layer over the Java Development Kit (JDK) and interacts with Java exactly the same as it interacts with Xtend code. Also Java can call Xtend functions in a completely transparent way. And of course, it provides a modern Eclipse-based IDE closely integrated with the Java Development Tools (JDT).

```
package org.xtext.example.loosetrip.xtend

import junit.framework.Assert
import junit.framework.TestCase

import static extension java.util.Collections.*

class Hello extends TestCase {

    def testHelloWorld() {
        Assert::assertEquals('Hello Joe!', sayHelloTo('Joe'))
        println('Hello Joe!')
    }

    def sayHelloTo(String to) {
        Hello::singletonList(this)
        new Hello().singletonList()
        "Hello "+to+"!"
    }
}
```

Package declaration and imports are essentially like in Java. Modulo the the lack of semicolons. And so is the class signature.

The static import makes all static methods of Collections available locally (see the second line of sayHelloTo).

Type inference, like in functional programming languages — return types for functions in the example are inferred (void and string). Everything is an expression. The value of the last expression is returned. So the return statement is also implicit.

Note that we extend a Java class seamlessly (the TestCase class from junit).

Again double colon is used for navigation over types.

This code is translated to the following Java code automatically (the incremental project builder takes care of that):

```
package org.xtext.example.loosetrip.xtend;

import java.util.Collections;

@SuppressWarnings("all")
public class Hello extends TestCase {

  public String testHelloWorld() {
    String _xblockexpression = null;
    {
      String _sayHelloTo = this.sayHelloTo("Joe");
      Assert.assertEquals("Hello Joe!", _sayHelloTo);
      String _println = InputOutput.<String>println("Hello Joe!");
      _xblockexpression = (_println);
    }
    return _xblockexpression;
  }

  public String sayHelloTo(final String to) {
    String _xblockexpression = null;
    {
      Collections.<org.xtext.example.loosetrip.xtend.Hello>singletonList(this);
      org.xtext.example.loosetrip.xtend.Hello _hello = new org.xtext.example.loosetrip.xtend.Hello();
      Collections.<org.xtext.example.loosetrip.xtend.Hello>singletonList(_hello);
      String _operator_plus = StringExtensions.operator_plus("Hello ", to);
      String _operator_plus_1 = StringExtensions.operator_plus(_operator_plus, "!");
      _xblockexpression = (_operator_plus_1);
    }
    return _xblockexpression;
  }
}
```

Note the inferred types. Most importantly note that this class becomes a java class that can be used from java, or from anything else, that expects a java class, seamlessly. In this example you can run it with the junit test case execution mechanism of Eclipse.

You can find the code in the xtend-gen directory. It is occasionally useful to look in there, if you get some weird type errors. Most often you will be able to understand them at the Java level.

Default class visibility is public, and fields are always private.

You do not need to declare rethrown exceptions. This is done automatically.

Local variables are introduced with the `val` keyword. Examples in the next section.

We will demonstrate the other features of the language trying to implement a simple M2M transformation.

14.3 Implementing M2M Tx in Xtend

This is the metamodel of our trip language.

We have created a similar metamodel, tripLoose, which only differs from the above, by not requiring that the passengers and trips are opposite references.

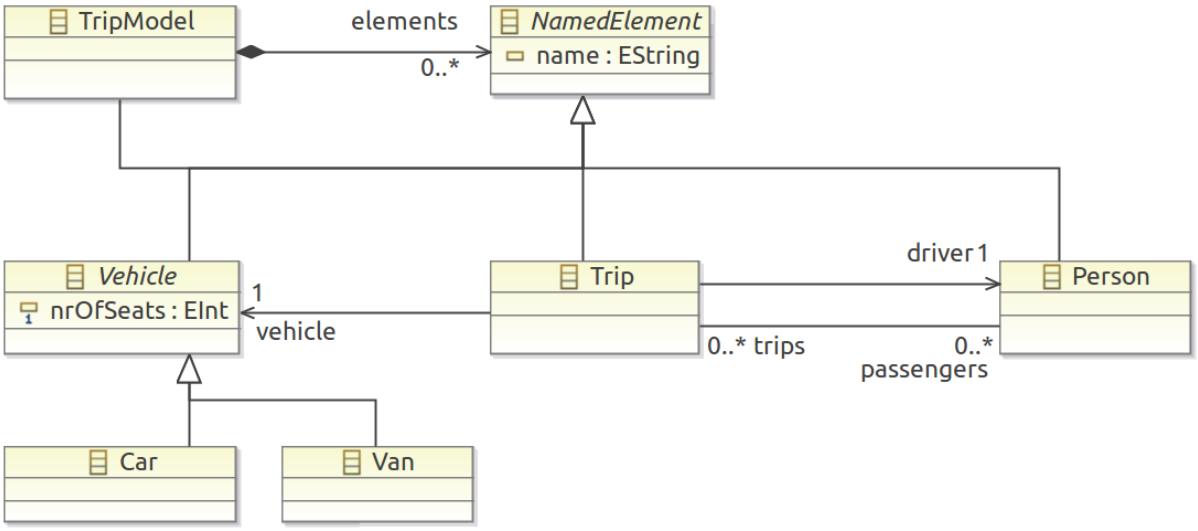


Figure 14.2: Abstract syntax (the meta-model) of the trip language

We want to write a transformation which reads in a tripLoose instance, and turns it into a trip instance. This is very simple: every object of type `tripLoose.T`, needs to be copied into an object of type `trip.T`. On top of that we need to populate the reverse reference Person.trips (in principle we should also populate Trip.passengers but we skip that to keep the code simpler).

Here is the naive way of implementing this in xtend:

```
package org.xtext.example.loosetrip.generators
import dk.itu.example.trip.*
import dk.itu.example.trip.impl.TripFactoryImpl
import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowContext


class Loose2StrictNaive extends WorkflowComponentWithSlot {

  java.util.List<Object> models
  dk.itu.example.tripLoose.TripModel model
  TripModel strictModel
  TripFactory strictFactory

  def dispatch void addElement (TripModel m, dk.itu.example.tripLoose.Person p) {
    val sp = strictFactory.createPerson();
    sp.name = p.name
    m.elements.add(sp)
  }

  def dispatch void addElement (TripModel m, dk.itu.example.tripLoose.Trip t) {
    val st = strictFactory.createTrip ();
    st.name = t.name
    m.elements.add(st)
  }

  (...)
```

```
  def dispatch Boolean namedTrip (Trip t, String n) { t.name == n }
  def dispatch Boolean namedTrip (Person t, String n) { false }
  def dispatch Boolean namedPerson (Trip t, String n) { false }
  def dispatch Boolean namedPerson (Person p, String n) { p.name == n }


  def dispatch void addAttr (TripModel m, dk.itu.example.tripLoose.Person p) {}
  def dispatch void addAttr (TripModel m, dk.itu.example.tripLoose.Trip t) {
    val Trip st = m.elements.findFirst(e | namedTrip(e,t.name)) as Trip
    for (p : t.passengers) {
      val String pname = p.name // clunky compiler it seems ...
      val Person sp = m.elements.findFirst (e | namedPerson(e,pname)) as Person
      st.passengers.add(sp) }
  }
  override void invoke(IWorkflowContext ctx) {
    strictFactory = new TripFactoryImpl ()
    strictModel = strictFactory.createTripModel ()
    models = (ctx.get ("model")) as java.util.List
    model = models.get(0) as dk.itu.example.tripLoose.TripModel
    strictModel.name = model.name

    for (e: model.elements) addElement (strictModel, e)
    for (e: model.elements) addAttr (strictModel, e)
    ctx.put ("strictmodel", strictModel)
  }
}
```

A *dispatch* is a special kind of method that is bound based on the actual type of the objects in actual parameters at runtime. So a dispatch allows you to write pattern matching on types, like in functional programming languages. A few simple examples are found in the above code.

Also note the use of anonymous functions (closures), which have a syntax akin to comprehended set expressions, making it possible to write set iterations in style very close to OCL.

Start reading the code with the invoke method (which will be called by the same kind of work flow as shown for capitalization example).

The code first obtains a factory for creating trip objects, and creates the root instance — TripModel. Then it obtains the input model from the context. We can, in principle, get a list of models, but for simplicity of the example we will only transform the first one found. This is why we only look at the first element of the list.

Then we do the work in two passes: first, we recreate all objects (named elements). In the second pass we create links between this objects using the addAttr function. This seems necessary, because we cannot create the links, before all objects are instantiated in the trip language (some links would have to be dangling otherwise).

Since M2M transformations are often this kind of graph rewriting, xtend offers a feature (create methods) that uses implicit memoisation, so that you can create your graphs in one pass.

A create function is a function that means to create an object, but it does this in two phases (each one represented by one body) . It first creates the object (the first body is a factory). Then it initializes the object (second body). If the object is created the first time it is cached (the ids of parameters being the key).

If in the initialization phase the object is created recursively again, a reference is simply returned from the cache. This allows to avoid the messy restoration of links, if you build you graphs in some kind of depth-first-search manner.

Here is an example of one such method:

```
def Trip create st: strictFactory.createTrip()
  copyElement(dk.itu.example.tripLoose.Trip t) {
    st.name = t.name
    st.vehicle = copyVehicle (t.vehicle)
    st.driver = copyElement (t.driver)
    st.passengers.addAll(
       t.passengers.map(p | copyElement (p)))
}
```

The important element is that when we get to the creation of the vehicle (or the driver), we can safely create the object since copyVehicle (and copyElement) are create methods, thus they will create the object only once. If the object is to be created again later (because we will see its declaration later, only after the trip declaration), we will simply operate on the copy created before.

igure 14.3 shows the complete xtend code of the more concise version of this M2M transformation.

