Model-Driven Development

Lecture Notes (pre-course edition, d. 05/02/2014)

Andrzej Wąsowski

IT University of Copenhagen. Spring 2014

Ingredients

Part I.

Introduction to Model-Driven Development with Domain Specific Languages

1. Introduction

Reading: This is a project course. 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 includes lecture notes, important papers, the literature relevant for your report, and your project report.

Please read [24], 20 minutes in total.

You may want to read [11], 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 [27].

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.

[1] http://www.boeing.com/commercial/747family/pf/pf_facts.html retrieved 2012/08/28

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.

> *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. [24]*

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 introduction of compilers and relational databases — 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).

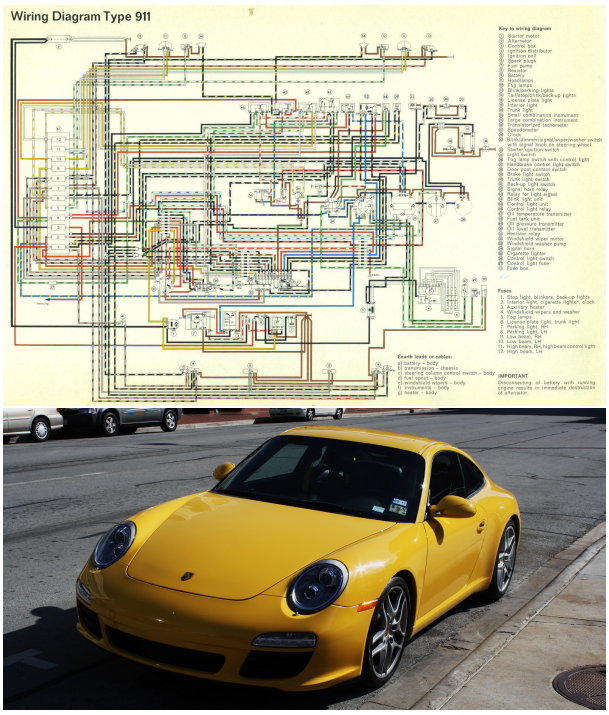


Figure 1.1.: The electrical scheme of Porsche Carrera 911 (above) and the actual system (below)

So what is a model?

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]

[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

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 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* [24].

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]

[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 [19].

- 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.
- Models provide useful oracles and visualizations for 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 languages specialized for model transformation.

1.4. MDD Practice

Hutchinson and coauthors [11] 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 balanced view on MDD.

Table 1.2 quantifies the above influences by looking, which MDD application areas lead to increase in productivity and maintainability and to what extent. 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 testing) are beneficial if they are used. For testing the beenfit is less clear cut.
- Use of models for communication, design, documenting, and code generation are most wide spread.

Other findings of the survey include:

- 83% of respondents think that MDE is a good thing and 5% that it is not.
- 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 domain specific languages (DSLs).
- Huge productivity gains are quoted in the qualitative study (at least two-fold, up to eight-fold), which are sometimes hidden or downplayed, to protect against cut downs.
- 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 [11].

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.

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.

We will cover the following subjects approximately:

- Introduction, motivation, overview
- Meta-modeling and Domain Specific Languages
- Constraining models (besides types and cardinalities)
- Design of Domain Specific Languages
- Support for Concrete Syntax (the Xtext tooling)
- Model transformation (the Xtend language)
- Product Line Architecture and Variability Modeling

These lecture notes also contain a number of methodological subjects:

Table 1.2.: The impact of MDE activities on productivity and maintainability according to the study presented in [11].

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%

- 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, and to support your project in this course.

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).

In the end of the semester, we will devote one lecture to exam preparation. We will show example exam questions, and, possibly, stage a mock-up exam in the room (if there are volunteer students).

Details of syntax of various technologies we discuss are not part of the exam pensum, but some rough idea of how to use them is needed, including ability to write examples on a whiteboard (perhaps using approximate syntax). Always ask if you are in doubt, if you should study some aspect of the material deeply, 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 [4], although I hope that material in the lecture note, and the free online tutorials, are sufficient for our needs.

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 [27]. This hierarchy can be found also in *UML 2.0 Infrastructure Specification* from OMG, however that spec is not for those of faint heart.

If you never been exposed to class diagrams, then *please read* [9], or a similar source on class modeling.

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.

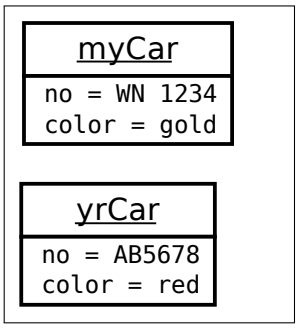
⚠ **Definition 2.** A *class* is an abstraction that specifyies attributes of a set of concept instances (objects) and their relations to other concepts (objects).

Observe that, in the above, we do not mean that a class is strictly 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 Java or C# classes are. This big leap from implementation to a conceptual application domain is necessary in order to obtain productivity gains promised by MDD.

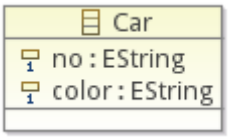
2.2. Class Diagram Primer (⚠)

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 says 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:

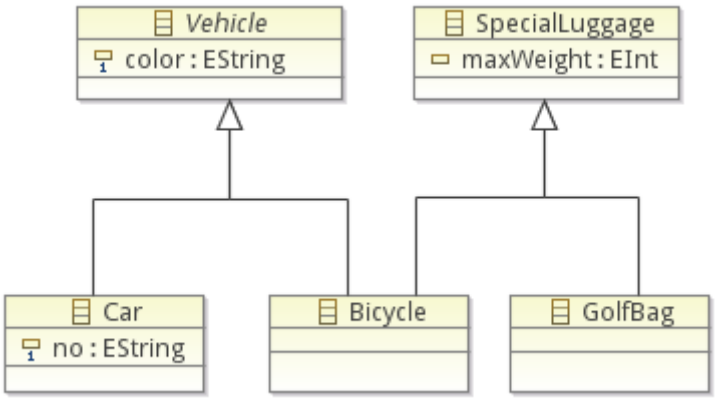


The above block represents class Car. The name is not underlined, and attributes have types, not values. They also have a *multiplicity* constraint (here simply "1"), which says that both attributes are mandatory for any instance of this class. By convention class names are capitalized, and instance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

In order to make the discussion a bit more precise, we introduce a formalization of class diagrams in first order logics. For each class named C we will introduce a unary predicate of the same name $C(x)$, that holds for objects of type C. So for this example we have a predicate Car describing objects that conform to the type Car.

For simplicity of discussion we will ignore attributes in the logical formalization.

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").

We can again formalize the inheritance relation using first order logic formulae. If class A generalizes class B, then there holds an implication from the predicate representing the latter $B(x)$ to the predicate representing the former, $A(x)$. For our example we have:

$$\forall x.\,\mathsf{Car}(x) \rightarrow \mathsf{Vehicle}(x) \tag{2.1}$$

$$\forall x.\,\mathsf{Bicycle}(x) \rightarrow \mathsf{Vehicle}(x) \tag{2.2}$$

and similarly for Bicycle and GolfBag with SpecialLuggage.

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.

To formalize the notion of abstract classes we need to introduce a predicate representing instances of each class. For class C lets call the predicate $C^{\mathrm{i}}(x)$. This is a stronger predicate than the predicate $C(x)$, which also covers instances of subclasses of C. Usually we have that for each class C the following holds:

$$\forall x.\,C^{\mathrm{i}}(x) \rightarrow C(x) \tag{2.3}$$

but **not**

$$\forall x.\,C(x) \rightarrow C^{\mathrm{i}}(x) \tag{2.4}$$

Now the class is abstract if it has no instances. So for example for the Vehicle class:

$$\forall x.\,\neg\mathsf{Vehicle}^{\mathrm{i}}(x) \tag{2.5}$$

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) have been made using tools of the *Eclipse Modeling Framework*. 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.

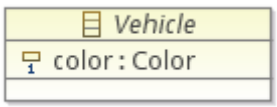
EMF provides simple types (for example **EString** used above), which are mapped to Java types during code generation.

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:

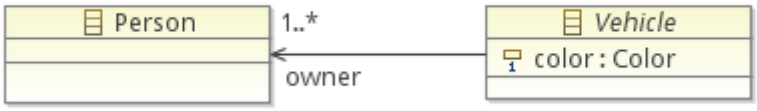
Enumerations can be used as types for attributes:



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 [4] (search for **EDataType** in the index).

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



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.

In EMF associations are unidirectional binary references. Unidirectional references can only be navigated in one direction. In UML references can be bidirectional and n-ary. For our purpose of meta-modeling, binary references are typically sufficient. Higher arity references can be always handled by creating an explicit class that will reify the association (similar to UML association classes). But, as already said, we rarely need it in language design.

Bidirectional references can be simulated using two unidirectional references. EMF allows to link two unidirectional references using the EOpposite property of the reference. In such case the generated code maintains links in both directions: whenever you add a link in one direction, the link in the other direction is created automatically. The mechanism is a bit complicated, and has shortcomings, so test well, when you rely on this. In particular, a reference cannot be EOpposite to itself,[1] and special care might be needed if you use references of multiplicity higher than 1.

In first order logics, we can model references using binary predicates. For each reference r we introduce a predicate $r(x,y)$ relating the objects being linked. For instance for the owner reference above, we would introduce a predicate $\mathsf{owner}(x,y)$. Associations (references) are typed so:

$$\forall x.\, \forall y.\, \mathsf{owner}(x,y) \rightarrow \mathsf{Vehicle}(x) \wedge \mathsf{Person}(y) \tag{2.6}$$

[1] This sounds a bit complicated, but in fact it appears in real domains for symmetric associations between objects of the same class. For example consider a class Person and unidirectional reference marriedTo. One way to model this in EMF would be to make the reference a bidirectional association, but this would require that it becomes an EOpposite of itself, which is not supported.

Now, since the multiplicity constraint is $1..*$ on this reference we have additionally that:

$$\forall x. \exists y.\ \mathsf{Vehicle}(x) \rightarrow \mathsf{owner}(x, y) \tag{2.7}$$

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—effectively, directed subgraphs of any object diagram consisting of reference links typed by references edges labelled with black diamonds have to be trees (or forests).

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

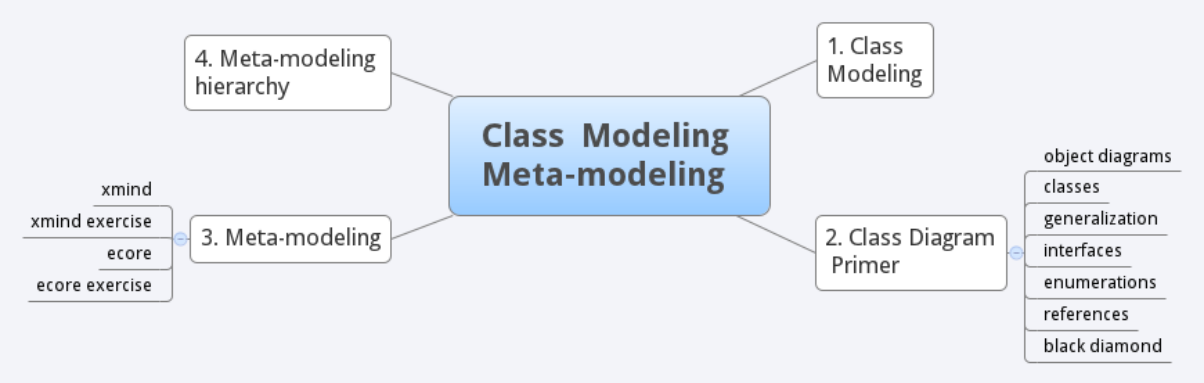
2.3. Meta-Modeling

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. It turns out that one can (retro-actively) provide an ecore model representing . . . the ecore abstract syntax (and MOF model representing MOF syntax, and a UML model representing UML syntax). Figure 3.13 shows this meta-model.

⚠

This has actually sometimes led to confusion that some languages are defined in themselves, for example 'UML defined in UML' or 'ecore is defined in ecore'. 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 (Fig. 3.13). 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 [4]. This model has only 4 classes, and makes it quite easy to understand the main idea behind representation of ecore in ecore.*

2.4. Meta-Modeling Hierarchy

OMG (Object Management Group) organized models and languages in a hierarchy of abstraction layers, also known as the OMG modeling architecture. This is exemplified in Fig. 2.3 using our mind-map language. At the very top level we have the ecore language (M3) which allows describing class diagrams. Instances of this language are class diagrams at level M2. A class diagram describing an abstract syntax (the meta-model) of the mindmap language belongs here. Note the *conformance* relations between languages (and instance-of relations) between intances at level M2 and classes at level M3.

One level below, at M1, we have concrete models in the mindmap language, here shown using notation resembling object diagrams (so their abstract syntax is shown). The model at M1 describes mindmap notes of a concrete lecture. Again, note the conformance (and instance-of) arrows crossing the two layers, M1 and M2. At the bottom level of the hierarchy (M0) we have the physical world (or the domain) that is described by the models at M1. Here at M0 we find a concrete lecture, which has been abstracted by notes at M1.

Figure 2.4 shows the same architecture but using the UML as an example, instead of the mind-map language. Design of the UML has actually been the main rationale for organizing this architecture. Since then, it has proven very useful for understanding layers in design of domain specific languages, like our mindmap.

The UML hierarchy is a bit tricky. Note that the entire UML 2.0 is in M2 in the Figure. But UML 2.0 contains conformance as part of the language. It contains both class diagrams and object diagrams. The figure actually shows how UML tools are implemented to support this, using general language processing stack, like EMF. The instance-of relation between a Class and Instance becomes a regular reference (association) in the implementation—see the arrow labelled classifier in M2.

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. ⚠

The final example (Fig. 2.5) shows the same architecture, as realized by W3C technology stack for structured data XML. At the top level we have the XML Schema Language that conforms to its specification in the XML Schema Language (again, after the language has been designed, it has been described in itself and the corresponding xsd file has been published). At level M2 we have XML Schema for concrete languages. Here, I use the XMI language as an example. At level M1 we have concrete XML files conforming to the schema of M2. In the example I use the mindmap.ecore file that conforms to the XMI schema for model representation in XML format.

The familiar XML stack has very similar aims to meta-modeling languages: describing structures and data in a standard manner. The main differences are that (1) XML documents are not really meant to be processed by humans, and (2) that XML processing stays largely on the level of strings or trees. The tools for processing models usually stay at a higher abstraction level. As we will see in the later lectures, models are processed using languages that support standard object-oriented programming model.

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.

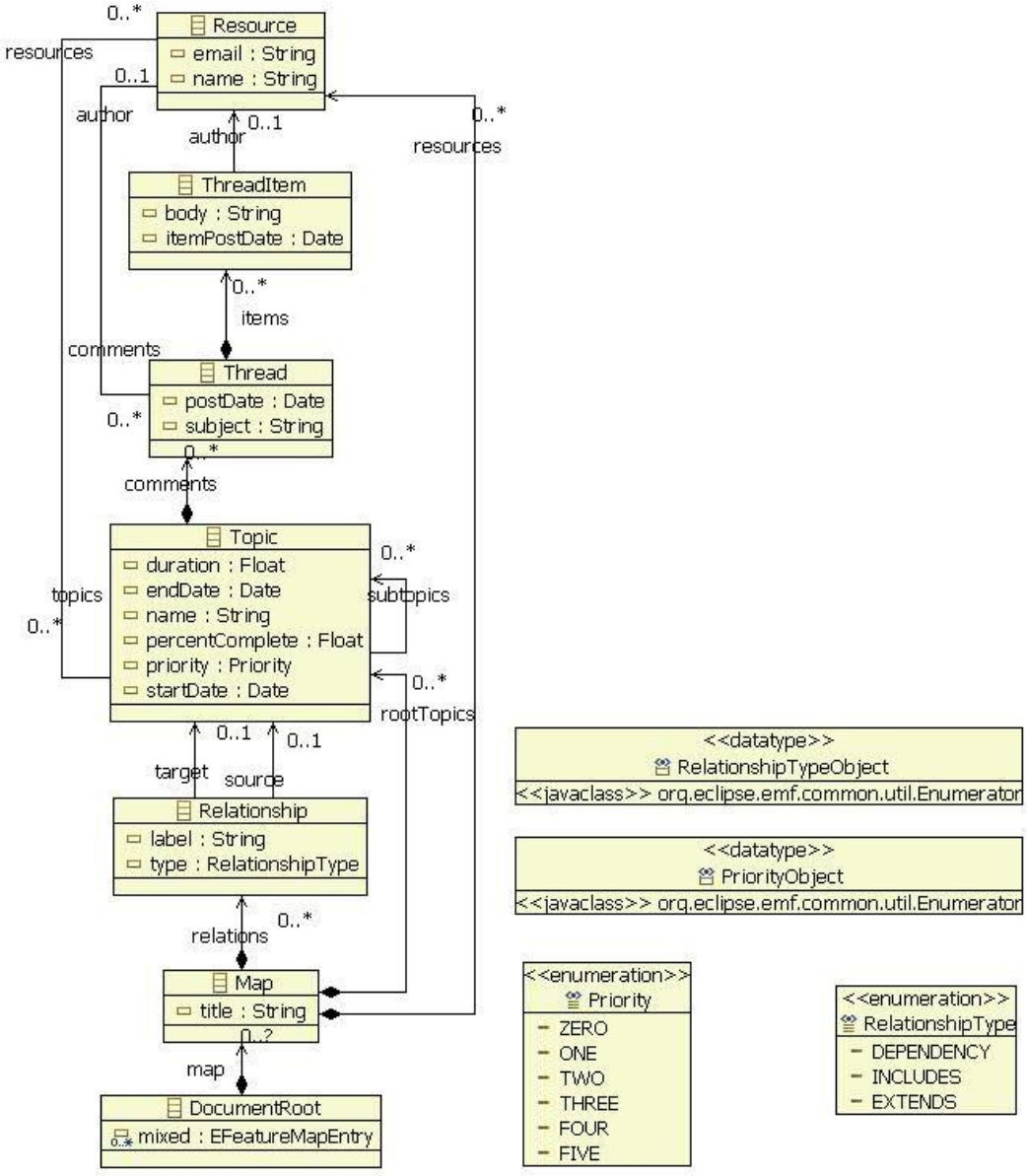


Figure 2.1.: Meta-model of a mind-mapping language.