Note the syntax of type casting, using the `as` keyword, unlike in Java.

This code is to be called with the same work flow as the two previous transformation (just change the transformation component in it).

14.4 Odds and Ends

While Xtend seems to be still a fairly low level language to implement M2M transformations in, there are opinions on the market that this is the most practical one. It strikes the balance between usability (effectiveness) and efficiency. There are rumors that tools based on graph transformation are much less efficient.

```
package org.xtext.example.loosetrip.generators
import dk.itu.example.trip.*
import dk.itu.example.trip.impl.TripFactoryImpl
import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowContext

class Loose2Strict extends WorkflowComponentWithSlot {

    TripFactory strictFactory
    // there should me a more concise way of testing adherance to an interface
    // I could not figure out, though.
    def dispatch Boolean isTrip (dk.itu.example.tripLoose.Trip t) {true}
    def dispatch Boolean isTrip (dk.itu.example.tripLoose.Person t) {false}

    def Person create sp: strictFactory.createPerson()
        copyElement(dk.itu.example.tripLoose.Person p) { sp.name = p.name }

    def Vehicle create sv: strictFactory.createCar()
        copyVehicle(dk.itu.example.tripLoose.Vehicle v) {
            sv.name = v.name
            sv.nr0fSeats = v.nr0fSeats
        }

    def Trip create st: strictFactory.createTrip()
         copyElement(dk.itu.example.tripLoose.Trip t) {
            st.name = t.name
            st.vehicle = copyVehicle (t.vehicle)
            st.driver = copyElement (t.driver)
            st.passengers.addAll(t.passengers.map(p | copyElement (p)))
        }

    override void invoke(IWorkflowContext ctx) {
        strictFactory = new TripFactoryImpl ()
        val strictModel = strictFactory.createTripModel ()
        val models = (ctx.get ("model")) as java.util.List<Object>
        val model = models.get(0) as dk.itu.example.tripLoose.TripModel
        strictModel.name = model.name

        for(v: model.vehicles) { strictModel.vehicles.add (copyVehicle(v)) }
        for(e: model.elements.filter (e | isTrip(e)))
            { strictModel.elements.add (copyElement(e as dk.itu.example.tripLoose.Trip)) }
        for(e: model.elements.filter (e | !isTrip(e)))
            { strictModel.elements.add (copyElement(e as dk.itu.example.tripLoose.Person)) }
        ctx.put ("strictmodel", strictModel)
    }
}
```

Figure 14.3: Abstract syntax (the meta-model) of the trip language

15 Exercises (Model Transformations)

Objectives

- To train MWE workflow implementation (minor)
- To practice programming in Xtend
- To train code generation using M2T, experiencing the core scenario of MDD.
- To train M2M transformation

We will use two exercise sessions (plus some hours of self-study) to complete this exercise sheet, which is also the last one in the mini-course. Read the entire exercise sheet before setting off to solve the tasks.

Task 1. Implement an MWE2 workflow that reads in a model in xtext syntax (using an Xtext reader), and saves it in the xmi (ecore) syntax (using the EMF Writer). Then implement an MWE workflow that reads a model in EMF/XMI format, and saves it in the Xtext format.

Hint: The xtext2xmi reader was actually presented in the lecture, so just make sure that it works in your context, and implement the other one.

Task 2. Implement the SayHello workflow component from Episode 13 using Xtend instead of Java.

Task 3. Implement the name capitalization transformation from Episode 14 using Xtend instead of Java

Task 4. Implement a code generator for the Assert language. First, establish which parts of the class will not be generated and factor them out to a super class, say AssertFramework. Once the class compiles, move on to the next task.

Task 5. Write a code generator that generates a subclass, called Assert, of AssertFramework, given as an input an instance of your Assert DSL. Use the instance developed last week, to generate the entire functionality of the class, and test it for correctness using the test harness developed last week.

Task 6. To train M2M transformations implement a converter, in Xtend, that reads instances of the Assert language of another group (or mine) and produces instances of your Assert language. Ecore and Xtext instances for my language are available in LearnIt.

Hints:

- This tasks requires that you also write a suitable MWE2 workflow that loads and saves the instances in the right syntax.
- The M2M transformation is completely Xtext independent. You only need to deal with abstract syntax, so the only dependency projects should be the EMF projects. Do not read or save Xtext syntax (then you involve text in the transformation, which makes it much more complicated).
- Save each ecore model (your Assert, the other Assert language) in seperate eclipse projects. The default code generators get confused if you keep more than one model in the same project.

Task 7. Write an MWE2 workflow that combines your M2M transformation with your M2T transformation, so that you obtain a code generation from another Assert language to Java using your code generation.

16 Another Take on M2M TX

Reading: We continue to rely on [6], already used in the previous lecture.

The QVT is specification is available at http://www.omg.org/spec/QVT/. If you want to learn more about ATL, the language guide is available at http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language

16.1 A Glimpse at QVT Relations

Remember, that our interest is in transforming models. In the past lecture, we have discussed how models can be translated to models in other languages using an imperative programming language Xtend. Today we look at rule based alternatives.

The cited survey lists multiple approaches to this problem. In particular it seems appealing to specify such transformation as rewrite rules, that match patterns in graphs and transform them into subgraphs adhering to another metamodel. This gives quite a high level mode of operation. Let us get a flavour of such an approach by looking at the following example of a *QVT relations* transformation.

The following example presents a simple meta-model of a class diagramming language, and a simple metamodel of relational database schema:

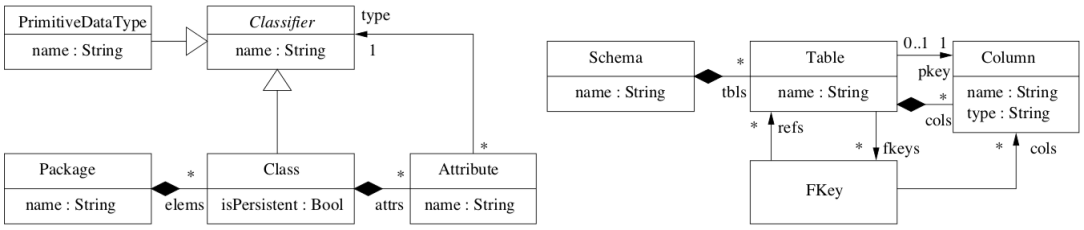


Figure 16.1: Figure from [6]

Our objective is to be able to translate between relational schema and class diagrams, in either direction. Our transformation would be able to create a proper relational schema for persistence of instances of our class diagram, and vice-verse: a proper class diagram describing structure of data that can be recovered from a give relational database. This can be specified by the following QVT Relations transformation [6]:

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {

  key Table (name, schema)
  key Column (name, table)

  top relation PackageToSchema {
    domain uml p:Package {name = pn}
    domain rdbms s:Schema {name = pn}
  }
```

```
top relation ClassToTable {
  domain uml c:Class {
    package = p:Package {},
    isPersistent = true,
    name = cn
  }
  domain rdbms t:Table {
    schema = s:Schema {},
    name = cn,
    cols = cl:Column {
      name = cn,
      cols = cl:column {
        name = cn + '_tid',
        type = 'NUMBER'},
      pkey = cl
  }
  when {
    PackageToSchema (p,s);
  }
  where {
    AttributeToColumn(c,t);
  }
}
...
```

Mappings are represented as relations, each having one domain declaration per domain involved in the transformation. Free variables in patterns are bound to values occurring in instances of the domain metamodels. So, for instance, in the first rule above 'pn' is a free identifier, and the first rule for Packages and Schemas, means that they will have the same name.

QVT Relations specifications are executable in either direction — the direction of execution is specified at runtime. Clearly, not all transformations can be implemented with QVT Relations, as not all transformations are reversible.

The QVT standard contains a few other languages (including an imperative language). The implementation of QVT within Eclipse is underway (http://wiki.eclipse.org/M2M/QVT_Declarative_(QVTd)).

16.2 Rule-based M2M Transformation with ATL

ATL is a M2M transformation language originally developed as a response to the OMG Request for Proposals for the QVT standard. It is now one of the popular transformation languages, with a free implementation compatible with Eclipse's modeling tools. If you followed our standard procedure for installing Eclipse, then you already have ATL tooling installed.

ATL supports both declarative and imperative style of transformations. We have largely covered the imperative style, when talking about xtend. This part of the lecture aims at demonstrating the rule-based style.

Figure 16.2 shows an example of an ATL transformation capitalizing all names in an instance of the tripLoose metamodel (Sec. 16.3). If you recall, we have implemented this transformation in Java in one of the earlier lectures. The Java transformation was an endo-transformation, so it modified the input model by raising the case of all names. ATL also allows making endo-transformations, which it calls refinements. We decided to show a copying transformation, which is a bit longer, but this way we can demonstrate the ATL rules more clearly.

```
1  module DoCapitalizeNames;
2  create OUT: tripLoose from IN: tripLoose;
3
4  abstract rule CapitalizeName {
5      from s :tripLoose!NamedElement
6      to t :tripLoose!NamedElement (name <- s.name.toUpper() )
7  }
8
9  rule CapitalizePerson extends CapitalizeName {
10     from s: tripLoose!Person to t: tripLoose!Person
11 }
12
13
14 rule CapitalizeCar extends CapitalizeName {
15     from s: tripLoose!Car to t: tripLoose!Car
16 }
17
18 rule CapitalizeTrip extends CapitalizeName {
19     from s: tripLoose!Trip
20     to t: tripLoose!Trip (
21             driver <- s.driver,
22             passengers <- s.passengers,
23             vehicle <- s.vehicle
24         )
25 }
26
27 rule CaptializeTripModel extends CapitalizeName {
28     from s: tripLoose!TripModel
29     to t: tripLoose!TripModel ( elements <- s.elements )
30 }
```

Figure 16.2: Capitalize all names in ATL.

The header of the transformation names the module, and establishes the source and target meta-models (the precise URI's to meta-model ecore files are established in the run configuration of the transformation, or in the workflow, if you incorporate it into a workflow). In this case the input and output metamodels are the same: we are going to copy a tripLoose instance, and while we do the copying, we will change all names to uppercase.

Then we have five transformation rules. The first rule is abstract: it applies to abstract classes, and it is extended by the other four rules. This way the actual capitalization is only written once. All the other rules simply do the copying of the input model elements.

Each rule has a **from** binding and a **to** section. In parentheses of the latter, attributes of the target model elements are created. The leftarrow symbol denotes assignments, and the right hand side expressions are written in OCL (so OCL is the expression language of ATL, which reflects the fact that ATL was developed as a proposal of an OMG standard).

Note that in comparison with Java and Xtend, one does not need to write any scheduling code, or any model traversal code — the rules are applied to all instances of types that satisfy the application conditions.

The other example (see Fig. 16.4) shows a transformation that converts a tripLoose model into a trip model. If you recall, this requires restoring the backwards reference from Persons to Trips, which lists the trips in which a Person participates. We have implemented such a transformation in Xtend in the previous lecture.

The transformation uses a helper function that given a person computes a sequence (a set) of all trips, which contain this person in the passenger list. The body of the function is just an OCL expression, that uses the select collection iterator. This function is called in the Person2Person rule.

The remaining rules just copy Cars and Trips (one more rules for TripModel is omitted for brevity).

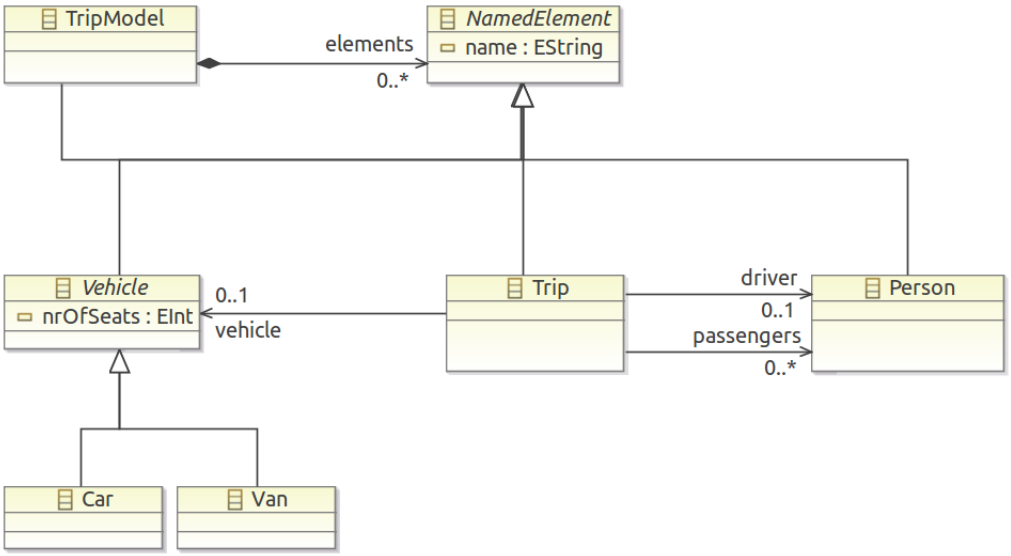


Figure 16.3: The tripLoose meta-model from previous lectures. It is identical to Trip, but does not contain the backwards reference from Persons to Trips.

Observe the apparent type mismatch in object creation rules. For example in line 35

```
vehicle <- s.vehicle
```

assigns an object of type `tripLoose!Vehicle` to an attribute of type `trip!Vehicle`. This is not an error. The ATL execution language calls the transformation on the Vehicle (here the Car2Car rule) to convert the object first, before the reference is assigned.

OCL also offers so called unique lazy rules, that avoid multiple creation of objects, even if called multiple times (they correspond to xtend's create methods).

We mentioned before that large collections of meta-models exists (so called meta-model Zoo) freely accessible online. ATL boasts a nice free collection of transformations at: http://www.eclipse.org/m2m/atl/atlTransformations/. It contains many interesting transformations. For example a converter from MOF models to proper UML class diagrams, extractors of information from Excel files, mapping from MySQL schema to meta-models, computations of metrics, make 2 ant, and many others ...

16.3 Example Exam Questions

⚠ What are the main characteristics of the ATL language? What are the advantages of writing transformations in a language like ATL over Java or Xtend? Explain a simple transformation provided to you on paper.

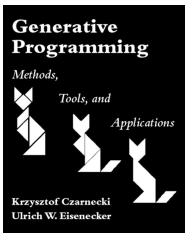
Of course, if you used ATL in your project, you will be asked more intricate questions about it.

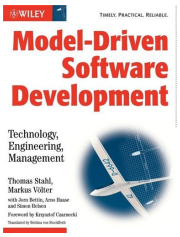
```
module tripLoose2trip;
create OUT: trip from IN: tripLoose;

helper def : tripsReferringTo(p :tripLoose!Person) : Sequence(tripLoose!Trip) =
    tripLoose!Trip.allInstances()->select(t | t.passengers->includes(p) );

rule Person2Person {
    from
        s: tripLoose!Person
    to
        t: trip!Person (
            name <- s.name,
            trips <- thisModule.tripsReferringTo (s)
        )
}

rule Car2Car {
    from
        s: tripLoose!Car
    to
        t: trip!Car (
            name <- s.name,
            nrOfSeats <- s.nrOfSeats
        )
}