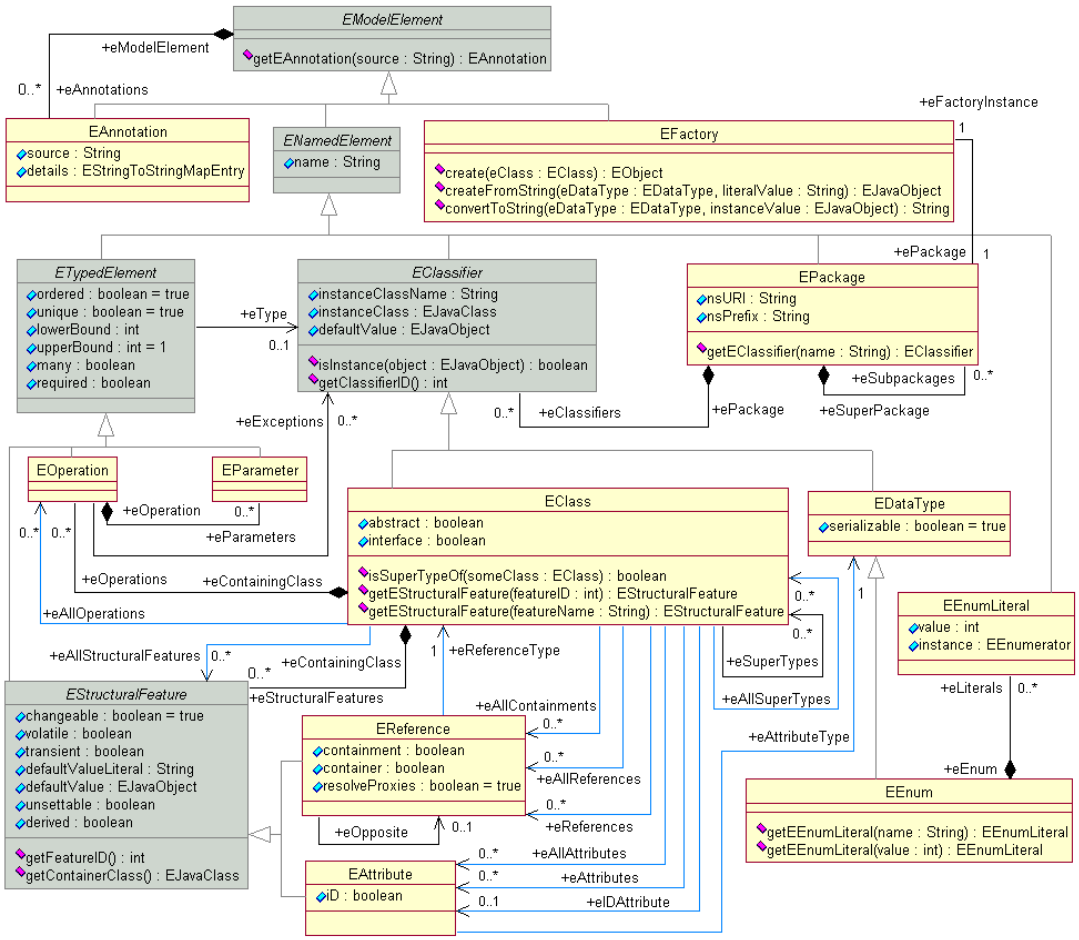


Figure 2.2.: An ecore meta-model of ecore.

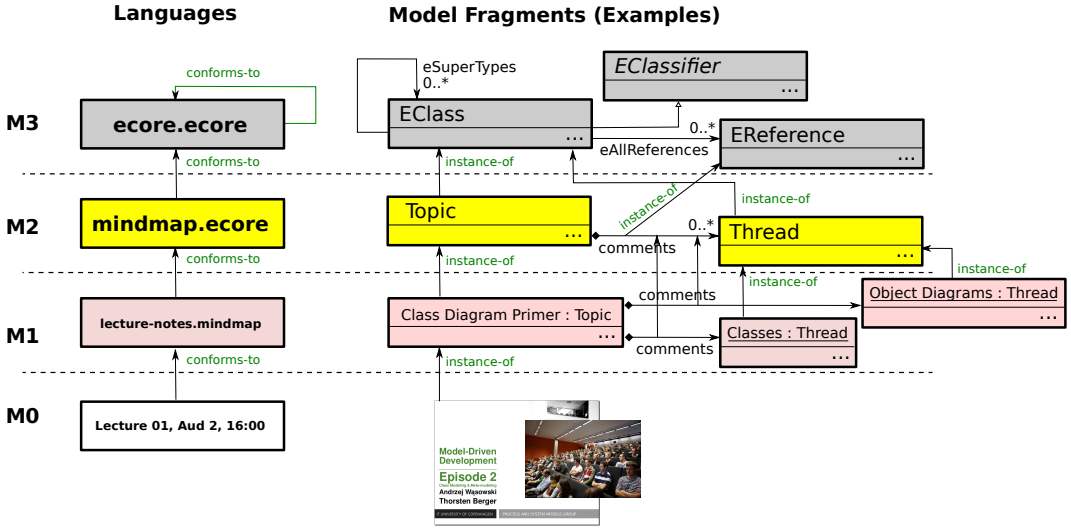


Figure 2.3.: Metamodeling-hierarchy illustrated using the mindmap example language.

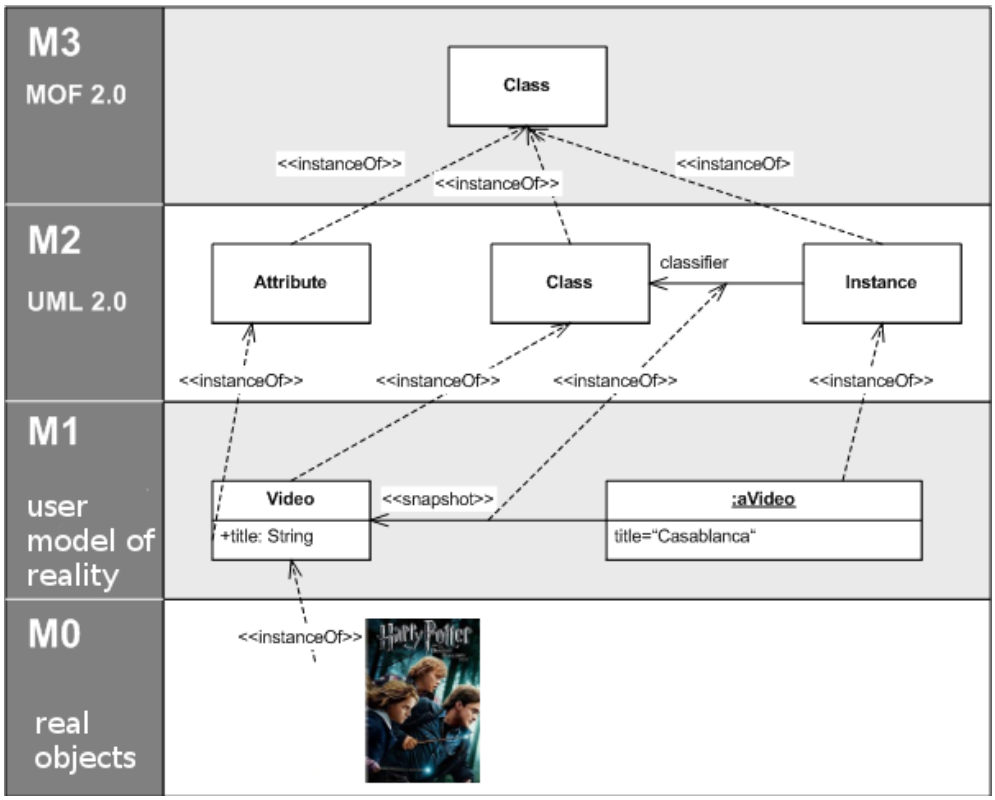


Figure 2.4.: Metamodeling-hierarchy illustrated using UML (source: Wikipedia).

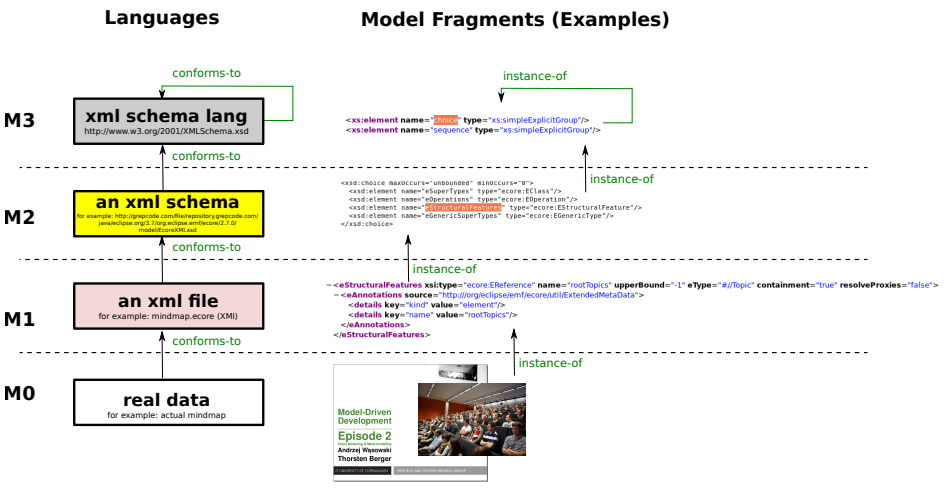


Figure 2.5.: Metamodeling-hierarchy illustrated using XML technology stack.

3. Exercises on Class Modeling

Objectives

- To install the modeling environment (Eclipse Modeling Distribution)
- To learn basic modeling tools of EMF
- To operationally recall modeling using class diagrams

I estimate that you have about 2 hours in class to complete this task + about 4 hours of self study time at home. One possible reference on class modeling is [9], but many other sources are available, also online.

Task 1. Install Eclipse modeling distribution 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.

Task 2. Read the class modeling tutorial in the bottom of this exercise set. Then proceed to the next task.

Task 3.* Create simple class diagram using the tree editor of Eclipse following this description of a domain:

A family consists of persons. Each person may be married to another person. ~~The relation of being married is symmetric (if I am married to you, you are married to me).~~ Each person may have a parent, and each parent may have multiple children. ~~Again a symmetric relation.~~ Each person has a name, age and a CPR number. Each person may be enrolled in a university. University may own one or more study programs.

Create a valid instance of your diagram representing Bob married to Alice, with their son Sam enrolled in the SDT programme of IT University.

The tree editor is much more robust and solid, but also try to initialize a diagram file and create a diagram in UML Class Diagram visual syntax.

Print your tree view, the diagram view, and a screenshot of your instance and hand-in to the teaching assistant as homework.

Due by: beginning of exercise session next week. Expected size: 1 page, no text.

3.1. A Short Tutorial on EMF

This tutorial is adapted after Vogel's tutorial [1].

Our objective is to show how to create a simple class model with EMF. In Eclipse, every file belongs to a project. So we need to first create a project. Moreover the project needs to have the right nature, to enable avialability of tools (here this has to be an EMG project).

Create a new project called `dk.itu.smdp.tutorial` via File → New → Other → Eclipse Modeling Framework → Empty EMF Project. This can be seen in Fig. 3.1

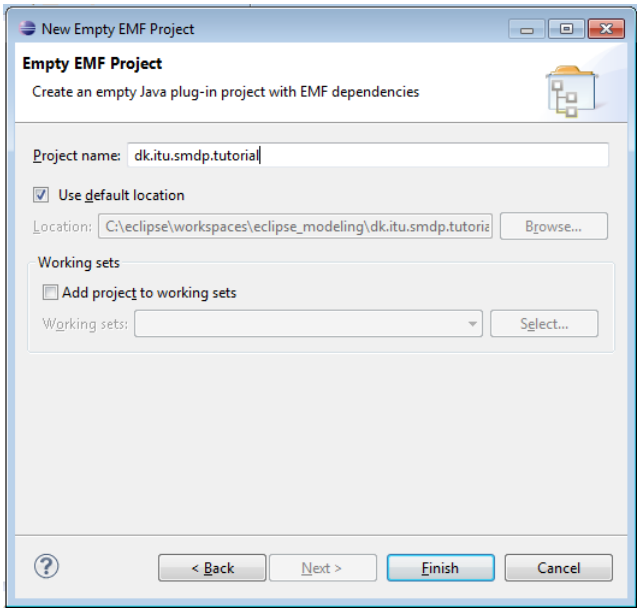


Figure 3.1.: Creating an EMF project

A model needs to be placed in a file, a model file. EMF distinguishes between model files (abstract syntax) and diagram files (concrete syntax). We can create both with one operation in the user interface.

Select the model folder, right-click on it and select New → Other → Ecore Tools → Ecore Diagram (c.f. Fig. 3.2).

Choose to create a new model. Enter *WebApp.ecore* as the Domain File Name parameter (Fig. 3.3) (domain model is the wording for "class diagram" here, this file will store the abstract syntax, while the diagram file will store the concrete syntax). This should open a visual editor for creating EMF models. You can see that in the models folder (in the Package Explorer) two files were created `WebApp.ecore` and `WebApp.ecorediag`.

Open the Properties view via the menu Window → Show View → Other... → General → Properties. This view allows you to modify the attributes of your model elements (Fig. 3.4). At this point it is also useful to mention the quick access feature of Eclipse. Almost any deeply

[1] http://www.vogella.com/tutorials/EclipseEMF/article.html

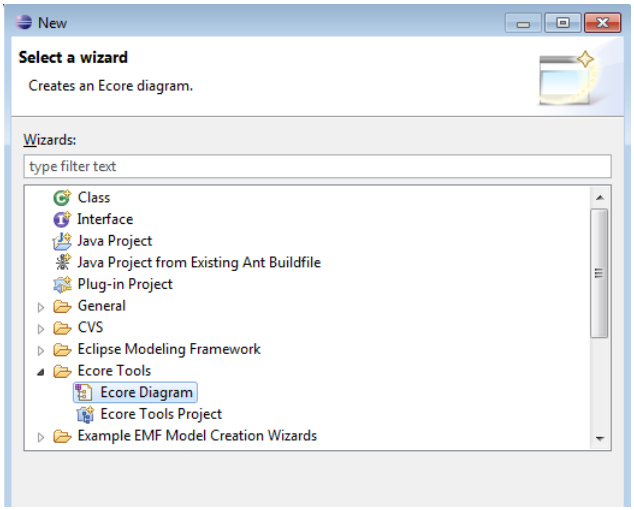


Figure 3.2.: Create Ecore diagram

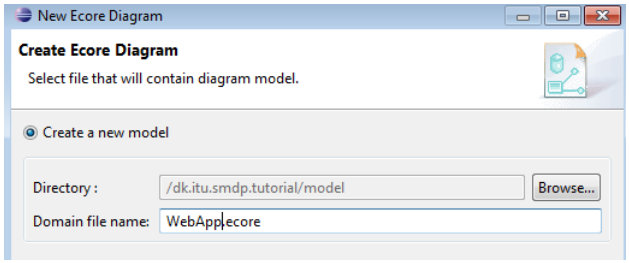


Figure 3.3.: Domain File Name

nested property or menu item, can be accessed quickly, if you remember its name. Press Ctrl+3 and type "properties", then select from the menu. You will open the same view as the previous long menu selection. This way a lot of complex operations can be speeded up.

Classes. In order to create a new class, select Eclass from the tool pallete on the left of the editor, and click anywhere on the canvas to create a class. Create classes for Website, Webpage, Category and Article (see Fig. 3.5).

Class Attributes. Use the EAttribute tool from the pallette to assign to each class an attribute *name* (see Fig. 3.6).

The name attribute should be of type String. Go to EType in the properties window and press in the white space or the icon on the right. The Object selection should open (see Fig. 3.7). Select EString and press OK. When you do this again for the other classes, EString will be in the top of the list. Add *title, description, keywords* to Website and Webpage, and set them as EString as well.

You should have a diagram the same as the one in Fig. 3.8

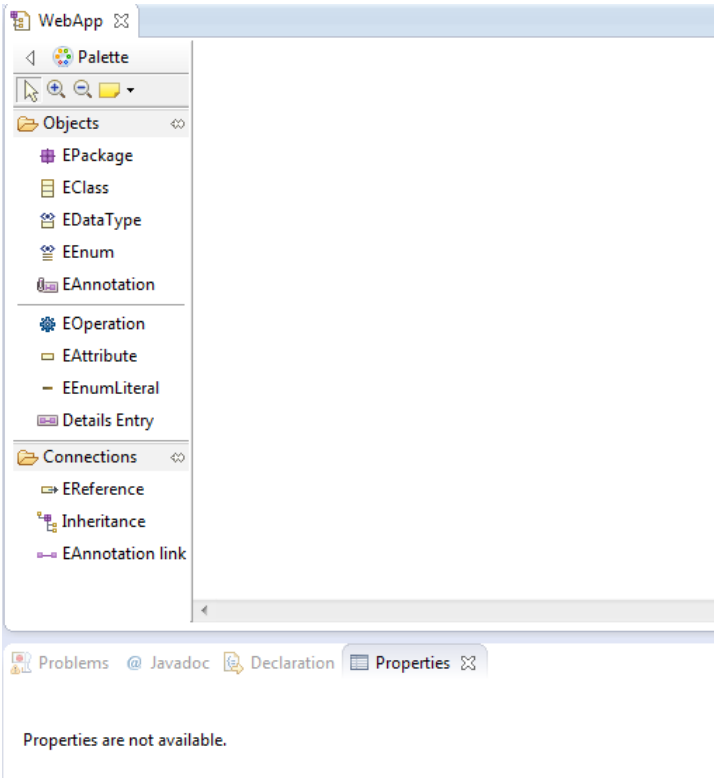


Figure 3.4.: View of the editor, with properties view in the bottom

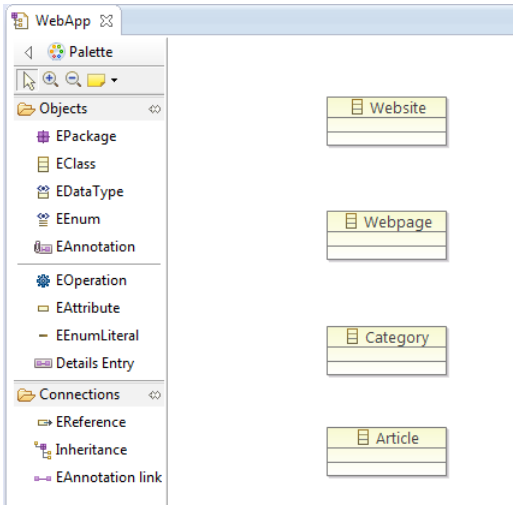


Figure 3.5.: An Ecore Diagram with 4 classes

References. Select EReference and create an arrow similar to the following picture. You first press on the Website class and drag it to the Webpage class. Name the reference as *pages*. Make sure the upper bound is set to "*" and that the "Is Containment" property is enabled (Fig. 3.9).