rule Trip2Trip {
    from
        s: tripLoose!Trip
    to
        t: trip!Trip (
            name <- s.name,
            driver <- s.driver,
            passengers <- s.passengers,
            vehicle <- s.vehicle
        )
}
```

Figure 16.4: Convert tripLoose instances to trip instances using ATL.

17 Software Product Lines

Reading: This material is based mostly on chapter 2 of [5] and chapters 7–8 of [24]. The latter also covers product line engineering in chapter 13, and shows an extensive case study in chapter 16. In chapter 18 cost and benefit of MDSD is discussed in economical terms.





This week we will be looking at a particular application of model-driven development, *software product lines*—architectures that aim at maximizing reuse of code and other artifacts and labour in producing famillies of related systems.

The observation leading to creating a software product line is that the *opportunistic reuse does not work*. In a typical opportunistic reuse scenario developers clone (copy) a piece of code that implements a given functionality. This gives very fast and easy reuse, but unfortunately leads to multiplication of maintanance efforts. The cloned code starts to live its own life. If you fix it, you usually do this without fixing the original source. Also if the original is fixed, it is difficult to systematically propagate corrections. Testing effort is not reused for the two copies as well.

Over time the shared code in a product system decreases, and te product-specific code is growing. The entire system family becomes too costly to maintain. Similar effects appear, when the clone and own appraoches are organized using version control systems and branching. Version control is not well suited to organize many variants of software in parallel (parallel versioning). It is much better suited to organize sequential variants (history).

17.1 Software Product Lines

The following quotes a definition of a Software Product Line (SPL) originating from one of the institutes that strongly promote this methodology:

⚠

> A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. [http://www.sei.cmu.edu/productlines/, seen on 2011-09-09]

Software product line is a concept in which technical, business, and management issues overlap. Organizing your software production into SPL, is usually linked with an intention of addressing a certain well scoped market niche, by providing well customizable software for this niche/target group. The production of this software should rely on systematic reuse.

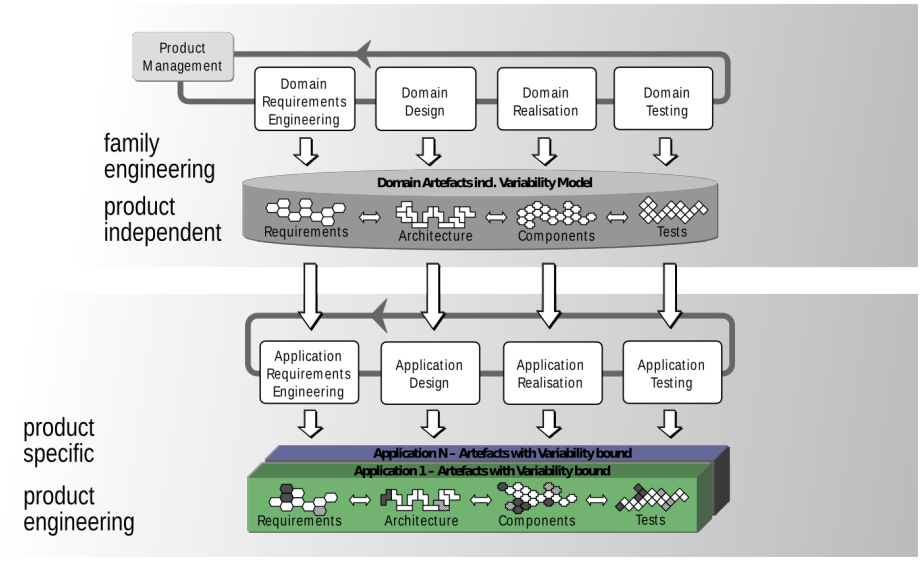


Figure 17.1: Two main process of product line development: Domain and Application Engineering. Figure adapted from [20]

Figure 17.1 summarizes the two main processes of product line engineering. The idea is to separate development of shared artifacts (the core, the platform, the family) from product-specific development. Both are full blown classic engineering processes.

⚠ Family engineering is the process that systematizes and collects knowledge, experience and assets accumulated in an organization (or in a software project) about a given domain, in order to provide means to reuse these efficiently when building new systems. The product engineering (bottom) derives the artifacts from the common domain artifacts produced in domain engineering (the top process). So design is done by completion of the shared design, application development is done by completing/deriving from the framework code. Test cases and documentation might be derived, too. By instituting this process systematically the cost of obtaining a single product is lowered.

It is important to note that in Fig. 17.1 the vertical arrows (derivation of applications from domain artifacts) are obtained using technologies presented in previous lectures.

In [24, chapter 18] there is indication that it often makes economical sense to consider an MDSD based product line architecture (PLA) if you can save about 30% of the code to be maintained. They indicate also that the cost of deriving a new variant is about 20-25% of making the reference implementation (one variant), and the total saving per early variant is conservatively estimated at 16%. This gives you some indication, when does it make economical sense to consider MDSD (sometimes people talk about a break even point at 3 products, for complex systems).

17.2 Domain Modeling Spectrum

In this course, we are primarily interested in technical support for SPLs. Here model-driven development appears very helpful. The idea is to build a domain model of the family that describes the differences and similarities between systems, and then link this model to the implementation either via code generation (generating individual products), or other means (annotations, preprocessing, interpretation, etc).

The first step in that process is building a domain model that describes *commonality* and *variability* of the family of products.

⚠

Commonality is all the aspects which are shared by the products in a SPL, while variability is all the aspects in which they differ. In Software Product Line Engineering (SPLE) one talks about *exploiting* commonality and *managing* (limiting, scoping) variability in order to obtain faster time to market, and better return of investment.

The first step in establishing a domain model is domain scoping. We distinguish two kinds of domains:

- vertical domains: are areas which are organized around classess of systems realizing specific business needs" For example "airline reservation systems, order processing systems, inventory management systems," [5]
- horizontal domains: are areas organized around classes of parts of systems are called horizontal domains" ("database systems, container libraries, workflow systems, GUI libraries, numerical code libraries and so on" — so a software product that aims at buildin user interfaces, or a platform for cloud computing address horizontal domains).

One meets product lines in both kinds of domains, but it is most classical to apply this methodology to narrow vertical domains (for example control of hearing aids, or enterprise-resource planning systems). An example of a product line in a horizontal domain is the linux kernel.

The scope of the domain needs to be established based on sales needs, maturity of products and knowledge in the organization, and the potential of reuse. In general you want the scope be as narrow as possible, and you need to continuously monitor and maintain it, to avoid the *scope-creep* problem.

Common and variable properties of the system can be described by a domain model. Such a mode defines the scope of the domain, defines its vocabulary and the main concepts of the domain.

Domain models can be expressed in many ways, but most commonly as domain specific languages, or as *feature models*

The domain can be implemented as a framework. It is common to start with a reference implementation — a complete manual implementation of one family member, which is later refactored (through templates for example), in order to support generation of other products.

17.3 Domain Modeling with Feature Models

An important notation for expressing domain models is *feature models*. A feature is an end-user-visible characteristic of a system [13], or a distinguishable characteristic of a concept that is relevant to some stake holder in the project.

For example choosing a manual or automatic transmission, when buying a car, might be interpreted as deciding a feature.

Feature models are a simple modeling notation, which allows expressing relations between features. It is worth mentioning that the upcoming OMG standard, CVL, is going to use a notation which resembles feature modeling a lot.

The following figure summarizes the basic syntax of feature modeling notation:

Typically additional dependencies between features (those that cross the tree hierarchy) are stated on the side. In this model we might for example want to add that Electric engine *requires* Automatic transmission.

A feature model can play the same role as a DSL model in an model driven product architecture.

A feature model can be used to derive a desirable configuration, which can be fed as parameters to the code generation. In this sense a feature model is an extremely simple meta-model, which describes its models — configurations adhering to the constraints of the feature tree.

- *Mandatory features*: Every car has a body, transmission, and engine.
- *Optional feature*: A car may pull a trailer or not.
- *Alternative features*: A car may have either an automatic or a manual transmissi
- *Or-features*: A car may have an electric engine, a gasoline engine, or both.

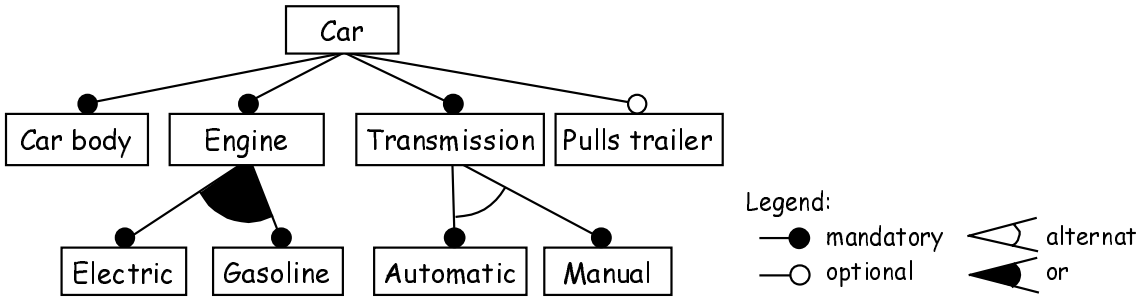


Figure 17.2: A simple feature modeling notation, after [4]

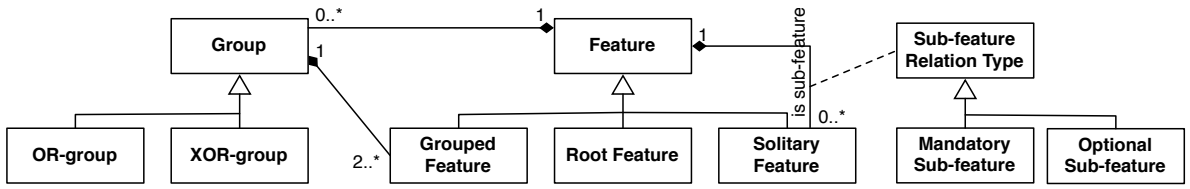


Figure 17.3: One possible meta-model for feature modeling, after [11]

More complex feature modeling notations exist. Extensions include adding references between features, and adding classifiers (feature cardinalities, or feature groups). The latter (so called cardinality based feature models) will likely be supported by the CVL standard.

It is worth mentioning that feature modeling languages are used by some commercial product line tools such as PureVariants and BigLever. Many configuration languages grown internally within various projects resemble feature modeling a lot. For instance the Linux Kernel project and the eCos operating system project, both use a hierarchical feature modeling-like language to describe their legal configurations, and to support deriving them. Linux Kernel's model encompasses some 5000 features, while eCos' model exceeds 1200 features.

[13] is the most popular work on feature oriented domain analysis, which has proposed the feature modeling notation. The best introduction to feature modeling is likely Chapter 4 of [5].

17.4 Domain Modeling with DSLs

DSLs and feature models are two different techniques that belong to the same continuum of modeling domains. See Figure 17.4.

Stahl and Völter advice that one should stay as much as possible to the left side of this spectrum. Namely, one should prefer modeling with feature models, or simple configuration parameters if possible, to avoid scaling up complexity needlessly.

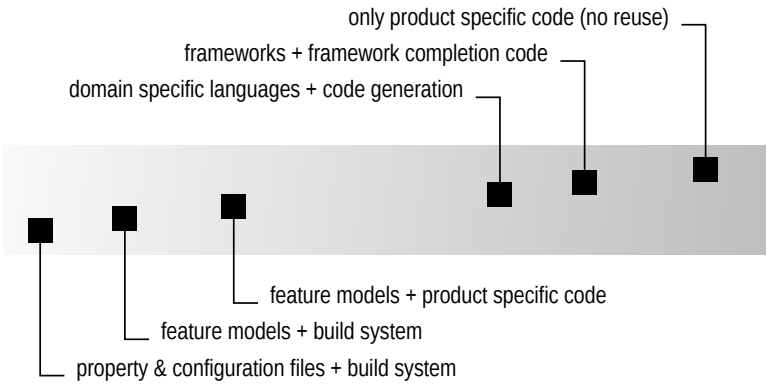


Figure 17.4: Spectrum of Domain Modeling [5].

Concepts that are common to all products in the domain belong to the platform implementation, while concepts and aspects that vary are expressed in your domain specific models. If you use feature modeling, the mandatory features correspond to common aspects of the system.

To decrease complexity the MDSD domain and the MDSD platform should be as close to each other as possible. Ideally the platform (or framework) should provide implementation of domain concepts.

If you use a DSL, note that most commonly structure is modeled, while behavior is provided by the framework/platform. If you do need to customize behaviour, it is recommended to reduce it to a small finite number of choices of different behaviour — and describe it using a feature model.

If this is not an option, try to reuse as much as possible existing languages such as statecharts, automata, BPMN/BPEL, activity diagrams and message sequence charts. Designing your own behavioral languages is known to be difficult to get right.

Inventing your own behavioral language, gives more flexibility than reusing existing one, but it increases risks and difficulty of achieving full automation.

Slogan: **narrow and simple is better than broad and complex!**

A good rule of thumb: if you need to introduce typical GPL constructs into your DSL, such as a loop, and they need to be generated from models (compiled into the target langauge) then you probably have grown your DSL too much.

DSLs should stay simple, and possibly declarative.

17.5 Domain Implementation

The implementation of the domain requires a choice of a suitable architecture. Or in other words we can only succeed with MDSD in some architectures.

[24]

In particular *frameworks* are a good architectural basis for MDD.

What is a framework? Framework is larger codebase, typically with quite a strict architecture, that can be extended into a complete application by providing adaptation code (for example via extending super classes), providing configuration parameters, XML models, or implementing call backs. Frameworks are ubiquitous in software development today. For example persistence frameworks, GUI frameworks, web programming frameworks, or enterprise systems frameworks.

Often framework adaptation code contains a lot of boilerplate, and it can be successfully generated from models describing the essence of the product.

Note that this is opposite to what UML-based CASE tools are doing. In such tools generated code is merely a skeleton of the classes, which needs to be filled in with custom code to obtain a running application. When you use a framework together with code generation, the framework provides the “meat” — the main code. This code combined with the logics generated from product models in your DSL (or feature models), together form a complete application in a process which requires no further manual coding.

It is an advantage of MDSD combined with frameworks, that due to use of automatic code generation it is easier to maintain invariants of usage for framework API — this is very handy, as violation of such invariants typically cannot be discovered early (errors manifests themselves only at runtime).

Another good architectural frameworks that work well with MDSD are component based architectures and middle-ware, which both require considerable boilerplate code to integrate together into complete products. This code can normally be generated automatically.

17.6 Other Advice on Realization

Stahl and Völter recommend placing domain concepts at the level of functionality, when implementing the platform. In their experience this decreases the semantic gap between models and code, and gives the biggest chances of success. So for example in an insurance application it makes sense to work with platform concepts like person or account.

This is consistent with advice they give also on other aspects: that MDSD should be applied on as narrow scope as possible.

They also advocate starting with an extremely small domain model, and narrow scope, and increasing it to the other aspects of the project iteratively as your expertise and your understanding of the domain increases. This is also a good advice for your projects.

nterestingly Clafer is a language used in some of the proposed projects, which allows seamless switch from feature modeling to structure modeling (class modeling). We are working on extending Clafer towards behavior modeling which would also allow incremental addition of behaviours for mature projects.

We recommend using a fully automated build process (and possibly also a continuous integration service). Once you require just one manual step in building an instance from your product line (adapting a single source file manually after generation), sooner or later the MDSD practices will be abandoned. This single step will bias you towards avoiding automation, and changes in models.

Of course this does not mean that you should not implement parts of the application manually — just this has to be done in a well-integrated manner; in separate files; separate directories, using programming language’s extension mechanisms, or weaving/composition via transformation.

ote that Stahl and Völter discourage using protected blocks in generated code, even though the technologys supports it well. In their experience, this blurrs the boundary between generated and manually written way too much, and also, sooner or later some code is lost if you do this.

Generate good looking code if possible. This raises trust and confidence in your generator (magic is scary), it also aids debugging. It is unrealistic to assume that developers will never look into the code.

Generate comments and white space. But it is important to make templates maintainable too. So in templates use indentation so that the template is readable. Obtain right indentation in the output by using a code beautifier component. Automatic beautifiers (pretty-printers) are available for many languages. Insert traceability links into the generated code.

A good trick is to generate some static code that uses user written manual code. This code can be dummy, but it would invoke the type checker and identify a lot of errors at build time, so that they are reported in the IDE. For example a missing interface extension, etc. In many frameworks such an error would be discovered at runtime, when the component is loaded.

18 Reading and Writing Research Papers

Reading: This note is largely inspired by my personal experience and Chapter 9 of Zobel's *Writing for Computer Science* [28], which is a generally good resource about writing for research students (including MSc thesis students). This the only book I know that is written with computing writers in mind, as opposed to aiming at social science students, or humanities students. Recommended! Our library should have a few copies.

If you need to learn more about evaluation and experimentation then read sections 6.7–6.8 in [27]. Otherwise this note should be sufficient for most projects in this course (and for most of your theses).

If you want additional inspiration on reading then google for tutorials (for instance "how to read a research paper in computer science"). In general, however, I think it is more important to practice reading by reading actual research papers, than reading papers on how to read research papers.

18.1 Introduction

Writing should start before the project's half way mark. Writing will structure your work. You will discover missing data early. You will uncover problems earlier. It will become more clear what work is necessary (because it influences the paper), and what is of secondary importance. Now it is a good time to start devoting resources to writing.

Recall that so far we have already discussed how to define a research problem (writing a project proposal) and that it is very important to have an idea what your claims will be and how you would provide the evidence for your claims.

This is the structure of the report that I suggested (see an earlier lecture note for an explanation):

1. Introduction
2. Background
3. *Real stuff comes here*
4. *Describing your technical solution*
5. *Or performing some analysis*
6. Evaluation
7. Threats to Validity / Discussion
8. Related Work
9. Conclusion

18.2 Scoping

Write down on a whitebaord or a piece of paper your results, small and big included. Identify the main result and organize the story around it.

- Which results are most surprising and interesting?
- Which outcomes are most likely to be used by someone?
- Which are the outcomes that do not justify publication? Are not worth of the space?
- Which data in experiments supports conclusions? Most experiments produce much more data than you can publish.

(list adapted from [28])

Then you repeat the same exercise as discussed a few weeks earlier: align the problem, the solution, and evidence speaking for this solution. It is inevitable to do small adjustments at the writing stage, as the course of the project very often invalidates your initial plan. Also very often we observe that the original scope was too ambitious, and one can write a better paper for a narrower scope.

18.3 Telling A Story

Zobel distinguishes four kinds of story:

- A *chain* is the most common structure in software engineering (and also in many theory papers). A chain is a simple story, starting with the problem and its significance, through development of the solution, to providing evidence for success (evaluation).
- In a theoretical paper it is often better to organize *by specificity*. You describe the story first on a simple framework, and then again on a complex one.
- Another possibility is *by example*, where you first present the problem solved by example, and also explain how the solution works by example. The presentation of the detailed algorithm is delayed until the later sections of the paper.

Most importantly it is inappropriate to structure the paper as a log of your work or of your thought process. It is not suitable to describe all the failed attempts, and conceptual improvements you made on the way. It is also not suitable to just report the experiments you have done in chronological order. All these approaches almost inevitably lead to unreadable papers.

Finally, always assume that most readers will not read your paper in detail (do you remember the lecture on reading a paper ? We adviced you to *avoid* reading most papers in detail!). So it is very important that the main points are visible early. This means that the crucial points need to be made in the title, in the abstract, and in the introduction. This also means that in each section, and indeed in each paragraph, you should strive to place the most important message of the paragraph in the first sentence. You should also reserve special typographical means such as emphasis, narrower text, enumerated and itemized lists, bold and *figures* (!) to the most important aspects of the story. Indeed, many authors believe that the main story should be possible to be grasped from just skimming through the figures in the paper.

In any case, if you succeed to draw readers' attention to the strongest points of the paper, there is higher chance that they would read it in detail.

For most unexperienced writers, it is a very good technique to write quickly. You jot down the outline for a section as bunch of points on a piece of paper. Then you quickly write the section following these points, as your thoughts dictate it. It is crucial not to perfect the write up. For many it is best to even avoid correcting

grammar and spelling mistakes at this stage. Simply press onwards without stopping, until you are done with a section, or its significant part. But do follow the outline, otherwise you will end up with a mess!

Do not stop to create figures or diagrams, just leave space for them, and refer to them as if they already existed. In any case, it is often very practical to delegate creation of figures to one of your co-authors.

If you are susceptible to writer's block, then definitely follow the above advice.

After the first draft is created this way, it is normally quite fast to improve in 2-3 sequential editing iterations. Often it helps a lot to read the text aloud when editing. Many people hear much more errors, and imperfections than they can see.

18.4 Abstract

Abstract is the executive summary of the entire paper. As a summary it should encompass not only the main result, but also the problem, and the evaluation. Shaw [22] reports that good ICSE papers discussed evaluation already in their abstract. This is important in software engineering, because of the emphasis put on the evaluation of results.

A good abstract is readable for non-experts, as it increases the chances that someone will build on your work (often application opportunities appear in other areas).

> I try to have four sentences in my abstract. The first states the problem. The second states why the problem is a problem. The third is my startling sentence. The fourth states the implication of my startling sentence.
>
> Kent Beck in [12]

Simon Peyton Jones summarizes this idea in the following points:[1]

- State the problem
- Say why it's an interesting problem
- Say what your solution achieves
- Say what follows from your solution

Here is an example that I invented on the fly:

> Feature modeling is an important element of development of Software Product Lines. Much research exists on feature modeling languages, and feature modeling tools. It is surprising though, that no experience reports exist of using feature modeling in industrial practice, and no characteristics of industrial models is openly available.
>
> In this work we identify, present, and analyze two real life feature models of operating system kernels, eCos and Linux. We find that many assumptions about feature models made by tool builders in academia do not hold in reality, and thus many tools are potentially not useful; real models are substantially larger, and with higher density of constraints than previously anticipated.
>
> 109 words

[1] research.microsoft.com/en-us/um/people/simonpj/papers/giving-a-talk/writing-a-paper-slides.pdf

It is not important to stick to the four sentences. Often it is easier to write two or three sentences for each of the four parts of the abstract. It is however recommended to stick to the four part structure, at least for your first draft.

Normally we avoid citing other papers in abstracts, since the abstracts get published in various websites and publication indexes, without the list of references attached. We only do this if the paper is improving directly on, or discussing in detail, some other paper.

Abstract is often limited to 150–200 words. In this project the abstract must be at most 150 words.

18.5 Introduction

Introduction is often very similar to the abstract. Sometimes people write the introduction by expanding a draft abstract into a section. Sometimes, we write the introduction, and then derive the main sentences from it to form the abstract.

In the introduction include the following parts:

- General statement introducing the area; You can most likely start with the first paragraph from your project description and evolve it.
- Explanation of the specific problem and why do we care about the problem.