Create similar references from Webpage to Category and from Category to Article. In the end

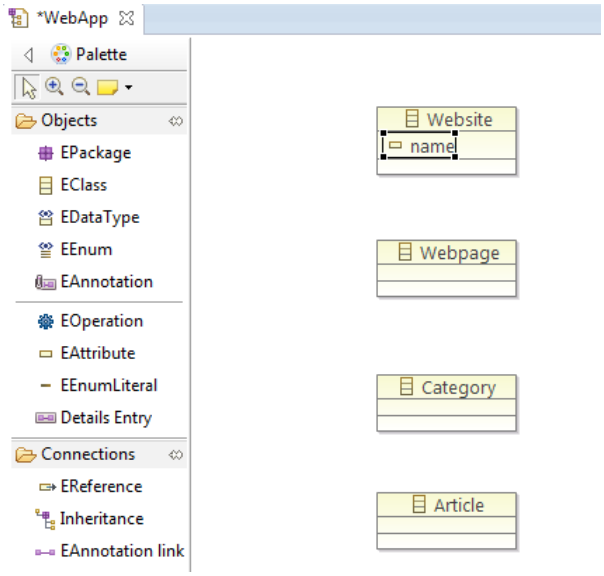


Figure 3.6.: Adding attributes with EAttribute

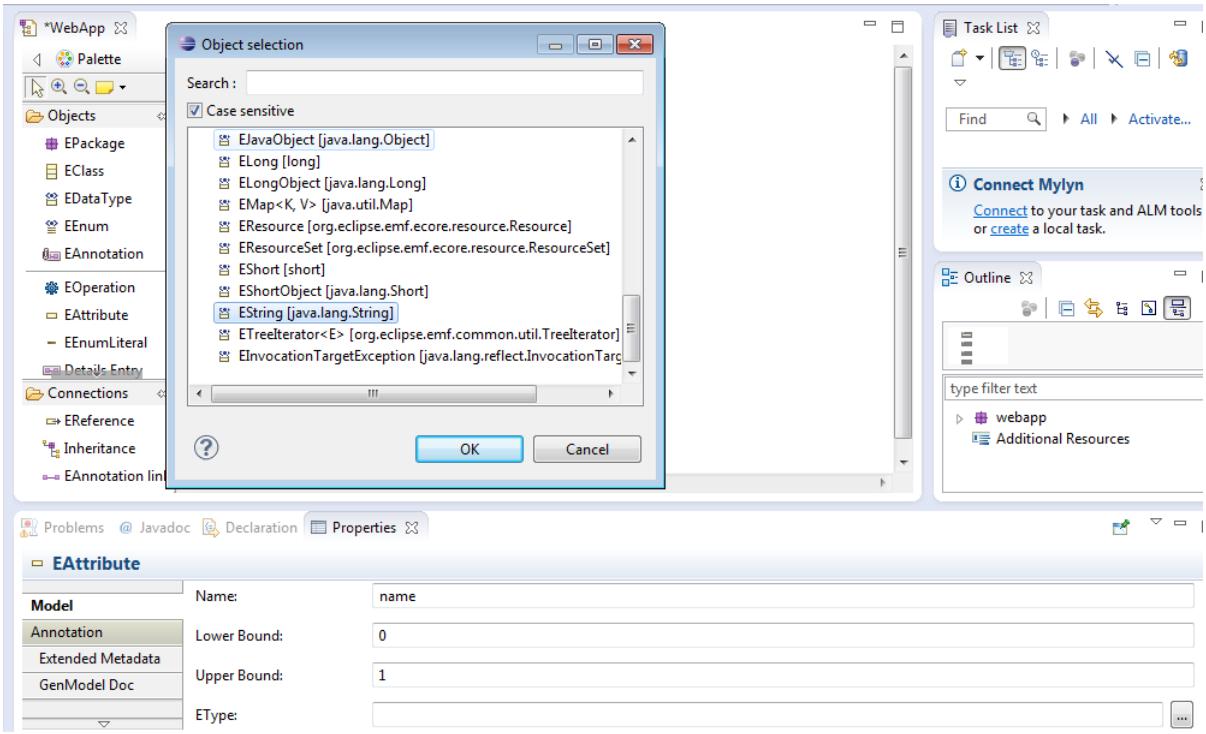


Figure 3.7.: EString type

your diagram should look like the one in Fig. 3.10

Tree view of the .ecore file. Close the diagram and open the WebApp.ecore file. The result should look like in Fig. 3.11. This view shows the actual model, so the abstract syntax of the diagram.

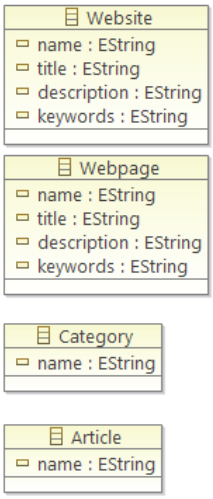


Figure 3.8.: Complete Diagram

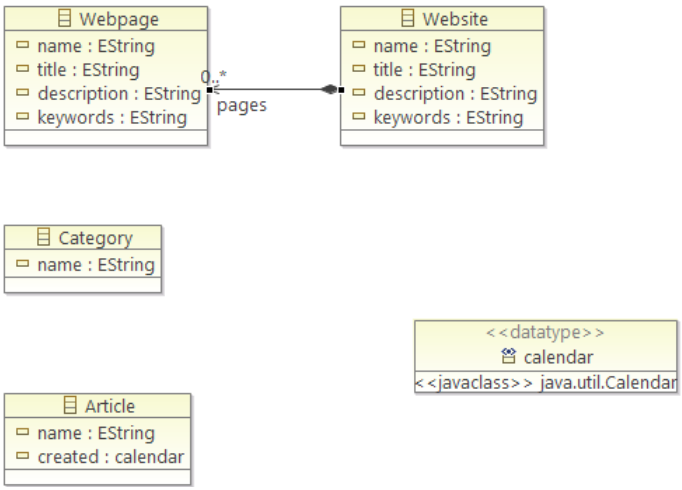


Figure 3.9.: EReferences

Now, in Eclipse, the diagram editor is rather clunky (we use it, since bringing another tool, would further complicate the course). So the recommend way to edit your models is actually to edit the tree view represenation of the model in the .ecore file. The .ecorediag file only contains the presentation layer, and it happens once in a while that it gets out of sync with the .ecore file. Then you need to recreate it (loosing all layout information, which can be quite irritating). This is why it is useful to learn to manipulate the tree view editor directly, by passing the class diagram view.

Create EMF Generator Model. Right-click your WebApp.ecore file and select New → Other → Eclipse Modeling Framework → EMF Generator model. Create the WebApp.genmodel file based on your Ecore model (see Fig. 3.12). The generator model configures the code generator. We are usually happy with the default settings, so we just create it and never

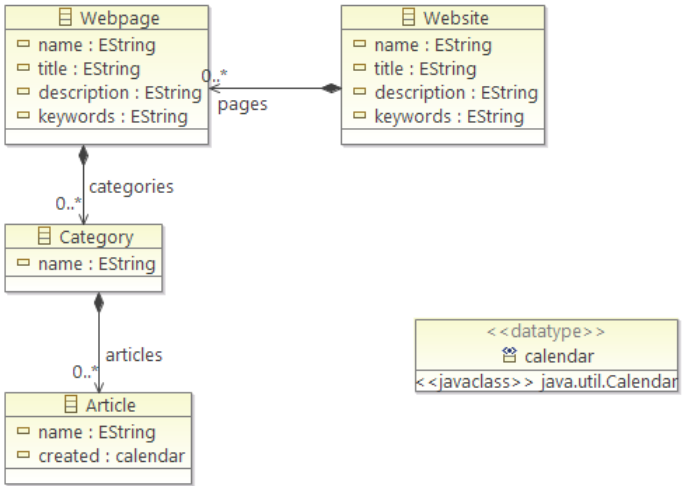


Figure 3.10.: Diagram with references

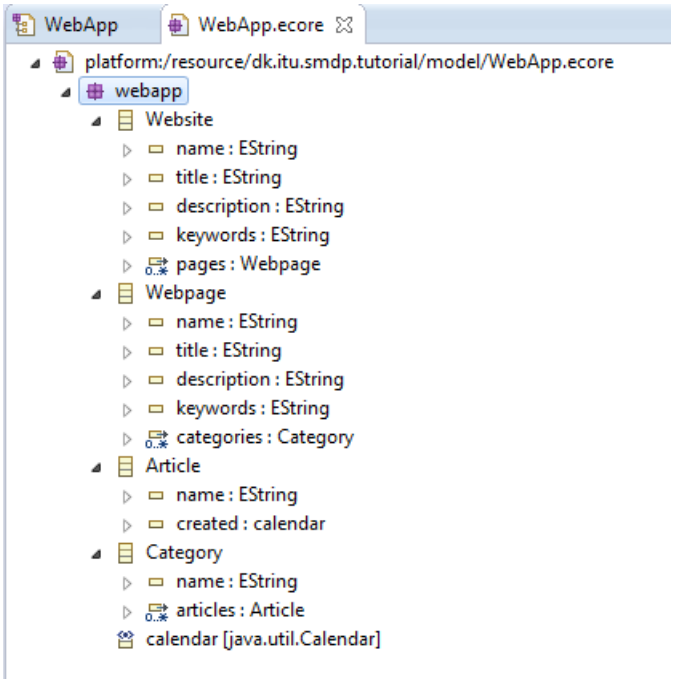


Figure 3.11.: Ecore diagram

modify it.

Select Ecore model.(see Fig. 3.13)

Select your model, press Load, then Next, and Finish.(see Fig. 3.14)

Now it should look like in Fig. 3.15. Set the base package to be `dk.itu.smdp.tutorial.WebApp.model`, and the prefix to be `MyWebApp`.

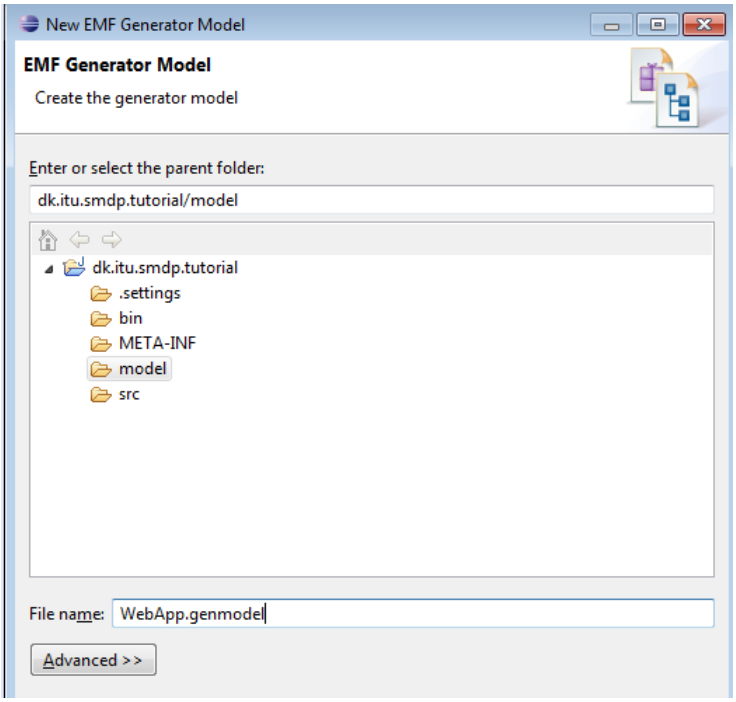


Figure 3.12.: Generator Model

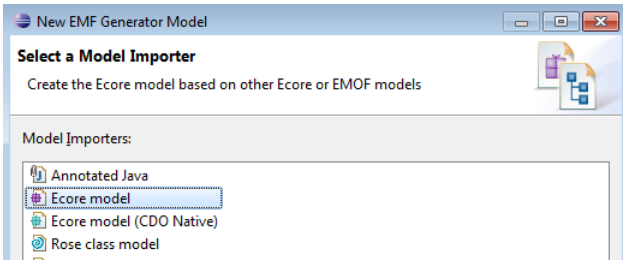


Figure 3.13.: Select Ecore model

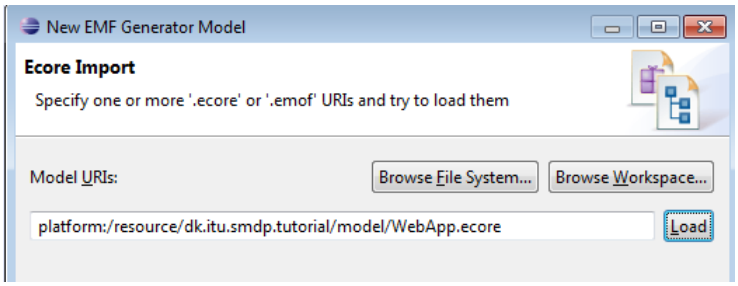


Figure 3.14.: Generator model

Creating a Dynamic Instance Up to now we have created a class diagram. Next we will create an instance of this class diagram. Open the WebApp.ecore from the package explorer. In the editor expand the blocks twice. Then right click on the Website and select *Create Dynamic Instance*. Then press Finish (see Fig. 3.16).

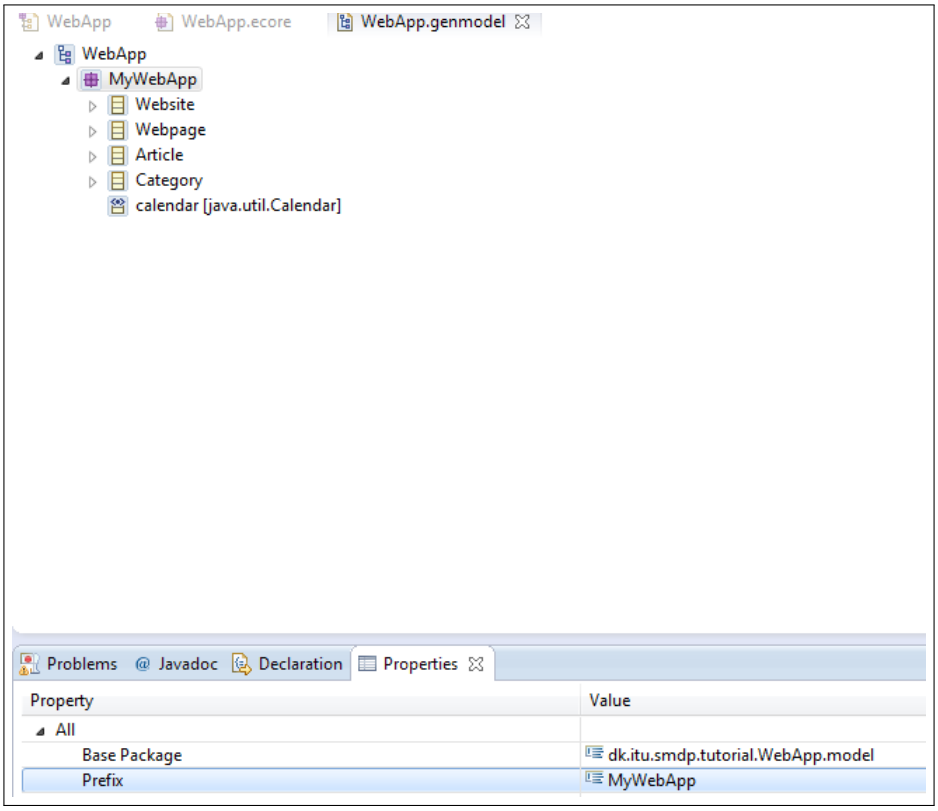


Figure 3.15.: Configuring the genmodel.

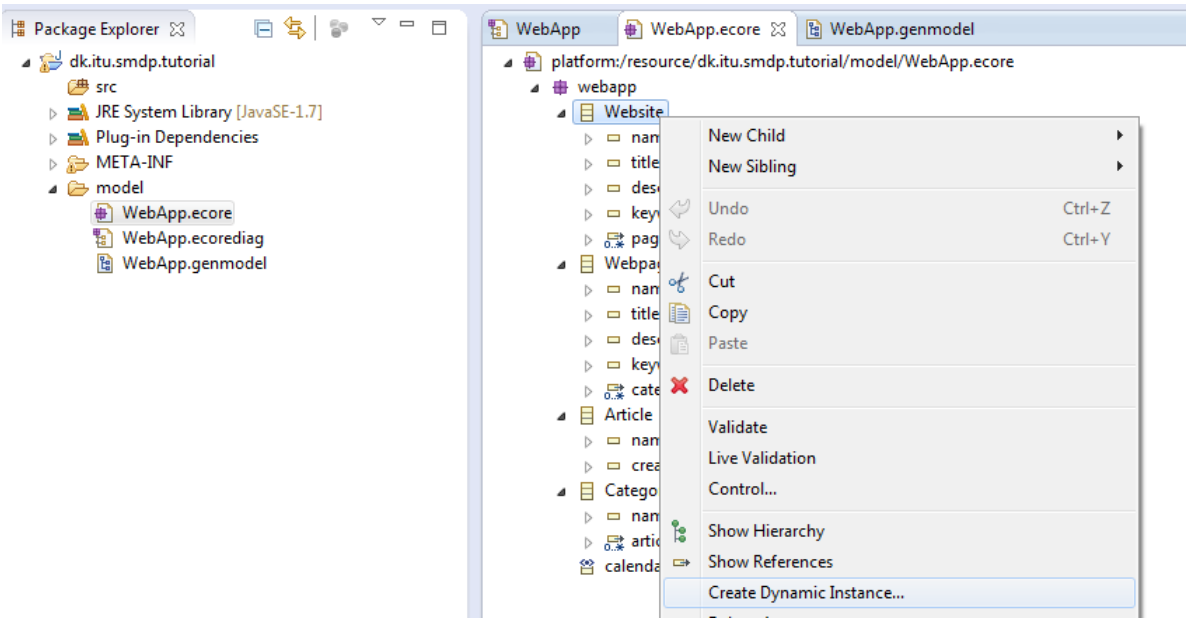


Figure 3.16.: Create Dynamic Instance

Open the Website.xmi file. In the editor expand the platform, select Website and enter values to the attributes of the class in the Properties window.(see Fig. 3.17)

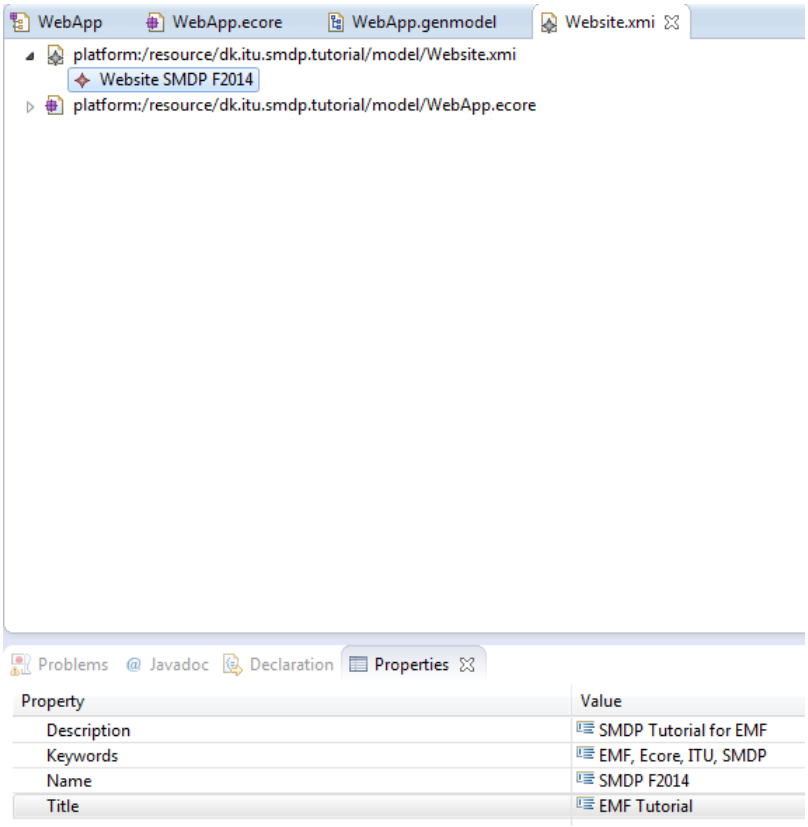


Figure 3.17.: Website.xmi

Right click on the Website class and create a child. Then create a child for Webpage. Then create a child for Category. Add values to the attributes of the classes (see Fig. 3.18).