- Explanation of your solution, and how it improves on the work by others. Relation to related work can be very brief, given that you have a separate extensive section devoted to this.
- A hint on how the solution was evaluated and what was the outcome of this evaluation.
- A summary (a "map") of how the paper is organized.

An introduction must be very reader friendly, assuming little prerequisites. Thus using technical jargon, complex terminology, mathematical formulae, and alike are discouraged in the introductions.

It is best to delay writing the abstract, conclusion and the introduction to the late phases of writing. Begin with writing the middle sections of the paper. Unless, you are really lost with what the main message of the paper should be — then often starting with an abstract (or an introduction) can help in scoping.

18.6 Evaluation

Section 6 (Evaluation) often takes form of an experiment that empirically demonstrates the quality (or preferably the superiority) of your result. Sometimes not just the evaluation, but the main body of the paper (sections 3–5) is empirical — if your objective is to study some existing phenomenon.

In such case you are obliged to discuss any possible weaknesses and limitations of the experiment and the limitations of the conclusions that one can draw out of it. These limitations take the name of *Threats to Validity*, and are always placed in the back part of a paper, directly after the experimental section; Section 7 in our example outline above.

In this note we discuss two main classes of such limitations and give example quotes from existing papers.

18.7 Threats to Validity

The two main sources of threats to validity in an experiment are the construction of the experiment and the generalizability of the results. In other words whether the results are correct for the population (sample) used in the experiment, and whether one can generalize them to a broader set of subjects. The former are called *internal* threats, the latter are called *external* threats.

The literature also includes other kinds (primarily *conclusion validity* and *construct validity*), but for our practice, and for many professional researchers, it will suffice to focus on internal and external threats. However, if you find it difficult to categorize the threats in your project into these two groups, you may want to study the more detailed classification; see for example [27].

First a bit of basic terminology on experimentation:

Subjects is the objects, or persons, being studied, whose properties or behavior is analyzed through the experiment. Subject and objects are sometimes used interchangeably, and some authors would prefer to reserve subject for human who are active in the experiment, and object for artifacts being manipulated (for example models, or program code).

Factor is the property, or technically a variable, whose correlation we study. For example in an experiment measuring speed of programming in C vs speed of programing in C#, the choice of programming language is a factor (and the speed of programming is another variable).

Treatment is one possible value of a factor, so in the above example C and C# will be two treatments.

Outcome is the result of the experiment, so some numbers, other data, or qualitative information, indicating some correlation between treatments and observed variables — in the example the correlation (or lack of thereof) of speed of programming and the choice of programming language.

18.7.1 Internal Validity

Internal Validity is concerned with confirming that the correlation between the treatment and the outcome is indeed casual, and not accidental, or caused by some third variable that has not been observed. For example we may discover that all programmers in C were faster than programmers in Java, but forget that all the programmers in Java took the experiment very late at night, when they were tired.

Thus internal threats to validity will often be related to the internal construction and execution of the experiment.

Main internal threats appearing in experiments with human subjects:

- History — when two treatments are applied to subject in order at different times. Subjects can be tired or change perspective due to this ordering.
- Learning effects — subjects can learn the task at hand over time. For example if the same person is first asked to implement a program in C, and then the same program in Java, she will likely be able to exploit the experience of solving the task in C to be more effective when working in Java.
- Testing — subjects should not know the results of the test on the fly (the 'election poll' effect)
- Mortality — subjects dropping out during experiment. You need to characterize whether this has not changed the representativeness significantly.

Main internal threats appearing in automatic experiments

- Intrusive instrumentation — the measurement or observing a program changes the variable being measured (for instance speed).

- Statistical insignificance (also in human experiments) — the sample is too small. Check experimentation handbooks if you want to design a statistical experiment.
- Direction of correlation — whether A causes B, or B causes A, or perhaps there are tertiary reasons that cause both A and B.

18.7.2 Examples of Threats to Internal Validity

The following comes from a paper that gathers a lot of statistics about the variability models of the Linux Kernel and of the eCos operating system:

> An internal threat is that our statistics are incorrect. To reduce this risk, we instrumented the native tools to gather the statistics rather than building our own parsers. We thoroughly tested our infrastructure using synthetic test cases and cross-checked overlapping statistics. We tested our formal semantics specification against the native configurators and cross-reviewed the specifications. We used the Boolean abstraction of the semantics to translate both models into Boolean formulas and run a SAT solver on them to find dead features. We found 114 dead features in Linux and 28 in eCos. We manually confirmed that all of them are indeed dead, either because they depended on features from another architecture or they were intentionally deactivated. [2]

Note that the paragraph has the following structure: first it gives the threat (here incorrectness), then explains what have we done to minimize this threat. You should always follow this pattern.

In another paper, we have studied evolution of the Linux Kernel model over time. Here is the internal threat paragraph from that paper:

> Git allows rewriting histories in order to amend existing commits, for example to add forgotten files and improve comments. Since we study the final version of the history, we might miss some aspects of the evolution that has been rewritten using this capability. However, we believe that this is not a major threat, as the final version is what best reflects the intention of developers. Still, we may be missing some errors and problems appearing in the evolution, if they were corrected using history rewriting. This does not invalidate any of our findings, but may mean that more problems exist in practice. [17]

18.7.3 Threats to External Validity

External validity discusses how far the results are generalizable, or in other words how representative the sample of subjects and the circumstances of the experiment were, to be able to draw general conclusion. Do you expect the same results to be confirmed in somewhat modified conditions?

Thus external threats are primarily focusing with the relation of the experiment conditions to reality (in the industrial practice, etc).

To enlist external threats ask yourself whether there is any interaction of the treatment with the way you chosen the sample (for a different sample the treatment could have different effect)? and Whether the set up of the experiment interacts with the treatment? (in a different context, the treatment could have different effect)

18.7.4 Examples of Threats to External Validity

The following quote comes from a paper studying two feature models of operating systems (Linux and eCos) to understand the nature of models and variability modeling in this domain. The main challenge of that paper was that it only draws on a small number of subjects—only two home grown modeling languages, and one model in each of them:

> The main threat to the external validity of our findings is that they are based on two languages and two operating systems only. On the other hand, both are large independently developed real-world projects, with different objectives: Linux is a general purpose kernel and eCos is an entire specialized RTOS for embedded systems. We believe that other related domains, especially embedded RT such as automotive and avionic control software, will share many characteristics with the studied systems. Further, comparison to other feature modeling languages, shows that both are representative of the space of feature modeling. [2]

The above argument was a mitigation argument (why do we think we can generalize the results). The one in the example below, is a scoping argument: instead of mitigating we explain two what kinds of subjects the results can apply:

> The Linux development process requires adding features in a way that makes them immediately configurable. As a consequence, it not only enables immediate configurability, but also makes the entire code evolution feature-oriented. Arguably, such a process requires a significant amount of discipline and commitment that may be hard to find in other industrial projects.
>
> Not all projects assume closed and controlled variability model. Many projects are organized in plugin architectures, where variability is managed dynamically using extensions (for example Mozilla Firefox or Eclipse IDE). Our study does not provide any insight into evolution of variability in such projects. [17]

18.8 Related Work

We now discuss how to collect, study and describe related literature for your project.

18.8.1 How to Find Related Papers

First, seek in the literature you already know.

In the introductory lectures for this project cluster we have given an overview of literature (books, and some papers) on the main subjects of modeling and transformation.

Second, ask your supervisor

Ask your supervisor in the meeting for information about the important papers, or names of people, or relevant workshops, conferences and journals.

Third, search online repositories

Check past proceedings of relevant venues or previous and newer work of the same authors by checking their DBLP page. Simply google for "dblp John Smith" to find most papers by John Smith, and "dblp Universal Laziness 2011" to find papers published in "Universal Laziness" conference/workshop/journal in 2011.

DBLP (http://www.dblp.org/search/index.php) is the most important index of computing research literature. It tends to index trustworthy outlets. Besides very new, and very old work, it is *often* the case that papers not listed at DBLP are not resspectably published.

Furthermore, use scholar.google.com to search research literature by keywords and/or names. While DBLP is useful in establishing what papers to find, often it links to paid sources (you should have access to most of these sources from ITU's campus). Google will often be able o find a free version of a paper, once you know what you are searching for.

If you absolutely cannot find the paper online, when Google fails, when ITU subscription fails, when our library fails, then it is entirely OK to politely contact the authors of the paper, asking for a copy.

Fourth, Follow References

While reading papers, mark positions in related work that seem relevant to your work. Obtain the papers online and read these, too.

Seeking by related work, has one serious disadvantage: it only allows you to move back in the past. You end up finding older and older papers. It is however very likely, that the newest papers are relevant for your project. To identify these, you can use reverse searching in Google Scholar. Find a paper that is likely to be cited by papers in your work area, search for it in Google Scholar, and click on "Cited by" link, to find newer papers that cite it.

18.8.2 How To Select Papers To Read

With so many sources of relevant literature, you will often find out that there are dozens, if not hundreds papers, that appear interesting to read on your subject. This is why it is much more important to read the right papers, than to read a lot.

With this in mind I propose the following 4 step method for reading (or not!) a research paper:

1. You start with the title. If the title is not promising a relevant paper, then discard the idea — do not even print it!
2. Then continue with the abstract. After reading the abstract ask yourself: can this work benefit my work ? Can I claim that I improve on it ? Is this work addressing a similar problem? Same problem? Does it appear to address a problem better than what I am doing ? If you answer any of these questions positively, then the paper should survive on your reading list. Otherwise discard!
3. Read the introduction, related work, and conclusion (I really recommend to read the conclusion so early!). Ask yourself the same questions as before. Do you still think that this paper is relevant ? If so, it might be time to print it. Otherwise discard!
4. Hopefully only a few papers survive until this stage. Probably you need to read them all. Remember to assess them critically, by comparing evidence provided with the claims made (see previous lecture).

Resist the temptation to read (print) many interesting papers. Focus on relevant papers.

18.8.3 Reading

When you got to reading the paper entirely, it is crucial that you try to read through the entire paper relatively fast, deck to deck, without stopping. Since you already know the introduction, related work, and conclusion, the paper should be easier to read, than if you approached it entirely sequentially. Still, It is completely fine to skip over the paragraphs that you do not understand.

It is more important that you do the first reading reasonably fast. During (or after) that, you may want to visualize for yourself the main points, and structure, of the paper. For instance pencil down an outline or a mind-map for the paper. Critically assess the claims and the evidence provided (see last week).

Most regular papers, with exception of Journal papers and very mathematical works, should be possible to process within one hour in this manner (even half an hour should be enough for first quick reading of many software engineering papers).

After that, if there is need, you should read it again. Normally all the hard parts are much easier to understand at the second reading. It takes less time to read twice (first fast, then slow), than just to read once (only slow).

18.8.4 Citing

Your paper should contain some 15-30 references. You can easily assume that the last page in the LNCS format (and sometimes a bit more) will be used by references. About half of the references will be related work. The rest will be background info, references to tools you are using, origins of standard definitions, background information and alike.

Assess trustworthiness of each cited paper. Articles with lots of typos and language mistakes, badly organized, or not written clearly, are often also flawed conceptually. Workshop papers, often present only simple ideas, with no supporting data. This is the same for short conference papers (say up to 6-8 pages). Full size conference papers are more solid. Journal papers are typically long, they provide solid evidence and much context information.

It is also important to distinguish respected authors and respected venues. To check the former, you can see the publishing record of the author in DBLP (see above). Ask your advisor for help, to assess reputation of publication outlets.

Avoid discussing at length (and sometimes citing altogether) papers that you assess as not very trustworthy.

Wikipedia entries, industrial white papers, student term projects, and alike are usually not trustworthy sources to cite for research results. If you want to cite an idea, you should always strive to cite the primary source that published it, not a book or website that described it post factum (as opposed to inventing it). Wikipedia tends to cite primary sources for lots of factual information. So it might be a good starting point to find literature (but cite the primary sources after verification; do not to cite Wikipedia directly).

Avoid long quotations. A long quotation should really be an important one, and it should really be important that you use someone else's words, instead of your own. **Never ever quote without citing the source.**

It is also safest to avoid quoting figures. If you do this, investigate whether you have the right to do this, and always cite the original source.

If you are using LaTeX, then you will find BiBTeX entries for most papers in DBLP. This saves a lot of time, and gives your reference list a uniform look.

18.8.5 Writing The Related Work

It is important that you discuss predecessor works, and works with similar objectives in related work. It is not enough to mention them! It is not even enough to say how they are related. Quotes like "a related work is XXX", or "YYY solves a similar problem in a different way" are extremely weak. You have to explain precisely what are the advantages (and disadvantages, if any) of your approach. What improvement do you offer ? **For each of them.**

18.9 Conclusion

Summarize the most important outcomes of the paper. Make the sentences sound strong (punch!). It should contain the main points that you want the reader (and the examiner) to remember.

It is also customary to include a paragraph about future work: what would, or will, you do approach as the next research step after these results.

18.10 Title and Author List

The title should be informative and short. Space is always precious, so it should ideally fit on one or two lines. Shorter title is also easier for other authors to reference within space limits (and normally you want to be referenced). It is fine to have a catchy, almost newspaper style, title, but remember that this should not fool search engines. Most people find papers using Google.

There are two dominant schemes in author ordering:

- Alphabetical by last name; more common in theoretical papers
- By degree of contribution of each author: dominating in software engineering.

It is also traditional, that if you co-author a published work with a faculty member, for instance your supervisor, then the ordering by contribution is only applied to students co-authoring the work, while professors are listed in the end of the author list.

Regardless the scheme, always the first author gets most visibility, and becomes the reference name for the paper, eventually. So if you feel that you contributed significantly more than others, you should insist on ordering by contribution.

For a project report or a joint thesis, I recommend alphabetical, to avoid group conflicts. If you choose to order authors by contribution, then please take this discussion early in the group, to avoid conflicts in the last day.

Individuals, who only contributed to a paper marginally, should not be listed in the list of authors, but acknowledged. This includes all people who did not contribute to the ideas and the story, but for example only implemented some tool under direction of others. Also people who contributed marginally, for example by giving feedback to drafts, should not be authors, but should be acknowledged. Acknowledgments are listed in the very end, under conclusion in papers, and early in a preface or introduction in theses or books.

18.11 Format

Plase use maximum 15 pages of LNCS format. Springer provides style files at:

- LaTeX: ftp://ftp.springer.de/pub/tex/latex/llncs/latex2e/llncs2e.zip
- Microsoft Word: http://www.springer.com/cda/content/document/cda_downloaddocument/CSProceedings_AuthorTools_Word_2003.zip?SGWID=0-0-45-1124637-0

A claerly marked appendix can be added on top of the 15 pages. You cannot expect however that the appendix will be read, or assessed properly.

18.12 Homework due latest on November 7th

- Agree in the group what will be the experiment in your project (if any)
- Designate the person to design the experiment (this person may need to read up about it).
- Sketch (=**write down**) the experiment design and a short discussion of threats to validity.
- Discuss it with your supervisor next week.

Usually one can identify 2–4 main internal threats and the same amount of external threats that should be described. The size of a typical threats to validity section in the LNCS format would probably not exceed one page.

18.13 Homework due latest on November 14th

- Identify 5-10 (for now) papers that are related work for your own work.
- Show the list to your supervisor
- Explain how it is related, whether you improve on them, or whether they have similar objectives.
- If you are able, already draft the related work section based on that, now (will require editing later)

18.14 Later This Semester

- **Nov 7:** experiment designed and threats to validity
- **Nov 14:** related work
- **Dec 6:** first complete draft of the report due
- **Dec 10:** reviews due (group receives 2 and delivers one review)
- **Dec 17:** final report due (formal hand in)

19 Theses & Projects in MDE

20 Exam Pensum & Format

20.1 Pensum (Exam Reading)

All lecture notes, slides, and exercises. Including the essential reading material linked from the notes, whenever indicated as necessary reading.

20.2 Exam Format

1. The exam is individual. There is no common presentation.
2. Exam takes 20 minutes including grading and communicating the grade.
3. You enter the room and you have 3 minutes to explain the main objectives of your project, its results and your role in the project.
4. You are allowed to bring in **printed** copies of maximum two slides to support your presentation.
5. After that we ask questions related to the project and to the course (see examples below) for about 10 minutes.
6. During the exam, examiners take notes (of what correct and what incorrect things you say).
7. After the entire group has been examined, we give a few minutes feedback for the group about the report.

Because we are a large group, please remember to bring your ID or student card to the exam — just in case we wanted to see it.

20.3 Example Exam Questions

The following are example questions that have been used in the exam after the 2011 edition of the course. They are real – I extracted them from my exam notes. Your questions will be similar, but not necessarily the same. Your project report has significant influence on the content of the discussion that follows.

We do not take the questions only from this list. We formulate the questions based on the curriculum of the lectures and based on your report. The list is provided to give you an impression of the exam contents. You are expected to be able to execute a knowledgeable scientific discussion about your project.

1. What is the most important finding of your project/paper?
2. What are the main limitations of your findings/conclusions/results?
3. There is an unresolved reference in your report. What should it point to? Tell me about this paper.
4. The paper [15] by Geiss is about improvents of the Varro Benchmark. Can you summarize the improvements they suggest? Why did not you use the improved work but an earlier one?
5. Explain the differences between direct manipulation approaches and graph transformation approaches to model transformation.

6. Explain which of the approaches (direct manipulation or graph transformation-based) is offering a higher level of abstraction? Which do you believe should be faster and why? [the report was about benchmarking model transformations]
7. Explain in detail Figure XX (sentence YY) from your report.
8. Summarize the methodology you assumed in your project.
9. Enumerate model transformation languages that you know, classify and characterize them.
10. What is meta-modeling? What meta-modeling language(s) do you know?
11. Summarize differences between Model-2-Model and Model-2-text transformations.
12. What are applications of model transformation tools? What are the applications of M2T? What are the applications of M2M?
13. You put forward a hypothesis that high level tools might be slower than low level. So why people would use the high level tools? Does your finding imply that high-level tools are useless?
14. Are the instances used for evaluation/benchmarking representative? How strong is your conclusion for the general usage scenario? [applies to many projects]
15. Why you do not use your ecore model in your Xtend program? [A very bad sign, that something is seriously broken. Report specific]
16. What is ecore? What is the relation to MOF? What is its relation with UML? (or what is the relation of MOF and UML)
17. What is OCL?
18. Can you explain the metamodeling hierarchy?
19. What does M3 mean?
20. Explain the metamodel presented in your report.
21. Write a simple OCL constraint formalizing this sentence in the report (say every trip has at least two drivers). [small syntactic deviations are permitted]
22. Explain to me the following OCL constraint. [often taken from the report]
23. How this constraint written in Java/Xtend would look in OCL? [often taken from the report/code]
24. What is a DSL?
25. What is the difference between using DSLs and using feature models?
26. Is the language you propose in your report a DSL? What is the domain it targets?
27. What is an internal (embedded) DSL? What is an external DSL? What implementation techniques you know for implementing external/internal DSLs?