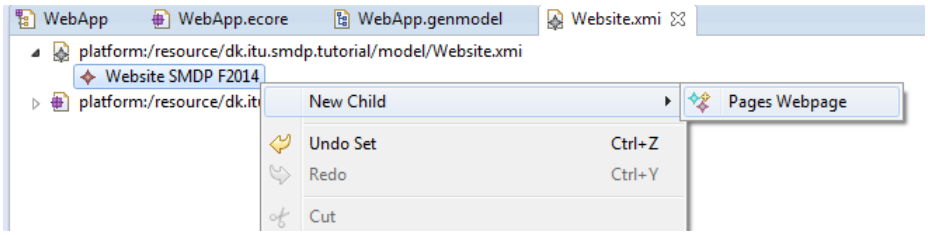


Figure 3.18.: Attribute values

4. Exercises on Concept Modeling

Objectives

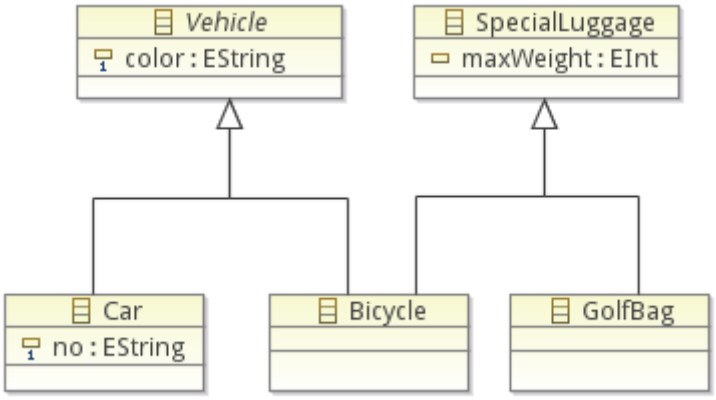
- To practice basic modeling tools of EMF
- To develop precise understanding of core class diagramming notation
- To use conceptual modeling as knowledge representation mechanism

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

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

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

Task 1. Recall the inheritance hierarchy of vehicles and luggage from the lecture:



Which of the following first order sentences hold?

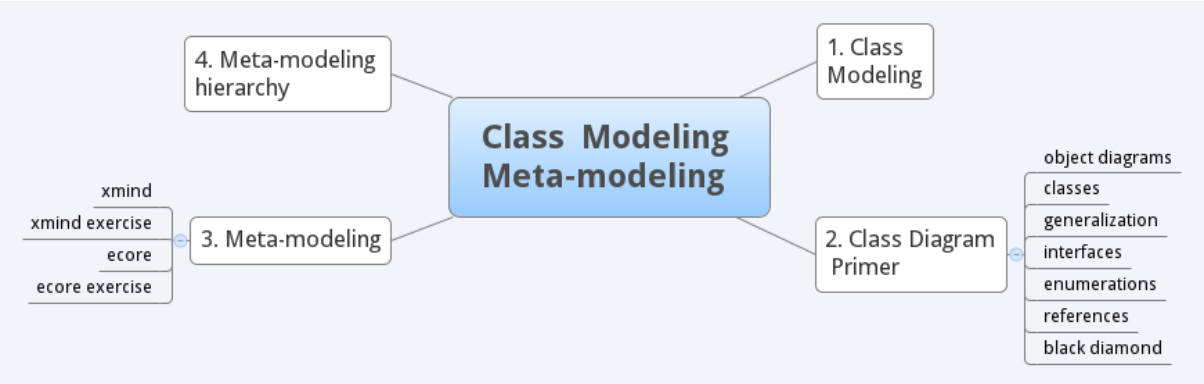
a) $\forall x.\,\mathsf{Bicycle}(x) \rightarrow \mathsf{Vehicle}(x) \wedge \mathsf{SpecialLuggage}(x)$?

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

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

d) $\forall x.\,\mathsf{Car}(x) \wedge \mathsf{Bicycle}(x) \rightarrow \mathsf{SpecialLuggage}(x)$? (tricky)

Task 2. 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.[1] 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:



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 course website.

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

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

3.1 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.

[1] http://www.vogella.com/articles/EclipseEMF/article.html

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.

3.2 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.

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

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

In my Eclipse, I needed to switch the Java compiler to 1.6 in order to make JUnit compile with no errors. It is easier if you work with a stable release, than with a snapshot code. 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.

Hand-in: Print out a screen shot of your JUnit domain model (before you throw it away ...) and hand-in to teaching assistants before the next exercise session.

[2] See also http://www.vogella.com/articles/EGit/article.html
[3] do keep it for next exercises, though

5. Object Constraint Language

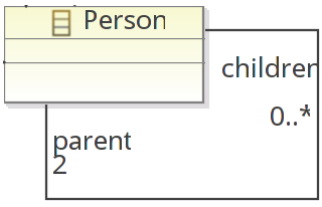
Reading: A standard reading on OCL is [29]. Please read chapter 3, which contains guidelines for writing constraints (see esp. Section 3.10 on tips and tricks).

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 [13] and the journal paper on Alloy [12] 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.

5.1. Introduction

In the last exercise session we looked at a model similar to the following diagram:



However we were unable to guarantee well formedness of this diagram easily. The problem was that we could not easily syntactically guarantee that if A is a parent of B, then the two are different (and that a person cannot be a parent of itself).

Today we look into more expressive ways of specifying such constraints easily, using the Object Constraint Language.

5.2. General Characterization of OCL

> 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**).[1]

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 [29].

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.

5.3. OCL Syntax and Semantics by Example

A usual invariant constraint has the following structure:

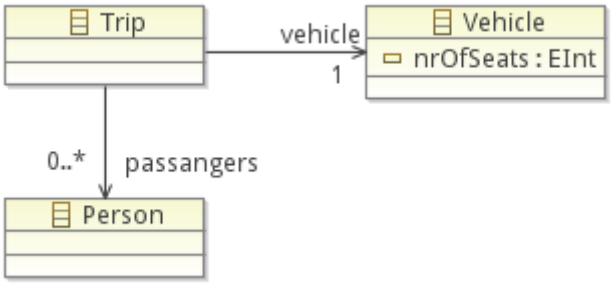
[1] Technically speaking, because OCL allows function definitions and recursion, it is more expressive than first order logics. It can for example express transitive closure. You stay within first-order logics if you do not use function definitions and recursion. Also we do not use operations (functions) in language design, which is the main application of class diagrams and OCL in this course.

```
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:

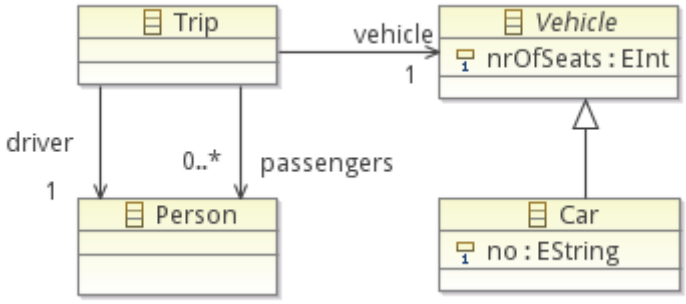


A natural integrity constraint is that the vehicle associated with the trip needs to be large enough to accommodate all the involved passengers. This constraint cannot be expressed directly in the diagrammatic language of class diagrams. It can be stated in OCL as follows:

```
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:

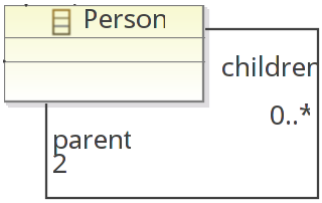
```
context Trip
inv: self.passengers->includes(self.driver)
```

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.

5.4. 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!)

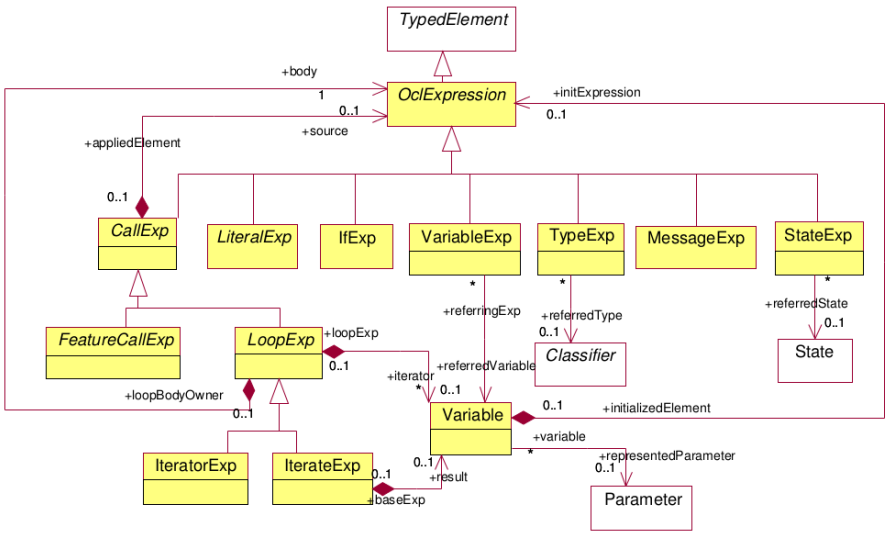


Figure 5.1.: Core OCL Model from OCL specification ver. 2.2, http://www.omg.org/spec/OCL/2.2/

- 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 */`

5.5. OCL as a Domain Specific Language

OCL is a DSL itself. Its concrete syntax is specified using a context free grammar, and its abstract syntax is specified using a meta-model. The core part of this meta-model is shown

in Fig. 5.1. Figure 5.2b shows how this metamodel fits the general layered architecture of meta-modeling, that we have shown previously.

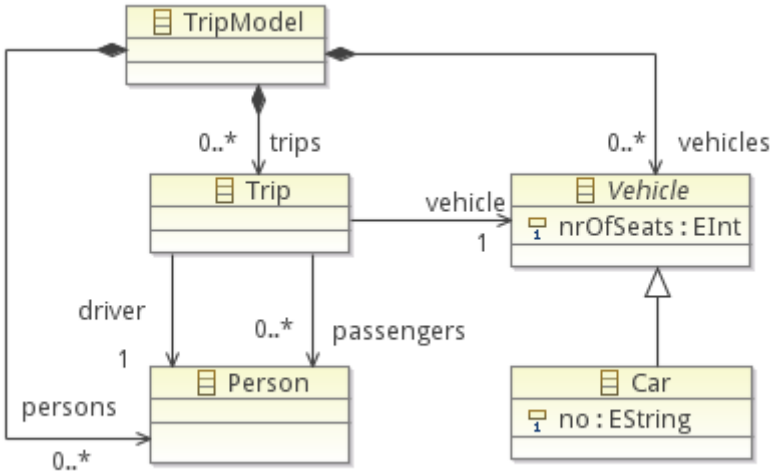
Figure 5.2a explains another point: that OCL Language, and OCL constraints semantically are similar to ecore itself, so one can also draw them one level higher. This is because the constraints written at a given level, constrain instances one level below. Thus meta-model constraints have similar semantic effect to the meta-models (M2), and thus OCL is a specification language at M3, similarly to ecore.

5.6. EMF and OCL in Practice

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. [2]

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):



[2] 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.

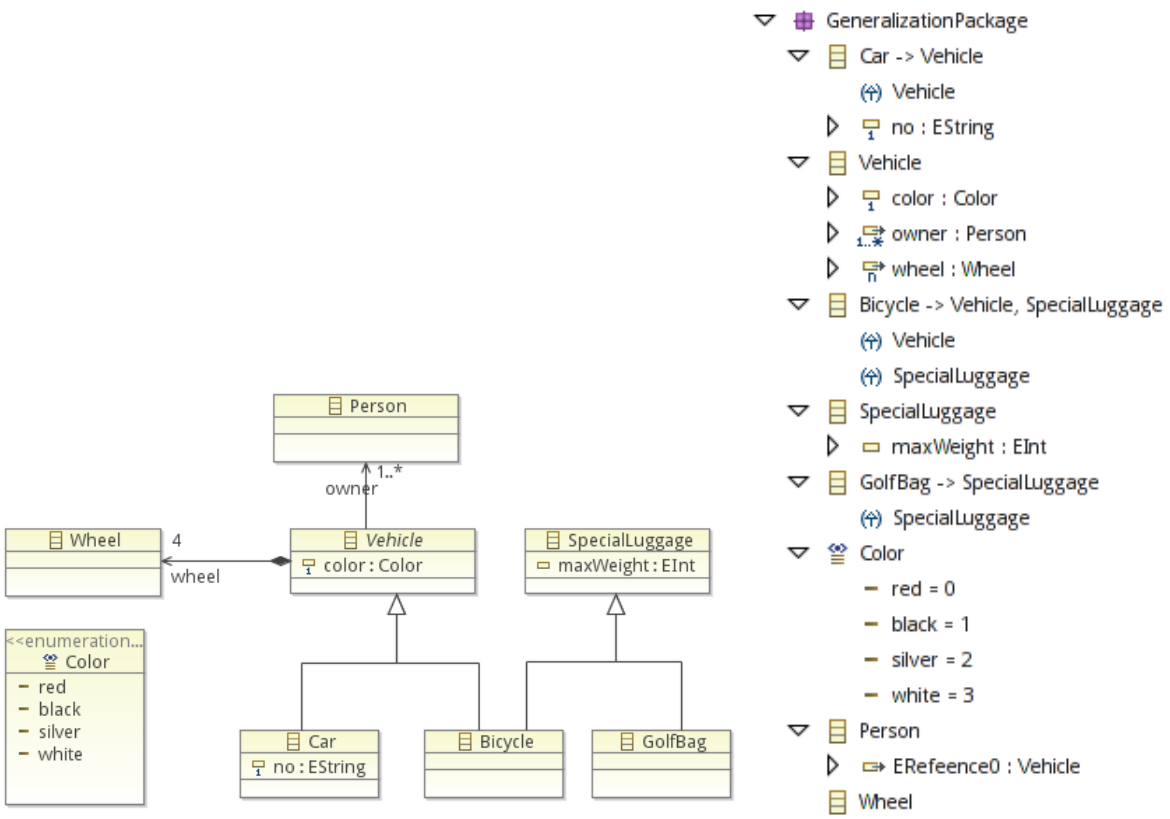
Once you created the instance dynamically, right-click on one of its elements, and open OCL console. In the console you can directly write invariant constraints. Omit the context and the `inv` keyword — instead make sure that the context element is selected in the model editor.

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.

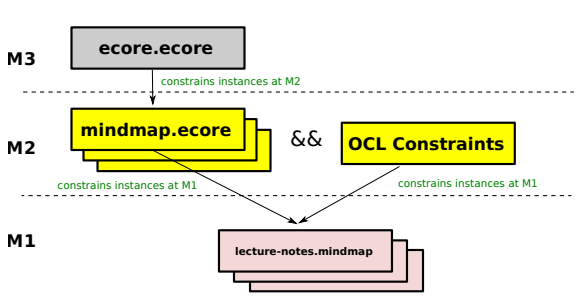
Note: Code completion works in the OCL console (ctrl-space).

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.

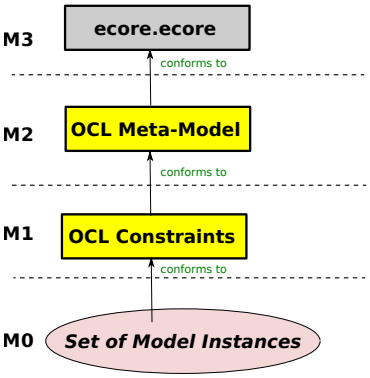


Such an editor can be automatically generated for your languages (or it can even run reflectively, as an interpreter for your meta-model). In my experience, being the main editor, the tree editor is much more stable than the diagram editor. The diagram editor stores the layout information in a diagram 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.



(a) OCL constrains instances in the same way as ecore does



(b) OCL as a language in the meta-modeling hierarchy

Figure 5.2.: Two views on OCL in the metamodeling hierarchy

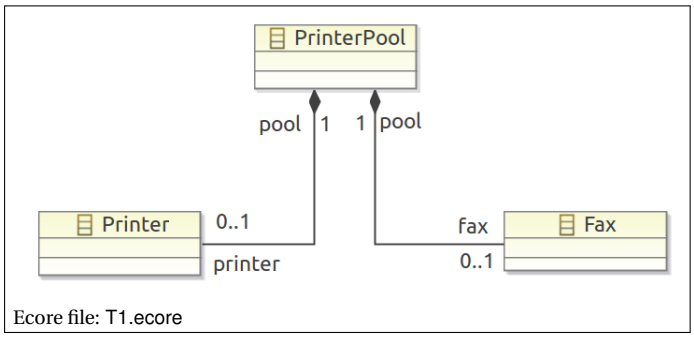
6. Exercises on Object Constraint Language

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 difference between the instance level and model level in practice (by interpreting constraints at the two levels)

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

Let us first start with an example (ecore files available from the course website):



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.

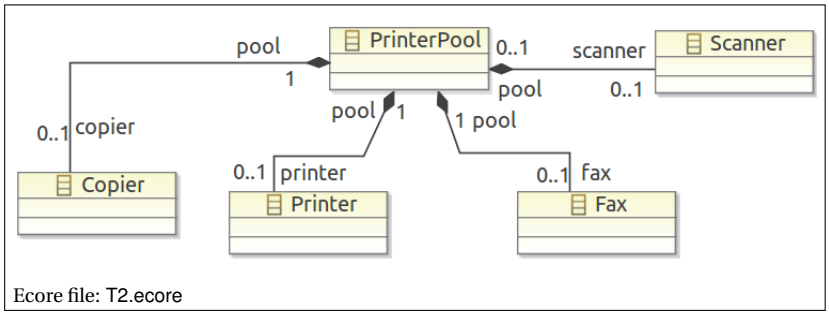
   After the .genmodel is created, open it (double click) and select "Generate Model Code" from the context menu of its root node. This will create an implementation of the model in Java. You wil find it in the src/ directory of the same Eclipse project/package.

   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).