28. Explain differences and similarities between Xtend and declarative ATL.
29. What does it mean for a model to conform to a meta-model? To what meta-model the model of Fig. XX conforms? To what the meta-model of this model conforms?
30. What is the key rationale for this work? [despite this being a mandatory project, you must be able to explain why taking up this project makes sense — not being able to do this normally indicates that you have a shallow attitude to the subject matter]
31. On page 2 you recall constructivist theory of learning. Why do you report it as related? In what way do you exploit it? [if you use some body of knowledge in your project, you are expected to account for it]

32. What languages/tools can you indicate that can be used for implementing internal/embedded DSLs? What are the advantages of implementing a DSL as an internal DSL?
33. You write that the documentation is scarce for DSL tools. Did you manage to access the book about this project? [you will be made accountable for using easily available literature on your subject — in this case the book was available in the library, but the group did not care to get it]
34. What is the difference between the Xpand and Xtend language ?
35. How would you design an evaluation experiment for your project if you had a possibility (and time) to use human subjects ? [assuming that this made sense for the particular project — not always using humans in evaluation makes sense]
36. Can you give examples of successful domain specific languages?
37. What is a general purpose language (GPL)? How it differs from a DSL?
38. What is a product line architecture?
39. What is the most common application scenario for Model Driven Development?
40. What is Xtext? Its use case?
41. Describe the process of implementing a DSL using Xtext.
42. What is the main purpose of CVL/Clafer/Ecdar/BIM/Entity-Relationship models? [the question is about the main modeling method/language used in your project]
43. What is the purpose of variability modeling? What variability modeling language do you use?
44. What similar approaches to the same problem can you point out in the literature? [this is a question about your related work section]
45. How paper [XX] helps/could help your project? [XX was cited in the report]
46. What tasks are left to complete your project ? [The project has not been completed]
47. Please draw on the whiteboard classes representing the core (2-3) concepts in your model, and the relations between them.
48. Is the transformation you implemented reversible?
49. Have you used automated testing in your project? How coul you have used it?
50. What is the difference between a conformance relation and inheritance relation in class models?
51. What is aggregation (composition)?
52. What is abstract syntax? Concrete syntax? How do we specify abstract syntax of a language? How do we specify concrete syntax?
53. What is Eclipse Modeling Framework? What it can offer to a software developer? How did you use EMF in your project?
54. What is a left-recursive grammar? Did you have a problem with it in your project? What is left-factorization of the grammar?

20.4 Example Assessment of The Report

Each report is assessed before the oral examination and this assessment influences the grade. In most typical cases, the oral exam can confirm, or lower/increase by a step the grade suggested by the report. We have had however cases of people failing with very good reports, and people achieving good grades with average report. In the former case it was clear that the students knows little about the work, in the latter case it was clear that the student knows the subject matter very well, but the report has been influenced by the generally lower level of the group.

Here is an example of my assessment notes for some report (to show you an example of what kinds of things are taken into account):

- A lot of operational statements in the report. [operational means low level reporting of what we did, step-by-step, instead of concise synthesized description, focused on crucial points, overall process, findings and conclusions. Operational reporting usually characterizes authors who do not understand what they do, so they simply report the steps they perform.]
- Threats to validity were quite good.
- The report is thin on the whole purpose of this endeavor. What are the conclusions they can make from it? [so the motivation was weak, and the conclusions were very narrow]
- Broken citations, quite bad, chatty, even spoken language used. Not up to the point, not focusing on the main value of the paper. [bad language, bad typesetting, bad figures are often also very good indicators of bad work]
- Examples are not used to explain the main differences between the benchmarked approaches.
- Background is mixed with methodology.
- Hypothesis is hidden in background.
- Often unrelated paragraphs tacked together [there is no reasoning flow from paragraph to paragraph]
- The report has figures (Fig .1) not referred from the text, and not explained in the text.
- References to individual methods in Java code, instead of high level description of concepts and computations that they realize.
- The project has essentially failed as F is shown to be faster than xtend (this in itself would be a very strong conclusion, if the methodology was trustworthy) [such a conclusion was very implausible, and was rather an indicator that the implementation in xtend was very lousy, not that F was faster]
- The report is not clear about the benchmark not being representative.
- I asses the report as being below average. [below average means below 7]

An example of an OK, albeit not perfect, report is *Model Driven Development: Co-Evolution of text-processor model and plain-text model* by Kim, Madsen, Izaka, Christensen, Larsen, Grøn and Aljarrah. Find it in the project base at: https://mit.itu.dk/ucs/pb/project.sml?project_id=1182864. The above assessment points are for a different report.

21 Writing, Reviewing

Reading: A commendable book on concise writing is *The elements of style* by Strunk and White. It is a tiny booklet, that every professional writer should study.

21.1 Writing

If you wondered why it is so slow to read many research papers, here comes one answer: many authors very consciously increase density of information by using shortest possible way to phrase their message. Thus a research paper is able to convey much more information on a page of text than, say, a daily newspaper.

There are few very simple principles:

- For every paragraph and section ask whether it is needed? Interesting stuff is not automatically granted space in a paper. What stays in the paper is only the things that advance your story: argue for your claims, and lead to your conclusions. Everything else should be removed.
- The very same principle applies to individual words: words with no added information should be removed.
- Passive voice should be avoided (unlike in this sentence). It is better to say: We tend to avoid passive voice. Passive voice usually makes the text longer and harder to read.
- Shorter phrases, and punctuation can usually replace longer phrases.

Well edited text reads smoothly. It adds a *lot* of *credibility* to your writing.

I use one method of text editing effectively.

- Read the text aloud (perhaps close the door to your room ...)
- Correct any mistakes you hear, and any phrases that do not read well.
- Once a paragraph is corrected this way, read it aloud again, and repeat the process until you cannot improve it any more.

Normally if you cannot see any mistakes and imperfactions, others can still see some. Ask someone else to proof read.

Never ever forget to run the spell checker.

An alternative method is to work with proofreading of printouts. I find it much less effective for small scale editing. Printouts are better for earlier phases of editing, like monitoring the story, order of paragraphs, etc.

21.2 Reviewing

The exercise with reviewing is aimed at you experiencing how to critically look at your own work. Invariably you will see lots of weaknesses with the work of others — so after writing reflect, whether you could not learn to read your own writing equally critically.

It is best if the reviewers in each group are people who will contribute most to the final shape of your paper.

1. Email a pdf paper (15 pages + evt. appendix) to wasowski@itu.dk by 5.00am on Dec 7th.
2. Soon after you will receive one of the papers of other groups to review in your group (emailed by Andrzej to contact person)
3. Write the review until Tuesday Dec 4th at 23.59 and send back to Andrzej
4. Your review should have the following section: summary, points in favour of the work, points against (weaknesses), detailed comments.
5. Summary: write the abstract of the paper in your own words. This is to get the authors an impression what did you understand from their presentation.
6. Points in favor: write what you find good about this work, including the problem solved, the solution method, the results, and the presentation. *Always write some positive points*, and always write them before the negative ones.
7. Points against: raise any issues, unclarities, flaws, errors, bad presentation, lack of significance, etc.
8. Detailed comments: attach any minor comments like grammar, sentences hard to understand, spelling, typesetting mistakes, etc.
9. Expected length of a review is 500-1000 words.

The review is anonymous, so that you can feel free to criticize (but be polite).

The written review is your last major opportunity to get feedback.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers: Principles, Techniques, and Tools. Edition 2.* Prentice Hall, 2006.

[2] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 73–82. ACM, 2010.

[3] Frank Budinsky, David Steinber, Ed Merks, Raymond Ellersick, and Timothy J. Groose. *Eclipse Modeling Framework.* Addison-Wesley, 2004.

[4] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative programming for embedded software: An industrial experience report. In Don S. Batory, Charles Consel, and Walid Taha, editors, *GPCE*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2002.

[5] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming. Methods, Tools, and Applications.* Addison-Wesley, 2000.

[6] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches, 2006.

[7] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages.* Addison-Wesley, 2011.

[8] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 471–480. ACM, 2011. http://doi.acm.org/10.1145/1985793.1985858.

[9] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. http://doi.acm.org/10.1145/505145.505149.

[10] Daniel Jackson. *Software Abstractions.* MIT Press, 2006.

[11] Mikolás Janota, Victoria Kuzina, and Andrzej Wasowski. Model construction with external constraints: An interactive journey from semantics to syntax. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 431–445. Springer, 2008.

[12] Ralph E. Johnson, Kent Beck, Grady Booch, William R. Cook, Richard P. Gabriel, and Rebecca Wirfs-Brock. How to get a paper accepted at oopsla (panel). In *OOPSLA*, pages 429–436, 1993.

[13] Kang et al. Feature-oriented domain analysis (foda) feasibility study. Technical report, CMU/SEI-90-TR-21, 1990.

[14] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. In *9th OOPSLA Workshop on Domain-Specific Modeling*, 2009.

[15] Steven Kelly and Risto Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, 2009.

[16] Anneke G. Kleppe. *Software language engineering: creating domain-specific languages using metamodels.* Addison-Wesley, 2009.

[17] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the linux kernel variability model. In Jan Bosch and Jaejoon Lee, editors, *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2010.

[18] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[19] Richard F. Paige, Jonathan S. Ostroff, and Phillip J. Brooke. Principles for modeling language design. *Information & Software Technology*, 42(10):665–675, 2000.

[20] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer Verlag, 2005.

[21] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003. http://csdl.computer.org/comp/mags/so/2003/05/s5019abs.htm.

[22] Mary Shaw. Writing good software engineering research paper. In *ICSE*, pages 726–737. IEEE Computer Society, 2003.

[23] Alan Snyder. How to get your paper accepted at oopsla. In *OOPSLA*, pages 359–363, 1991.

[24] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Wiley, 2005.

[25] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, 2003.

[26] David S. Wile. Lessons learned from real dsl experiments. *Sci. Comput. Program.*, 51(3):265–290, 2004.

[27] Claes Wohlin, Per Runeson, Martin Host, Magnus C. Ohlsson, Bjorn Regnell, and Anders Wesslen. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.

[28] Justin Zobel. *Writing for Computer Science*. Springer Verlag, 2004.