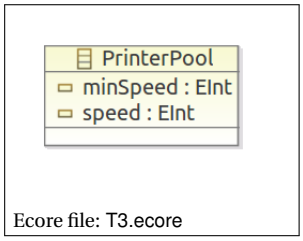
   To save time, we will not run our assertion. We just want it to typecheck and compile. Discuss with a fellow student what is more readable: your Java assertion, or the OCL constraint. Why is that?

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.



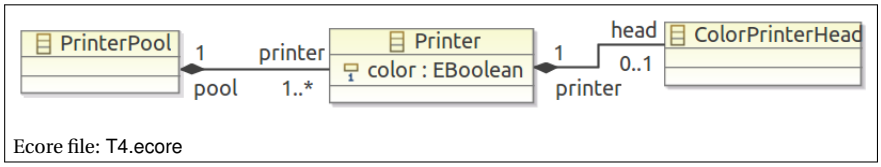
Ecore file: T2.ecore

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



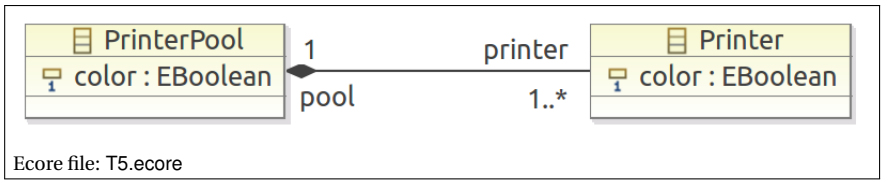
Ecore file: T3.ecore

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



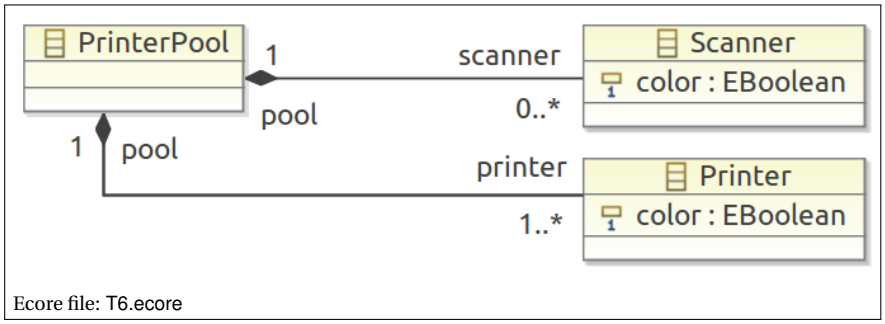
Ecore file: T4.ecore

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.*



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?

    Which 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.

Hand-In: Hand in a signed printout of your junit conceptual model (presumably the same or very similar to the one handed in last week) and several OCL constraints for this model. Deadline: before exercises the following week.

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 his book [10]. If you follow a re-engineering project, or any other project that requires DSL design, I recommend reading these: [18, 17, 22, 30, 21].

A very recent book on DSL implementation and design by one of the main evangelists in this area is *DSL Engineering* by Markus Völter available from http://dslbook.squarespace.com/ [28].

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)

These tasks became considerably easier, due to great progress in language implementation technology.

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 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.

   Fowler [10] presents a classic view on implementation of DSLs.

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.

   Language workbenches are primarily advocated in the recent book by Voelter [28].

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.

   There is a book about Microsoft DSL Tools [5].

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:

[1] The only harder way would be to manually implement a parser — nobody sane does that today.
[2] Xtext is part of Eclipse TMF, which stands for *Textual Modeling Framework*.

```
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
   )
```

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, [10] has a chapter about them, which is a good gentle starting point.

7.3. Sinatra: an Example from Ruby World

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
```

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*.

[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

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.

We explain `yield` implementing a standard higher order function `map`. A typical call to `map` in Ruby looks like: `map(l) f)`. It takes a list `l` and a code block `f`, and returns a list created from `l` by applying `f` to each of its elements. Here is an implementation of `map` in Ruby, using yield:

```
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, say, 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.*

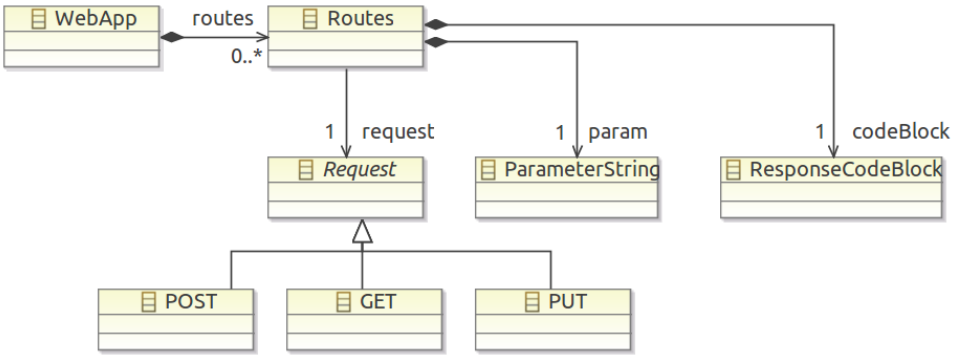
This exercise turns into a larger project, if you also wanted to integrate this with EMF, so that models are interchangeble with EMF through files. An easy way is to use textual concrete syntax, and make it precisely the same as in the XText example in the following section. If you would like to go via 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 too much. 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 use cases? 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 [17].

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 [18, 17].

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 [18, 27]. 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 [17].

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 [17]. 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. [18]

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...). [18]

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

8. Exercises on Abstract Syntax of DSLs

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 6 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.

In the upcoming exercises, we will be generating (almost) all methods that start with the "assert" prefix. Each of assert methods comes in two versions: with and without a message. For example the simplest assertion method is available in these two versions:

```
static public void assertTrue(String message, boolean condition)
```

```
static public void assertTrue(boolean condition)
```

Both versions can be generated from the same information, so we will ignore the methods without messages today.

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(expected - actual) <= delta
```

Repeat this for all the public assert methods (ignore `assertThat` for simplicity). After the task is completed you will have a number of expressions written down on paper, or in a file.

Task 2.* 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 (Task 3).

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 instantiated 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.

Try to 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 in the upcoming exercise sessions.

Hand In: Print out the class diagram of your meta-model and the syntax trees of your instances showing abstract syntax trees of expressions representing the individual assertion functions. Hand in before exercises next week.

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.

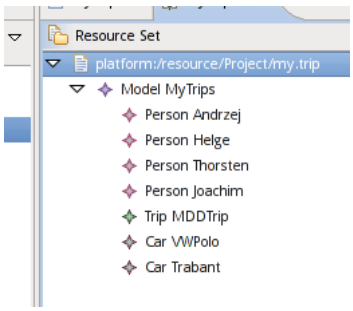
9. Textual Concrete Syntax with XText

Reading: We will talk about parsing, grammars, left-recursion, etc. The Dragon book [1] is your friend, if you are rusty on these.

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.

9.1. Introduction

A 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.

9.2. 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:

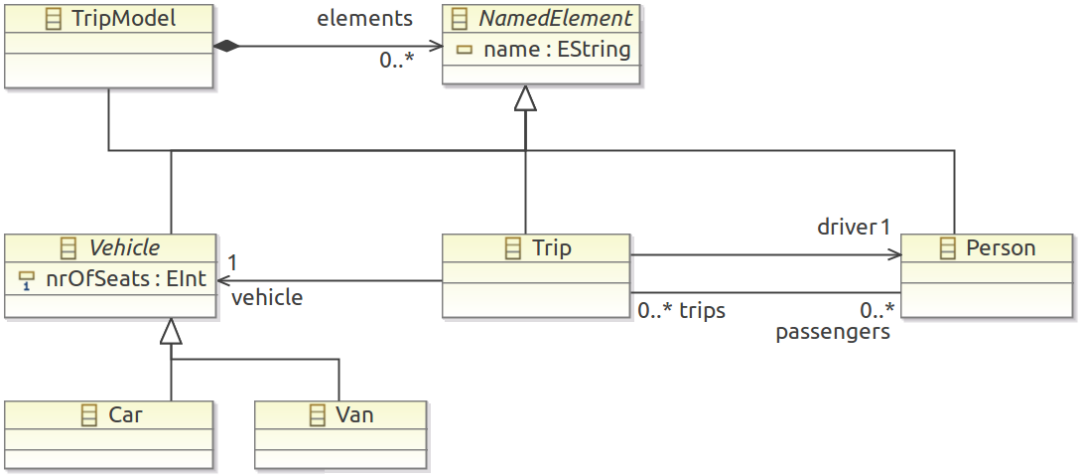


Figure 9.1.: Abstract syntax (the meta-model) of the trip language

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/pathWithinProject/trip.ecore`)

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. 9.2).

Now, we generate the xtext artifacts (just a call from the *Run As...* context menu of the grammar editor). In a few seconds we can run the generated Eclipse application (plugin) and enjoy editing of trips with text highlighting, code completion, name resolution and interactive error reporting. See Figure 9.3.

The trip model in Fig. 9.3 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 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.

An 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.*

```
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 9.2.: Automatically derived concrete syntax definition, based on the trip metamodel.

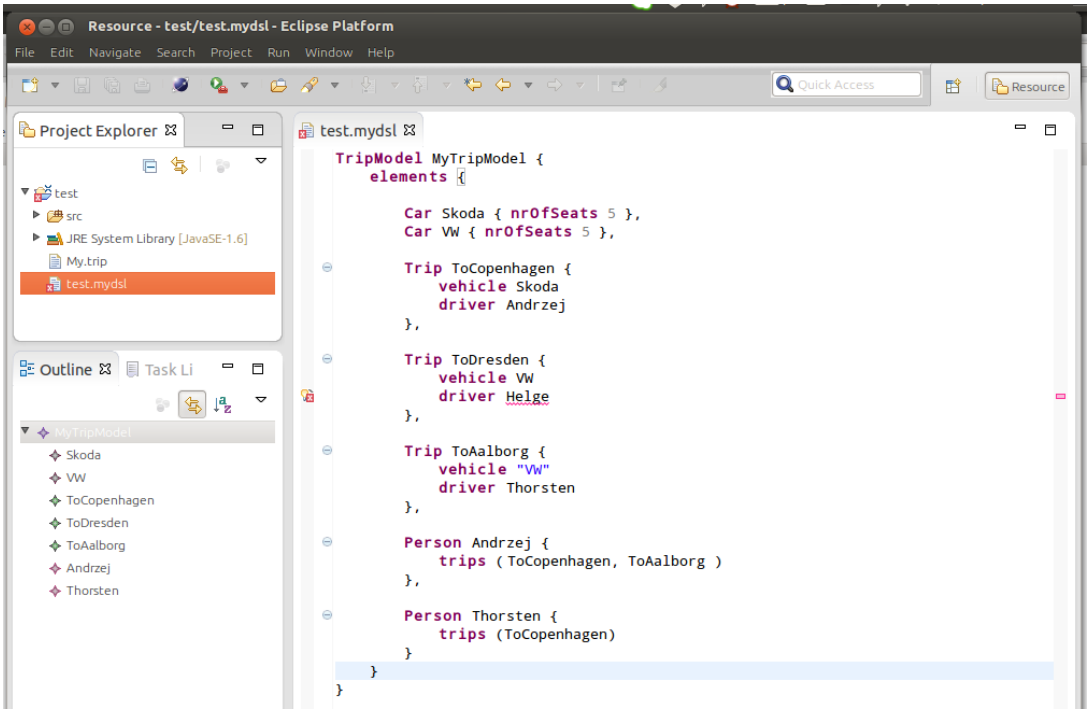


Figure 9.3.: Generated default plugin for the trip language (in action)

> *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.

9.3. 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:

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

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. 9.2 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 12.2. 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. 9.2 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.

9.4. DSL Design Guidelines (Concrete Syntax)

It is known that 75% of population prefers visual to textual syntax. Here Xtext is not very useful [18]. 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 matter experts alone 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. [18]

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. [18].

Whenever possible you should adopt existing notations used 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) [17]. 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, use descriptive terms (English words).

Always allow comments. These are very useful for evolving and experimenting with models [17].

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 [17].

Karsai and coauthors [17] give many other very practical guidance, thus I again encourage you to read their paper when designing your DSLs.

10. Exercises on Concrete Textual Syntax

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 of the most laborious exercise this semester. We reserve two weeks for work on this exercise.

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 profficiency with grammars, and/or ANTLR.

Just to inspire you, here is a fragment of my model in 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 in the upcoming weeks, 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. We will use this test harness to test generated code. (note that this task has nothing to do with Xtext and your DSL)

Hand-In: Printout of your grammar and model (Tasks 3 and 4). These are due for the exercise session two weeks later.

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.

11. Model-To-Text Transformations (M2T)

Reading: Czarnecki and Helsen [8] 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.

11.1. Model Transformations: Applications & Classification

⚠ Let us recall the distinction between syntax and semantics of a language:

> *syntax:* the principles and processes by which sentences are constructed in particular languages [Chomsky]
>
> *semantics:* the study of meanings [Merriam-Webster English dictionary]

So far we were focused with defining the syntax of DSLs in efficient ways. We worked with abstract and concrete syntax. We have seen tools that can transform syntax definitions into model editors. Now we are going to turn our attention more towards the semantics of modeling languages — so to implementing language processors.

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 [8]:

- **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.

Figure 11.1 shows an overview of a model transformation set-up.

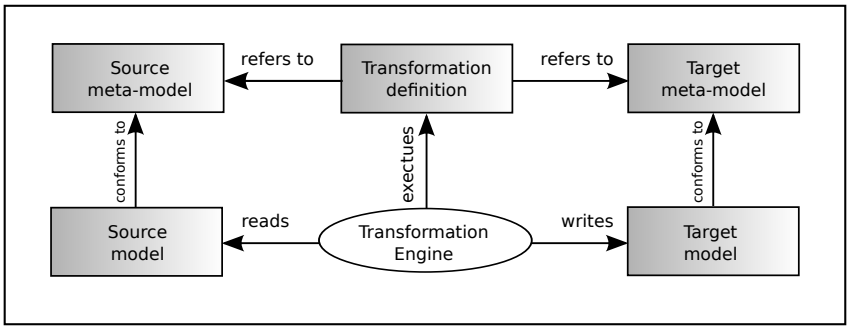


Figure 11.1.: Overview of the model transformation languages and concepts. Figure from [8]

11.2. 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 [8].

In the following we show two 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 (these templates are sometimes known under an older name, Xpand). Figure 11.2 shows the template generating a simple HTML report about our trips. Observe how concise is the template:

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

```
«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 11.2.: Xpand template generating a simple report about trips.

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:

```
«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 11.3.: Xpand template to generate simple SQL.

11.3. 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 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:

```
module episode13m2t.trip.generators.M2T

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

    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.

12. Model-To-Model Transformations (M2M)

Reading: The manual for the xtend language is available from http://www.eclipse.org/xtend

---

12.1. Implementing M2M Transformations 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 12.1.: 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 graphs 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.

12.2. Xtend at a Glance

For us, Xtend is a pragmatic language that happens to be used in TMF. 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 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).

```
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+"!"
    }
}
```

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):

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.

12.3. Implementing M2M Tx in Xtend

Figure 12.2 shows the metamodel of our trip language.

```
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;
  }
}
```

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

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 T in `tripLoose`, needs to be copied into an object of type T in `trip`. From type system's point of view these two types are different, even though the contents of objects is the same. 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
```

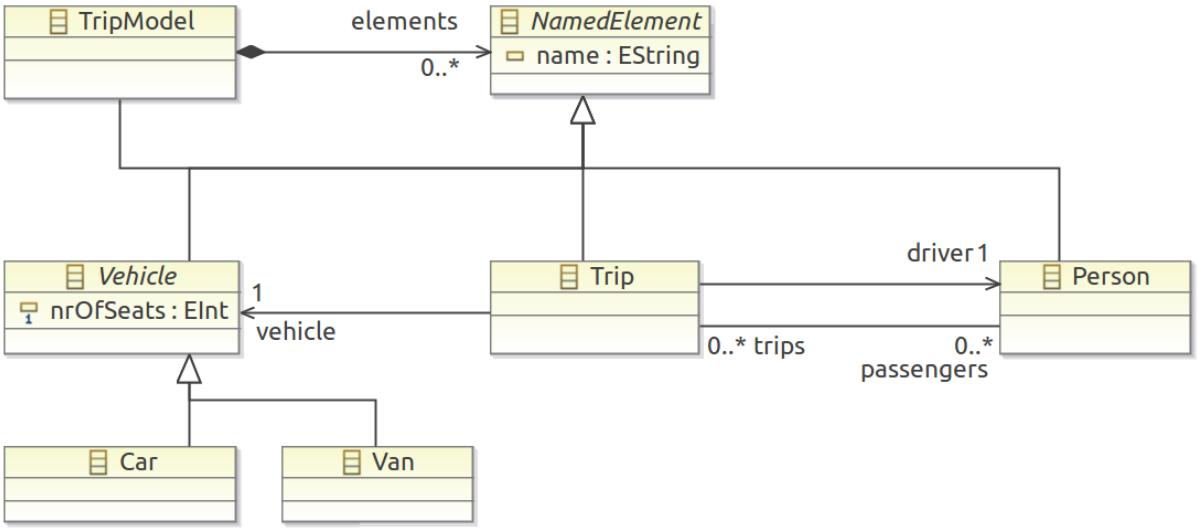


Figure 12.2.: Abstract syntax (the meta-model) of the trip language

```
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 12.3 shows the complete xtend code of a more concise version of this M2M transformation.

```
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.nrOfSeats = v.nrOfSeats
        }

    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 12.3.: An example of an M2M transformation: convert instances of TripLoose to strict Trip.

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).

12.4. 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.

13. 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. Use your own meta-model of JUnit assert in this exercise. You will also need some instances in textual and xmi syntax. The latter you will obtain by successfully running the first part of the task (converting from textual syntax). You should have got textual examples after last week's exercises.

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 on the course website.

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.

Hand-In: Printouts of your M2M and M2T transformations (so an xtend and xpand files).

14. Rule-based Transformations

Reading: We continue to rely on [8], 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

---

14.1. 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:

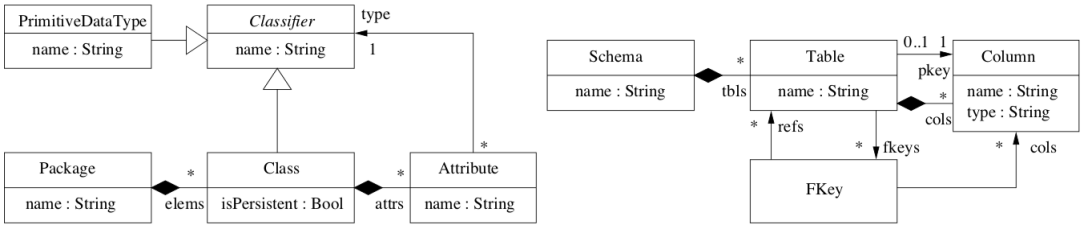


Figure 14.1.: Figure from [8]

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 [8]:

```
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)).

```
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 14.2.: Capitalize all names in ATL.

14.2. 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 14.2 shows an example of an ATL transformation capitalizing all names in an instance of the tripLoose metamodel (Sec. 14.3). 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.

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

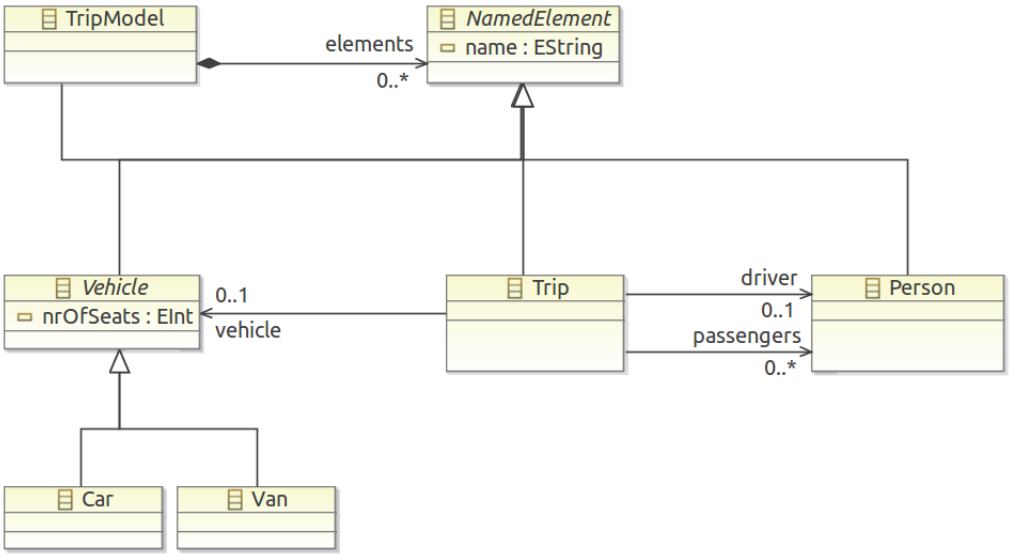


Figure 14.3.: The tripLoose meta-model from previous lectures. It is identical to Trip, but does not contain the backwards reference from Persons to Trips.

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. 14.4) 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).

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

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

```
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 14.4.: Convert tripLoose instances to trip instances using ATL.

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.

ATL 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 ...

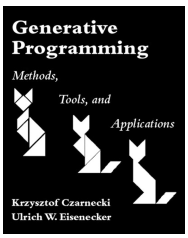
14.3. Example Exam Questions

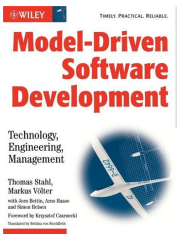
What are the main characteristics of the ATL language? What are the advantages of writing transformations in a language like ATL over Java or Xtend? Explain a simple transformation provided to you on paper.

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

15. Software Product Lines

Reading: This material is based mostly on chapter 2 of [7] and chapters 7–8 of [27]. 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).

15.1. 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]

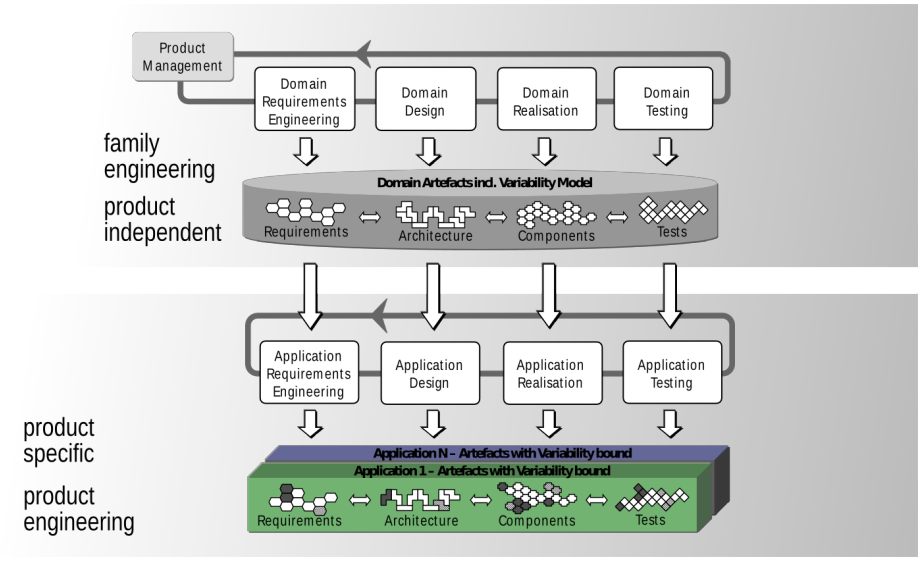


Figure 15.1.: Two main process of product line development: Domain and Application Engineering. Figure adapted from [23]

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 15.1 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. 15.1 the vertical arrows (derivation of applications from domain artifacts) are obtained using technologies presented in previous lectures.

In [27, 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).

15.2. 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," [7]
- 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.

15.3. 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 [16], 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:

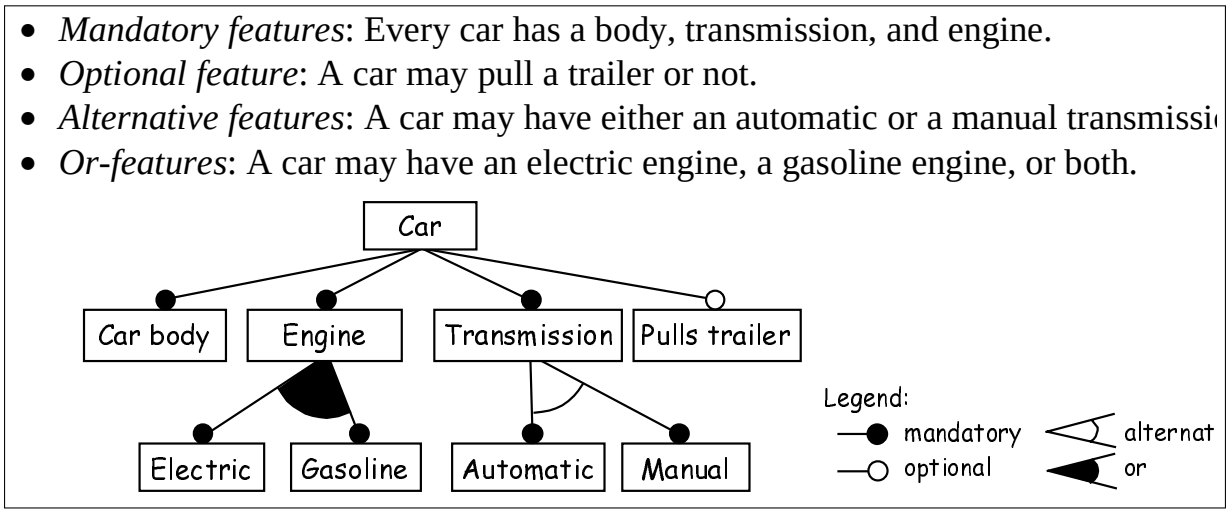


Figure 15.2.: A simple feature modeling notation, after [6]

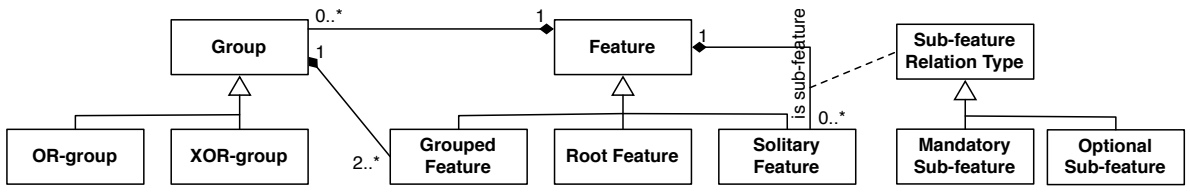


Figure 15.3.: One possible meta-model for feature modeling, after [14]

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.

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.

[16] 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 [7].

15.4. Domain Modeling with DSLs

DSLs and feature models are two different techniques that belong to the same continuum of modeling domains. See Figure 15.4.

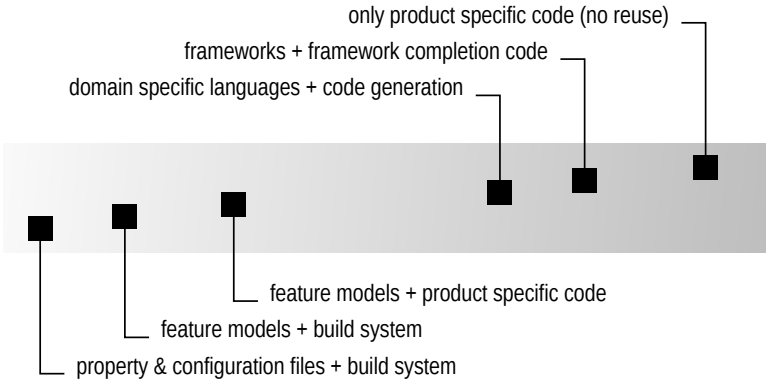


Figure 15.4.: Spectrum of Domain Modeling [7].

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.

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.

15.5. 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.

[27]

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.

15.6. 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.

Part II.

Software Engineering Projects

16. Evidence in a Software Engineering Project

Reading: The two sources that I suggest today [25, 26] cover somewhat broader topic than just evidence. They show how to construct a report that is convincing in proving its hypothesis.

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

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.

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, 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.

16.2. Matching Evidence to Claims

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

- **Empirical** projects
- **Engineering** projects
- **Theoretical** (mathematical) projects in computer science (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.

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

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...

MT Example

Question: What is The Claim?

> Model transformation (MT) technology 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 lowering the number of programmer errors; In the second case—the usability as a benchmark and as a bridge between technological spaces; In the third case—providing guidance for various use cases. In the fourth 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 [2] we systematically discuss how Clafer meets its requirements, and why it does it better than other similar languages.

16.3. 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?

16.4. Assorted Advice

Mary Shaw [25] 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 [26]. 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 [26]; *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.* [25]

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!

17. 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* [32], 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 [31]. 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.

17.1. 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

17.2. 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 [32])

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.

17.3. 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 (later in this note, we advise 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.

17.4. 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 [25] 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 [15]

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

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

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

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.

17.5. 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.

17.6. 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.

17.7. 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 [31].

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.

17.7.1. 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.

17.7.2. 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. [3]

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. [20]

17.7.3. 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)

17.7.4. 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 en- tire 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. [3]

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. [20]

17.8. Related Work

We now discuss how to collect, study and describe related literature for your project.

17.8.1. 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.

17.8.2. 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.

17.8.3. 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).

17.8.4. 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.

17.8.5. 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.**

17.9. 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.

17.10. 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.

17.11. 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.

17.12. 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.

17.13. 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)

17.14. Later This Semester

Process from now on

- **Apr 3:** experiment designed
- **Apr 10:** threats to validity
- **Apr 17:** related work
- **Apr 23:** evaluation section written
- **Apr 28:** at 23.59 a complete draft for final review
- **May 5:** reviews due
- **May 15:** final deadline (to study admin)

18. Exam Pensum & Format

18.1. Pensum (Exam Reading)

All lecture notes, slides, and exercises. Including the essential reading material linked from the notes, whenever indicated as necessary reading.

18.2. 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.

18.3. 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?

18.4. 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.

19. 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.

19.1. 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.

19.2. 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
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 and send it 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.

Part III.

Selected Readings

Below we include for your convenience the most significant papers in the area, referred from the lecture notes.

The Pragmatics of Model-Driven Development

Bran Selic, *IBM Rational Software*

Using models to design complex systems is de rigeur in traditional engineering disciplines. No one would imagine constructing an edifice as complex as a bridge or an automobile without first constructing a variety of specialized system models. 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. However, for historical reasons, models in software engineering are infrequent and, even when used, they often play a secondary role. Yet, as we shall see, the potential benefits of using models are significantly greater in software than in any other engineering discipline.

Model-driven development methods were devised to take advantage of this opportunity, and the accompanying technologies have matured to the point where they are generally useful. A key characteristic of these methods is their fundamental reliance on automation and the benefits that it brings. However, as with all new technologies, MDD's success relies on carefully introducing it into the existing technological and social mix. To that end, I cite several pragmatic criteria—all drawn from industrial experience with MDD.

Model-driven development holds promise of being the first true generational leap in software development since the introduction of the compiler. The key lies in resolving pragmatic issues related to the artifacts and culture of previous generations of software technologies.

The challenge

Software engineering is in the unfortunate position of being a new and relatively immature branch of engineering of which much is expected. Seduced by the relative ease of writing code—there is no metal to bend or heavy material to move—and compelled by relentless market pressures, software users and developers are demanding systems whose complexities often exceed our abilities to construct them.

This situation is not without precedent in the history of technology; similar situations occurred when the Industrial Revolution introduced new technologies such as steam and electrical power.[1] What seems to be unique,

0740-7459/03/$17.00 © 2003 IEEE

Many practitioners have given up all hope that significant progress will result from fundamental advances in programming technologies.

however, is how slowly software technologies have evolved to meet the obvious need for improving product reliability and productivity.

In particular, since the introduction of third-generation languages in the late 1950s, the essence of programming technology has hardly changed. Although we've introduced several new programming paradigms since then—such as structured and object-oriented programming—and much work has been done to polish the details, the level of abstraction of market-dominant programming languages has remained almost constant. An If or Loop statement in a modern programming language such as Java or C++ is not that much more potent than an If or Loop statement in early Fortran. Even promising mechanisms such as classes and inheritance, which have potential for producing higher forms of abstraction, remain underused. Objects, for example, are relegated to relatively fine-grained abstractions confined to a single address space (such as stacks, data structures, or graphic primitives) consistent with the granularity and abstraction level of the languages in which they appear.

In an industry that prides itself on its rapid advances, this apparent reluctance to move forward despite an obvious need might seem surprising. However, consider the sheer scale of investment—fiscal and intellectual—in those early-generation technologies. There are countless lines of code written in traditional programming languages that programmers must maintain and upgrade. This, in turn, creates a continuous demand for professionals who are trained in and culturally attuned to these technologies. Because of their intricate nature, attaining competency in such programming technologies requires significant investments in time and effort. This, quite understandably, fosters a conservative mindset in both individuals and corporations. Unless we properly account for such factors, no technical breakthrough is likely to succeed, regardless of how advanced and promising it might be.

Many practitioners have given up all hope that significant progress will result from fundamental advances in programming technologies; instead, they are placing their hopes on process improvements. This partly explains the current surge of interest in methods such as Extreme Programming and the Rational Unified Process.[2]

Although following a proper process is critical to any engineering endeavor's success, it's too soon to discount the possibilities that new programming technologies can achieve. After all, software development consists primarily of expressing ideas, which means that our ability to devise suitable facilities is mostly limited by our imagination rather than by unyielding physical laws. Taking advantage of this opportunity is one of the central ideas behind MDD and one of the reasons why it represents the first true generational shift in basic programming technology since the introduction of compilers.

I recognize that similar software "revolutions" have been proclaimed many times in the past but have had little or no fundamental impact in the end. Is there any reason to expect otherwise in this case? After all, MDD is based on the old idea of modeling software—a technique that has produced more than its share of skeptics.

The essentials

MDD's defining characteristic is that software development's primary focus and products are models rather than computer programs. The major advantage of this is that we express models using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain relative to most popular programming languages. This makes the models easier to specify, understand, and maintain; in some cases, it might even be possible for domain experts rather than computing technology specialists to produce systems. It also makes models less sensitive to the chosen computing technology and to evolutionary changes to that technology (the concept of *platform-independent* models is often closely connected to MDD).

Of course, if models end up merely as documentation, they are of limited value, because documentation all too easily diverges from reality. Consequently, a key premise behind MDD is that programs are automatically generated from their corresponding models.

As noted, however, both software modeling and automatic code generation have been tried before, meeting with limited success at best and mostly in highly specialized domains. But things have progressed since the early days. Aside from the fact that we now better understand how to model software, MDD is more useful today because of two key evolutionary developments: the necessary automation tech-

nologies have matured and industry-wide standards have emerged.

Automation technologies

Automation is by far the most effective technological means for boosting productivity and reliability. However, most earlier attempts at applying automation to software modeling were limited to "power-assist" roles, such as diagramming support and skeletal code generation. These are often not substantive enough to make a significant difference to productivity. For example, once the code is generated, the models are abandoned because, like all software documentation, they require scarce and expensive resources to maintain. This is why solutions based on so-called *round-trip engineering*, which automatically converts code back into model form, are much more useful. One drawback here, though, is that an automated conversion from code to model usually can't perform the kind of abstraction that a human can. Therefore, we can attain MDD's full benefits only when we fully exploit its potential for automation. This includes

- Automatically generating *complete* programs from models (as opposed to just code skeletons and fragments)
- Automatically verifying models on a computer (for example, by executing them)

Complete code generation simply means that modeling languages take on the role of implementation languages, analogous to the way that third-generation programming languages displaced assembly languages. With complete code generation, there is rarely, if ever, a need to examine or modify the generated program directly—just as there is rarely a need to examine or modify the machine code that a compiler produces.

Automatically verifying models means using a computer to analyze the model for the presence of desirable properties and the absence of undesirable ones. This can take many different forms, including formal (mathematical) analyses such as performance analysis based on queuing theory or safety-and-liveness property checking. Most often, though, it means executing (simulating) models on a computer as an empirical approach to verification. In all cases, it is critical to be able to do this on highly abstract and incomplete models that arise early in the development cycle, because this is when software designers make most of the fundamental design decisions.

The techniques and tools for doing this successfully have now reached a degree of maturity where this is practical even in large-scale industrial applications. Modern code generators and related technologies can produce code whose efficiency is comparable to (and sometimes better than) hand-crafted code. Even more importantly, we can seamlessly integrate such code generators into existing software production environments and processes. This is critical because it minimizes the disruption that occurs when MDD is deployed.

Standards

The last decade has seen the emergence of widely supported industry standards, such as those that the Object Management Group provides. The OMG is a consortium of software vendors and users from industry, government, and academia. It recently announced its Model-Driven Architecture initiative, which offers a conceptual framework for defining a set of standards in support of MDD (see www.omg.org/mda/index.htm). A key MDA standard is the Unified Modeling Language, along with several other technologies related to modeling.[3–5] In addition, other formal and de facto standards, such as various Web standards (XML, SOAP, and so forth) are also major enablers of MDD.

Standardization provides a significant impetus for further progress because it codifies best practices, enables and encourages reuse, and facilitates interworking between complementary tools. It also encourages specialization, which leads to more sophisticated and more potent tools.

Still, with all the benefits of automation and standardization, model-driven methods are only as good as the models they help us construct.

Despite all the benefits of automation and standardization, model-driven methods are only as good as the models they help us construct.

The quality of models

Models and modeling have been an essential part of engineering from antiquity (Vitruvius, a Roman engineer from the first century B.C., discusses the effectiveness of models in the world's oldest known engineering textbook[6]). Engineering models aim to reduce risk by helping us better understand both a complex problem and its potential solutions before undertaking the expense and effort of a full implementation. In

The rejection of modeling for software is ironic when you consider that software is the engineering medium best positioned to benefit from it.

contrast, large software projects typically involve great uncertainty about the design's viability until the final implementation phases—unfortunately, this is when the cost of fixing fundamental design flaws is greatest.

To be useful and effective, an engineering model must possess, to a sufficient degree, the following five key characteristics. The most important is *abstraction*. A model is always a reduced rendering of the system that it represents. By removing or hiding detail that is irrelevant for a given viewpoint, it lets us understand the essence more easily. Considering the steady demand for ever-more sophisticated functionality from our software systems, abstraction is almost the only available means of coping with the resulting complexity.

The second key characteristic is *understandability*. It isn't sufficient just to abstract away detail; we must also present what remains in a form (for example, a notation) that most directly appeals to our intuition. Understandability is a direct function of the expressiveness of the modeling form used (expressiveness is the capacity to convey a complex idea with little direct information). A good model provides a shortcut by reducing the amount of intellectual effort required for understanding. One reason why programs are not particularly expressive, even when based on languages that support sophisticated abstractions, is that they require too much detailed parsing of text to be properly understood. Classical programming statements assault the reader with a profusion of syntactical detail assembled according to intricate lexical rules. The amount of information that must be absorbed and recognized to understand linear programs is enormous and requires significant intellectual effort.

The third key characteristic of useful models is *accuracy*. A model must provide a true-to-life representation of the modeled system's features of interest.

Fourth is *predictiveness*. You should be able to use a model to correctly predict the modeled system's interesting but nonobvious properties, either through experimentation (such as by executing a model on a computer) or through some type of formal analysis. Clearly, this depends greatly on the model's accuracy and modeling form. For instance, a mathematical model of a bridge is much better at predicting the maximum allowable load on a bridge than even a very precise and detailed scale model constructed out of balsa wood.

Finally, a model must be *inexpensive*—that is, it must be significantly cheaper to construct and analyze than the modeled system.

Probably the main reason why software modeling techniques had limited success in the past is that the models often failed to meet one or more of the criteria just listed. In particular, the techniques tended to be weak in terms of accuracy (which also meant that the models weren't very useful for prediction). In part, this is because it wasn't always clear how the concepts used to express the models mapped to the underlying implementation technologies such as programming language constructs, operating system functions, and so forth. This semantic gap was exacerbated if the modeling language was not precisely defined, leaving room for misinterpretation.

Also, because the models weren't formally connected to the actual software, there was no way of ensuring that the programmers followed the design decisions captured in a model during implementation. They would often change design intent during implementation—thereby invalidating the model. Unfortunately, because the mapping between models and code is imprecise and the code is difficult to comprehend, such digressions would remain undetected and could easily lead to downstream integration and maintenance problems. (Changing design intent isn't necessarily a bad thing, but it is bad if the change goes unobserved.) Given these difficulties, many software practitioners felt that software models were untrustworthy, merely adding useless overhead to their already difficult task.

This rejection of modeling for software is particularly ironic when you consider that software is the engineering medium best positioned to benefit from it. This is because it is possible to gradually evolve an abstract software model into the final product through a process of incremental refinement, without requiring a change in skills, methods, concepts, or tools. The advantage of this is self-evident: there are no risk-laden semantic gaps to overcome when transferring a design into production. Model accuracy is guaranteed because the model eventually becomes the system that it was modeling. Furthermore, it is particularly conducive to an incremental iterative development style that is optimal when building complex engineering systems, because there

are no conceptual discontinuities that preclude backtracking. This unique property of software models is another cornerstone of MDD.

The pragmatics

Many software practitioners, when first faced with the notion of MDD, express concern about the technical difficulties involved in translating models into code. Will the code be fast enough and compact enough? Will it be a correct rendering of design intent? These, of course, are the very same questions that were asked when compilers were introduced more than 40 years ago. Although they were valid questions to ask at the time, it is worth noting that hardly anyone questions compiler technology these days because it is quite mature and extensively proven in practice.

In fact, experience with MDD in industrial settings indicates that code efficiency and correctness, although very important, are not the top-priority or even the most technically challenging issues associated with MDD. In fact, most standard techniques used in compiler construction can also be applied directly to model-based automatic code generation.

Model-level observability

Like all compilers, automatic code generators are idiosyncratic and often generate program code that, as a result of various internal optimizations, is not easily traceable to the original model. Thus, if an error is detected in the generated program, finding the place in the model that must be fixed either at compile time or runtime might be difficult. In traditional programming languages, we expect compilers to report errors in terms of the original source code and, for runtime errors, we now expect a similar capability from our debuggers.

The need for such facilities for models is even greater because the semantic gap between the modeling language's high-level abstractions and the implementation code is wider. This means that model-level error reporting and debugging facilities (in essence, "decompilers") must accompany practical automatic code generators. Otherwise, the practical difficulties encountered in diagnosing problems could be significant enough to nullify much of MDD's advantage. Programmers faced with fixing code that they don't understand will easily break it and will likely be discouraged from relying on models in the future.

This is a particularly important factor to consider for model-driven systems that are based on the notion of customizable transformation "templates." Such templates capture rules for translating models into corresponding code. By exposing these to developers, it is possible to streamline the generated code for specific target environments. This is a highly appealing and useful capability, but it must be matched by a similar facility for specifying inverse transformations, or model observability will most certainly be an issue.

Also related to model-level observability are two other critical facilities: model mergers and model difference tools. These tools are typically an integral part of configuration management systems that help us track different versions of the same model. Model merging tools merge two or more possibly overlapping models into one. In contrast to source-code merging used for traditional text-based programming languages, a model-level merge is much more complex because it requires a deeper understanding of the more complex semantics of the modeling language. The result must be a well-formed model. Furthermore, the tools must report any problems in a form that is meaningful to the modeler. Model difference tools identify the difference between two models (usually two different versions of the same model). They too must work at a semantically meaningful level.

Model executability

One of the fundamental ways that we learn is through experimentation—that is, through model execution (David Harel compares models that can't be executed to cars without engines). One important advantage of executable models is that they provide early direct experience with the system being designed. (When learning a new programming language, we are always inspired by the successful run of our first trivial "hello world" program. Simple as it is, that experience raises our confidence level and gives us a reference point for further exploration.) The intuition gained through experimentation is the difference between mere formal knowledge and understanding.

A common experimental scenario with executable models involves refining some high-risk aspect of a candidate solution down to a relatively fine level of detail, while other parts of the model remain sketchy or even undefined. This means that even incomplete models

Experience with MDD in industrial settings indicates that code efficiency and correctness are not the primary challenges of MDD.

It is now common knowledge that modern optimizing compilers can outperform most practitioners when it comes to code efficiency.

should be executable, as long as they are well formed. It also requires suitable runtime system support: the ability to start, stop, and resume a model run at any meaningful point; to "steer" it in the desired direction by simulating inputs at appropriate points in space and time; and to easily attach automated instrumentation packages to it. Finally, it is also extremely useful for developers to be able to execute a model in a simulation environment (for example, on a development workstation), on the actual target platform, or—and this is the most useful—on some combination of the two.

Efficiency of generated code

As mentioned earlier, one of the first questions asked about MDD is how the automatically generated code's efficiency compares to handcrafted code. This is nothing new; the same question was asked when compilers were first introduced. The concern is the same as before: humans are creative and can often optimize their code through clever tricks in ways that machines cannot. Yet, it is now common knowledge that modern optimizing compilers can outperform most practitioners when it comes to code efficiency. Furthermore, they do it much more reliably (which is another benefit of automation).

We can decompose code efficiency into two separate areas: performance (throughput) and memory utilization. Current model-to-code generation technologies can generate code with both performance and memory efficiency factors that are, on average, within 5 to 15 percent (better or worse) of equivalent manually crafted systems. And, of course, we can only expect the situation to improve as the technology evolves. In other words, for the vast majority of applications, efficiency is not an issue.

Still, there might be occasional critical cases where manually crafted code might be necessary in specific parts of the model. Such hot spots are often used as an excuse to reject MDD altogether, even when it involves a very small portion of the complete system—the proverbial "baby and bathwater" scenario. A useful MDD system will allow for seamless and overhead-free embedding of such critical elements.

Scalability

MDD is intended for—and most beneficial in—large-scale industrial applications. This sometimes involves hundreds of developers working on hundreds of different but related parts of a model, and the tools and methods must scale up to such situations.

The important metrics of concern here are *compilation time* and *system size*. We can divide compilation time into two separate parts: *full-system generation time* and the *turnaround time* for small incremental changes. Perhaps surprisingly, the latter is much more important because of its greater impact on productivity. Namely, small changes are far more frequent during development than full-system recompiles. Therefore, if a small, localized change requires regenerating a disproportionately large part of the code, development can slow to an unacceptable pace. This is particularly true in the latter phases of the development cycle, when programmers make many small changes as they fine-tune the system. To keep this overhead low, it is crucial for the code generators to have sophisticated change impact analysis capabilities that minimize the amount of code regeneration.

We can divide the system generation process into two phases. First, code generators translate the model into a program in some programming language and then compile the program using standard compilers for that language. After compiling the code, they link it to the appropriate libraries in the usual way. With modern automatic code generation technology, the compilation phase is significantly longer. Typically, compilation is an order of magnitude longer than code generation. This means that the overhead of automatic code generation is almost negligible compared to the usual overhead of compilation.

Regarding size, the largest systems developed to date using full MDD techniques have involved hundreds of developers working on models that translate into several million lines of standard programming language.

Integration with legacy environments and systems

A prudent and practical way to introduce new technology and techniques into an existing production environment is to apply them to a smaller-scale project such as a relatively low-profile extension to some legacy system. This implies not only that the new software must work within legacy software but also that the development process and development environment used to produce it must be integrated into the legacy process and legacy development environment.

This is not only a question of mitigating risk

but also of leveraging previous (usually significant) investments into such processes and environments. For example, a useful MDD tool should be able to exploit a range of different compilers, build utilities, debuggers, code analyzers, and software versioning control systems rather than requiring the purchase of new ones. Furthermore, this type of integration should work "out of the box" and should generally not require custom "glue" code and tool expertise. Fortunately, most legacy development tools have evolved along similar lines, supporting similar usage paradigms so that it is usually possible to construct MDD tools that can access these capabilities in a generic fashion.

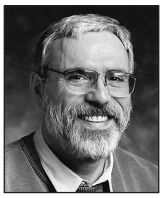
Last but not least, an MDD project must be able to take advantage of legacy code libraries and other legacy software. These often capture domain-specific knowledge garnered over many years and often represent an organization's prime intellectual property. This can be accomplished either using customizable code generators or by allowing direct calls to such utilities from within the model. For example, a model that uses Java to specify the details of actions along a statechart transition can simply make the appropriate Java calls without any intervening translation or having to go through a layer interface.

MDD's success is not predicated only on resolving obvious technical issues such as defining suitable modeling languages and automatic code generation. Our experience with these methods in industrial environments on large-scale software projects clearly indicates that solving the unique pragmatic issues described in this article is at least equally, if not more, important.[7] Unless the experience of applying MDD is acceptable from the day-to-day perspective of the individual practitioner and project manager, it will be rejected despite its obvious potential for yielding major productivity and reliability benefits. Fortunately, over the past decade, numerous commercial vendors have developed tools that address these issues successfully. The time for MDD has come.

References

1. R. Pool, *Beyond Engineering: How Society Shapes Technology,* Oxford Univ. Press, 1997.
2. P. Kruchten, *The Rational Unified Process,* Addison-Wesley, 1999.
3. *Unified Modeling Language,* ver. 1.4, Object Management Group, 2002.
4. *Meta-Object Facility (MOF),* ver. 1.4, Object Management Group, 2002; www.omg.org/cgi-bin/doc?formal/2002-04-03.
5. *Common Warehouse Metamodel (CWM) Specification,* ver. 1.1, Object Management Group, 2003; www.omg.org/cgi-bin/doc?formal/03-03-02.
6. Vitruvius, *The Ten Books on Architecture,* Dover Publications, 1960.
7. B. Selic, G. Gullekson, and P.W. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.

About the Author



Bran Selic is principal engineer at IBM Rational Software in Kanata, Ontario, Canada. He is also cochair of the OMG task force that is finalizing the UML 2.0 modeling language standard. He received his Magister Ing. Degree in systems theory and Dipl. Ing. Degree in electrical engineering from the University of Belgrade, Yugoslavia. He is a member of the IEEE and ACM. Contact him at IBM Rational Software, 770 Palladium Dr., Kanata, Ontario, Canada K2V 1C8; bselic@ca.ibm.com.

Design Guidelines for Domain Specific Languages

Gabor Karsai
Institute for Software Integrated Systems
Vanderbilt University
Nashville, USA

Holger Krahn
Software Engineering Group
Department of Computer Science
RWTH Aachen, Germany

Claas Pinkernell
Software Engineering Group
Department of Computer Science
RWTH Aachen, Germany

Bernhard Rumpe
Software Engineering Group
Department of Computer Science
RWTH Aachen, Germany

Martin Schindler
Software Engineering Group
Department of Computer Science
RWTH Aachen, Germany

Steven Völkel
Software Engineering Group
Department of Computer Science
RWTH Aachen, Germany

ABSTRACT

Designing a new domain specific language is as any other complex task sometimes error-prone and usually time consuming, especially if the language shall be of high-quality and comfortably usable. Existing tool support focuses on the simplification of technical aspects but lacks support for an enforcement of principles for a good language design. In this paper we investigate guidelines that are useful for designing domain specific languages, largely based on our experience in developing languages as well as relying on existing guidelines on general purpose (GPLs) and modeling languages. We defined guidelines to support a DSL developer to achieve better quality of the language design and a better acceptance among its users.

1. INTRODUCTION

Designing a new language that allows us to model new technical properties in a simpler and easier way, describe or implement solutions, or to describe the problem resp. requirements in a more concise way is one of the core challenges of computer science. The creation of a new language is a time consuming task, needs experience and is thus usually carried out by specialized language engineers. Nowadays, the need for new languages for various growing domains is strongly increasing. Fortunately, also more sophisticated tools exist that allow software engineers to define a new language with a reasonable effort. As a result, an increasing number of DSLs (Domain Specific Languages) are designed to enhance the productivity of developers within specific domains. However, these languages often fit only to a rather specific domain problem and are neither of the quality that they can be used by many people nor flexible enough to be easily adapted for related domains.

During the last years, we developed the frameworks MontiCore [13] and GME [2] which support the definition of domain specific languages. Using these frameworks we designed several DSLs for a variety of domains, e.g., a textual version of UML/P notations [17] and a language based on function nets in the automotive domain [5]. We experienced that the design of a new DSL is a difficult task because different people have a varying perception of what a "good" language actually is.

This of course also depends on the taste of the developer respectively the users, but there are a number of generally acceptable guidelines that assist in language development, making it more a systematic, methodological task and less an intellectual ad-hoc challenge. In this paper we summarize, categorize, and amend existing guidelines as well as add our new ones assuming that they improve design and usability of future DSLs.

In the following we present general guidelines to be considered for both textual and graphical DSLs with main focus is on the former. The guidelines are discussed sometimes using examples from well-known programming languages or mathematics, because these languages are known best. Depending on the concrete language and the domain these guidelines have to be weighted differently as there might be different purposes, complexity, and number of users of the resulting language. For example, for a rather simple configuration language used in only one project a timely realization is usually more important than the optimization of its usability. Therefore, guidelines must be sometimes ignored, altered, or enforced. Especially quality-assurance guidelines can result in an increased amount of work.

While we generally focus in our work on DSLs that are specifically dedicated to modeling aspects of (software) systems, we believe that these guidelines generally hold for any DSL that embeds a certain degree of complexity.

1.1 Literature on Language Design

For programming languages, design guidelines have been intensively discussed since the early 70s. Hoare [8] introduced simplicity, security, fast translation, efficient object code, and readability as general criteria for the design of good languages. Furthermore, Wirth [22] discussed several guidelines for the design of languages and corresponding compilers. The rationale behind most of the guidelines and hints of both articles can be accepted as still valid today, but the technical constraints have changed dramatically since the 70s. First of all, computer power has increased significantly. Therefore, speed and space problems have become less important. Furthermore, due to sophisticated tools (e.g., parser generators) the implementation of accompanying tools is often not a necessary part of the language development any more. Of course, both articles

concentrate on programming languages and do not consider the greater variety of domain specific languages.

More recently, authors have also discussed the design of domain specific modeling languages. General principles for modeling language design were introduced in [14]. These include simplicity, uniqueness, consistency, and scalability, on which we will rely later. However, the authors did not discuss how these higher level principles can be achieved. In [12] certain aspects of the DSL development are explained and some guidelines are introduced. More practical guidelines for implementing DSLs are given in [10]. These focus on how to identify the necessary language constructs to generate full code from models. The authors explain how to provide tool support with the MetaEdit+ environment. [20] explains 12 lessons learned from DSL experiments that can help to improve a DSL. Although more detailed discussions on explicit guidelines are missing, these lessons embed documented empirical evidence – a documentation that many other discussions, including ours do not have. In [16] the authors introduce a toolset which supports the definition of DSLs by checking their consistency with respect to several objectives. Language designers can select properties of their DSL to be developed and the system automatically derives other design decisions in order to gain a consistent language definition. However, the introduced criteria cover only a subset of the decisions to be made and hence, cannot serve as the only criteria for good language design. Quite the contrary, to our experience many design guidelines cannot be translated in automatic measures and thus cannot be checked by a tool.

1.2 Categories of DSL Design Guidelines

The various design guidelines we will discuss below, can be organized into several categories. Essentially, these guidelines describe techniques that are useful at different activities of the language development process, which range from the domain analysis to questions of how to realize the DSL to the development of an abstract and a concrete syntax including the definition of context conditions. An alignment of guidelines with the language development activities and the developed artifacts has the advantage that a language designer can concentrate on the respective subset of the guidelines at each activity. This should help identifying and realizing the desired guidelines. Therefore, we decided for a development phase oriented classification and identified the following categories:

Language Purpose discusses design guidelines for the early activities of the language development process.

Language Realization introduces guidelines which discuss how to implement the language.

Language Content contains guidelines which focus on the elements of a language.

Concrete Syntax concentrates on design guidelines for the readable (external) representation of a language.

Abstract Syntax concentrates on design guidelines for the internal representation of a language.

For each of these categories we will discuss the design guidelines we found useful. Please be aware that the subsequently discussed guidelines sometimes are in conflict with each other and the language developer sometimes has to balance them accordingly. Additionally, semantics is explicitly not listed as a separate step as it should be part of the entire development process and therefore has an influence on all of the categories above.

2. DSL DESIGN GUIDELINES

2.1 Language Purpose

Language design is not only influenced by the question of what it needs to describe, but equally important what to do with the language. Therefore, one of the first activities in language design is to analyze the aim of the language.

Guideline 1: *"Identify language uses early."* The language defined will be used for at least one task. Most common uses are: documentation of knowledge (only) and code generation. However, there are a lot more forms of usage: definition or generation of tests, formal verification, automatic analysis of various kinds, configuration of the system at deployment- or run-time, and last but increasingly important, simulation.

An early identification of the language uses have strong influence on the concepts the language will allow to offer. Code generation for example is not generally feasible when the language embeds concepts of underspecification (e.g., nondeterministic Statecharts). Even if everything is designed to be executable, there are big differences regarding the overhead necessary to run certain kinds of models. If efficient execution on a small target machine is necessary (e.g., mobile or car control device) then high-level concepts must be designed for optimized code generations. For simulation and validation of requirements however, efficiency plays a minor role.

Guideline 2: *"Ask questions."* Once the uses of a language have been identified it is helpful to embed these forms of language uses into the overall software development process. People/roles have to be identified that develop, review, and deploy the involved programs and models. The following questions are helpful for determining the necessary decisions: Who is going to model in the DSL? Who is going to review the models? When? Who is using the models for which purpose?

Based thereon, the question after whether the language is too complex or captures all the necessary domain elements can be revisited. In particular, appropriate tutorials for the DSL users in their respective development process should now be prepared.

Guideline 3: *"Make your language consistent."* DSLs are typically designed for a specific purpose. Therefore, each feature of a language should contribute to this purpose, otherwise it should be omitted. As an illustrative example we consider a platform independent modeling language. In this language, all features should be platform independent as well. This design principle was already discussed in [14].

2.2 Language Realization

When starting to define a new language, there are several options on how to realize it. One can implement the DSL from scratch or reuse and extend or reduce an existing language, one can use a graphical or a textual representation,

and so on. We have identified general hints which have to be taken into account for these decisions.

Guideline 4: *"Decide carefully whether to use graphical or textual realization."* Nowadays, it is common to use tools supporting the design of graphical DSLs such as the Eclipse Modeling Framework (EMF) or MetaEdit+. On the other hand, there exist sophisticated tools and frameworks like MontiCore or xText for text-based modeling. As described in [6], there are a number of advantages and disadvantages for both approaches. Textual representations for example usually have the advantage of faster development and are platform and tool independent whereas graphical models provide a better overview and ease the understanding of models. Therefore, advantages and disadvantages have to be weighted and matched against end users' preferences in order to make a substantiated decision for one of the realizations. From this point on, a more informed decision can be made for a concrete tool to realize the language based on their particular features and the intended use of the language. Comparisons can be found in [21] or [3].

Guideline 5: *"Compose existing languages where possible."* The development of a new language and an accompanying toolset is a labor-intensive task. However, it is often the case that existing languages can be reused, sometimes even without adaptation. A good example for language reuse is OCL: it can be embedded in other languages in order to define constraints on elements expressed in the hosting language.

The most general and useful form of language reuse is thus the unchanged embedding of an existing language into another language. A more sophisticated approach is to have predefined holes in a host language, such that the definition of a new language basically consists of a composition of different languages. For textual languages this compositional style of language definitions is well understood and supported by sophisticated tools such as [11] which also assists the composition of appropriate tools.

However, according to the seamlessness principle [14], the concepts of the languages to be composed need to fit together. In the UML, the object oriented paradigm underlies both class diagrams and Statecharts which therefore fit well together. Additionally, when composing languages care must be exercised to avoid confusion: similar constructs with different semantics should be avoided.

Guideline 6: *"Reuse existing language definitions."* If the language cannot be simply composed from some given language parts, e.g., by language embedding as proposed in guideline 5, it is still a good idea to reuse existing language definitions as much as possible. In [18] more possible realization strategies, such as language extension or language specialization are analyzed. This means, taking the definition of a language as a starter to develop a new one is better than creating a language from scratch. Both the concrete and the abstract syntax will benefit from this form of reuse. The new language then might retain a look-and-feel of the original, thus allowing the user to easily identify familiar notations. Looking at the abstract syntax of existing languages, one can identify "language pattern" (quite similar to design pattern), which are good guidelines for language design. For example, expressions, primary expressions, or statements have quite a common pattern in all languages.

Only if there is no existing language/notation or the disadvantages do not allow using the strategies mentioned above, a standalone realization should be considered. The websites of parser generators like Antlr [1] or Atlantic Zoo [19] are a good starting point for reusing language definitions.

Guideline 7: *"Reuse existing type systems."* A DSL used for software development often comprises and even extends either a property language such as OCL or an implementation language such as Java. As described in [8], the design of a type system for such a language is one of the hardest tasks because of the complex correlations of name spaces, generic types, type conversions, and polymorphism.

Furthermore, an unconventional type system would be hard for users to adopt as well. Therefore, a language designer should reuse existing type systems to improve comprehensibility and to avoid errors that are caused by misinterpretations in an implementation. Furthermore, it is far more economical to use an existing type system, than developing a new one as this is a labor intensive and error-prone task. A well-documented object-oriented type system can be tailored to the needs of the DSL or even an implemented reusable type system can be used (e.g. [4]).

2.3 Language Content

One main activity in language development is the task of defining the different elements of the language. Obviously, we cannot define in general which elements should be part of a language as this typically depends on the intended use. However, the decisions can be guided by some basic hints we propose in this Section.

Guideline 8: *"Reflect only the necessary domain concepts."* Any language shall capture a certain set of domain artifacts. These domain artifacts and their essential properties need to be reflected appropriately in the language in a way that the language user is able to express all necessary domain concepts. To ensure this, it is helpful to define a few models early to show how such a reflection would look like. These models are a good basis for feedback from domain experts which helps the developer to validate the language definition against the domain. However, when designing a language not all domain concepts need to be reflected, but only those that contribute to the tasks the language shall be used for.

Guideline 9: *"Keep it simple."* Simplicity is a well known criterion which enhances the understandability of a language [8, 14, 22]. The demand for simplicity has several reasons. First, introducing a new language in a domain produces work in developing new tools and adapting existing processes. If the language itself is complex, it is usually harder to understand and thus raises the barrier of introducing the language. Second, even when such a language is successfully introduced in a domain, unnecessary complexity still minimizes the benefit the language should have yielded. Therefore, simplicity is one of the main targets in designing languages. The following more detailed Guidelines 10, 11, and 12 will show how to achieve simplicity.

Guideline 10: *"Avoid unnecessary generality."* Usually, a domain has a finite collection of concepts that should be reflected in the language design. Statements like "maybe we can generalize or parameterize this concept for future changes in the domain" should be avoided as they unneces-

sarily complicate the language and hinder a quick and successful introduction of the DSL in the domain. Therefore, this guideline can also be defined as "design only what is necessary".

Guideline 11: *"Limit the number of language elements."* A language which has several hundreds of elements is obviously hard to understand. One approach to limit the number of elements in a language for complex domains is to design sublanguages which cover different aspects of the systems. This concept is, e.g., employed by the UML: different kinds of diagrams are used for special purposes such as structure, behavior, or deployment. Each of them has its own notation with a limited number of concepts.

A further possibility to limit the number of elements of a language is to use libraries that contain more elaborated concepts based on the concepts of the basic language and that can be reused in other models. Elements which were previously defined as part of the language itself can then be moved to a model in the library (compare, e.g., I/O in Pascal vs. C++). Furthermore, users can extend a library by their own definitions and thus, can add more and more functionality without changing the language structure itself. Therefore, introducing a library leads to a flexible, extensible, and extensive language that nevertheless is kept simple. On the other hand, a language capable of library import and definition of those elements must have a number of appropriate concepts embedded to enable this (e.g., method and class definitions, modularity, interfaces - whatever this means in the DSL under construction). This principle has successfully been applied in GPL design where the languages are usually small compared to their huge standard libraries.

Guideline 12: *"Avoid conceptual redundancy."* Redundancy is a constant source of problems. Having several concepts at hand to describe the same fact allows users to model it differently. The case of conceptual richness in C++ shows that coding guidelines then usually forbid a number of concepts. E.g., the concept of classes and structs is nearly identical, the main difference is the default access of members which is `public` for structs and `private` for classes. Therefore, classes and structs can be used interchangeably within C++ whereas the slight difference might be easily forgotten. So, it should be generally avoided to add redundant concepts to a language.

Guideline 13: *"Avoid inefficient language elements."* One main target of domain specific modeling is to raise the level of abstraction. Therefore, the main artifacts users deal with are the input models and not the generated code. On the other hand, the generated code is necessary to run the final system and more important, the generated code determines significant properties of the system such as efficiency. Hence, the language developer should try to generate efficient code.

Furthermore, efficiency of a model should be transparent to the language user and therefore should only depend on the model itself and not on specific elements used within the model. Elements which would lead to inefficient code should be avoided already during language design so that only the language user is able to introduce inefficiency [8]. For example, in Java there is no operator to get all instances of one class as this would increase memory usage and operating time significantly. However, this functionality can be implemented by a Java user if needed.

2.4 Concrete Syntax

Concrete syntax has to be chosen well in order to have an understandable, well structured language. Thus, we concentrate on the concrete syntax first and will deal with the abstract syntax later.

Guideline 14: *"Adopt existing notations domain experts use."* As [20] says, it is generally useful to adopt whatever formal notation the domain experts already have, rather than inventing a new one.

Computer experts and especially language designers are usually very practiced in learning new languages. On the contrary, domain experts often use a language for a longer time and do not want to learn a new concrete syntax especially when they already have a notation for a certain problem. As already mentioned, it is often the case that the introduction of a DSL makes new tools and modified processes necessary. Inventing a new concrete syntax for given concepts would raise the barrier for domain experts. Thus, existing notations should be adopted as much as possible. E.g., queries within the database domain should be defined with SQL instead of inventing a new query language. Even if queries are only part of a new language to be defined SQL could be embedded within the new language.

In case a suitable notation does not already exist, the new language should be adopted as close as possible to other existing notations within the domain or to other common used languages. A good example for commonly accepted languages are mathematical notations like arithmetical expressions [8].

Guideline 15: *"Use descriptive notations."* A descriptive notation supports both learnability and comprehensibility of a language especially when reusing frequently-used terms and symbols of domain or general knowledge. To avoid misinterpretation it is highly important to maintain the semantics of these reused elements. For instance, the sign "+" usually stands for addition or at least something semantically similar to that whereas commas or semicolons are interpreted as separators. This applies to keywords with a widely-accepted meaning as well. Furthermore, keywords should be easily identifiable. It is helpful to restrict the number of keywords to a few memorizable ones and of course, to have a keyword-sensitive editor.

A good example for a descriptive notation is the way how special character like Greek letters are expressed in Latex. Instead of using a Unicode-notation each letter can be expressed by its name (`\alpha` for α, `\beta` for β, and so on).

Guideline 16: *"Make elements distinguishable."* Easily distinguishable representations of language elements are a basic requirement to support understandability. In graphical DSLs, different model elements should have representations that exhibit enough syntactic differences to be easily distinguishable. Different colors as the only criteria may be counterproductive, e.g., when printed in black and white. In textual languages usually keywords are used in order to separate kinds of elements. These keywords have to be placed in appropriate positions of the concrete syntax, as otherwise readers need to start backtracking when "parsing" the text [8, 22]. The absence of keywords is often based on efficiency for the writer. But this is a very weak reason because models are much more often read than written and therefore to be designed from a readers point of view.

Guideline 17: *"Use syntactic sugar appropriately."* Languages typically offer syntactic sugar, i.e., elements which do not contribute to the expressiveness of the language. Syntactic sugar mainly serves to improve readability, but to some extent also helps the parser to parse effectively. Keywords chosen wisely help to make text readable. Generally, if an efficient parser cannot be implemented, the language is probably also hard to understand for human readers.

However, an overuse of the addition of syntactic sugar distracts, because verbosity hinders to see the important content directly. Furthermore, it should be kept in mind that several forms of syntactic sugar for one concept may hinder communication as different persons might prefer different elements for expressing the same idea.

Nevertheless the introduction of syntactic sugar can also improve a language, e.g., the enhanced for-statement in Java 5 is widely accepted although it is conceptually redundant to a common for-statement. This is a conflict to guideline 12, but the frequency of occurrence of common for-statements in Java legitimates a more effective alternative of this notation.

Guideline 18: *"Permit comments."* Comments on model elements are essential for explaining design decisions made for other developers. This makes models more understandable and simplifies or even enables collaborative work. So a widely accepted standard form of grouped comments, like `/* ... */`, and line comments, like `// ...` for textual languages or text boxes and tooltips for graphical languages should be embedded.

Furthermore, specially structured comments can be used for further documentation purposes as generating HTML-pages like Javadoc. In [8] it is mentioned that the "purpose of a programming language is to assist in the documentation of programs". Therefore we recommend that every DSL should allow a user to generally comment at various parts of the model. If desired, the language may even contain the definition of a comment structure directly, thus enforcing a certain style of documentation.

Guideline 19: *"Provide organizational structures for models."* Especially for complex systems the separation of models in separate artifacts (files) is inevitable but often not enough as the number of files would lead to an overflowed model directory. Therefore, it is desirable to allow users to arrange their models in hierarchies, e.g., using a package mechanism similar to Java and store them in various directories.

As a consequence, the language should provide concepts to define references between different files. Most commonly "`import`" is used to refer to another name space. Imports make elements defined in other DSL artifacts visible, while direct references to elements in other files usually are expressed by qualified names like "`package.File.name`". Sometimes one form of import isn't enough and various relations apply which have to be reflected in the concrete syntax of the language.

Guideline 20: *"Balance compactness and comprehensibility."* As stated above, usually a document is written only once but read many times. Therefore, the comprehensibility of a notation is very important, without too much verbosity. On the other hand, the compactness of a language is still a worthwhile and important target in order to achieve effectiveness and productivity while writing in the language. Hence a short notation is more preferable for frequently used elements rather than for rarely used elements.

Guideline 21: *"Use the same style everywhere."* DSLs are typically developed for a clearly defined task or viewpoint. Therefore, it is often necessary to use several languages to specify all aspects of a system. In order to increase understandability the same look-and-feel should be used for all sublanguages and especially for the elements within a language. In this way the user can obtain some kind of intuition for a new language due to his knowledge of other ones. For instance, it is hardly intuitive if curly braces are used for combining elements in one language and parentheses in another. Additionally, a general style can also assist the user in identifying language elements, e.g., if every keyword consists of one word and is written in lower case letters.

A conflicting example is the embedment of OCL. One the one hand it is possible to adapt the OCL syntax to the enclosing language to provide the same syntactic style in both languages. On the other hand different OCL styles impede the comprehensibility of OCL, what endorses the use of a standard OCL syntax.

Guideline 22: *"Identify usage conventions."* Preferably not every single aspect should be defined within the language definition itself to keep it simple and comprehensible (see guideline 11). Furthermore, besides syntactic correctness it is too rigid to enforce a certain layout directly by the tools. Instead, usage conventions can be used which describe more detailed regulations that can, but need not be enforced.

In general, usage conventions can be used to raise the level of comprehensibility and maintainability of a language. The decision, whether something goes as a usage convention or within a language definition is not always clear. So, usage conventions must be defined in parallel to the concrete syntax of the language itself. Typical usage conventions include notation of identifiers (uppercase/lowercase), order of elements (e.g. attributes before methods), or extent and form of comments. A good example for code conventions for a programming language can be found in [9].

2.5 Abstract Syntax

Guideline 23: *"Align abstract and concrete syntax."* Given the concrete syntax, the abstract syntax and especially its structure should follow closely to the concrete syntax to ease automated processing, internal transformations and also presentation (pretty printing) of the model.

In order to align abstract and concrete syntax three main principles apply: First, elements that differ in the concrete syntax need to have different abstract notations. Second, elements that have a similar meaning can be internally represented by reusing concepts of the abstract syntax (usually through subclassing). This is more a semantics-based decision than a structurally based decision. Third, the abstract notation should not depend on the context an element is used in but only on the element itself. A pretty bad example for context-dependent notations is the use of "`=`" as assignment in OCL-statements (let-construct) and as equality in OCL-expressions. Here, the semantics obviously differs whilst the syntax is equal.

Furthermore, the use of a transformation engine usually also requires an understanding of the internal structure of a language, which is related to the abstract syntax. Therefore,

the user to some extent is exposed to the internal structure of the language and hence needs an alignment between his concrete representations and the abstract syntax, where the transformations operate on.

Alignment of both versions of syntax and the seamlessness principle discussed in [14] assures that it is possible to map abstractions from a problem space to concrete realizations in the solution space. For a domain specific language the domain is then reflected as directly as possible without much bias, e.g., of implementation or executability considerations.

Guideline 24: *"Prefer layout which does not affect translation from concrete to abstract syntax."* A good layout of a model can be used to simplify the understanding for a human reader and is often used to structure the model. Nevertheless, a layout should be preferred which does not have any impact on the meaning of the model, and thus, does not affect the translation of the concrete to the abstract syntax and the semantics. As an example, this is the case for computer languages where the program structure is achieved by indentation. From a practical point of view, line separators, tabs, and spaces are often treated differently depending on editors and platforms and are usually difficult to distinguish by a human reader. If these elements gain a meaning, developers have to be much more cautious and a collaborative development requires more effort. For graphical languages a well-known bad example is the twelve o'clock semantics in Stateflow [7] where the order of the placement of transitions can change the behavior of the Statechart. To simplify the usage of DSLs, we recommend that the layout of programs doesn't affect their semantics.

Guideline 25: *"Enable modularity."* Nowadays, systems are very complex and thus, hard to understand in their entirety. One main technique to tackle complexity is modularization [15] which leads to a managerial, flexible, comprehensible, and understandable infrastructure. Furthermore, modularization is a prerequisite for incremental code generation which in turn can lead to a significant improvement of productivity. Therefore, the language should provide a means to decompose systems into small pieces that can be separately defined by the language users, e.g., by providing language elements which can be used in order to reference artifacts in other files.

Guideline 26: *"Introduce interfaces."* Interfaces in programming languages provide means for a modular development of parts of the system. This is especially important for complex systems as developers may define interfaces between their parts to be able to exchange one implementation of an interface with another which significantly increases flexibility. Furthermore, the introduction of interfaces is a common technique for information hiding: developers are able to change parts of their models and can be sure that these changes do not affect other parts of the system when the interface does not change. Therefore, we recommend that a DSL should provide an interface concept similar to the interfaces of known programming languages.

One example of interfaces are visibility modifiers in Java. They provide a means to restrict the access to members in a simple way. Another common example are ports, e.g., in composite structure diagrams, which explicitly define interaction points and specify services they provide or need, thus declaring a more detailed interface of a part of a system.

3. DISCUSSION

In the previous sections we introduced and categorized a bundle of guidelines dedicated to different language artifacts and development phases. Some of them already contained notes on relationships with other guidelines and trade-offs between them, and some of them briefly discussed their importance in different project settings. However, the following more detailed discussion shall help to identify possible conflicting guidelines and their reasons and gives hints on decision criteria.

The most contradicting point is reuse of existing artifacts versus the implementation of a language from scratch (cf. No. 5, 6, and 7). The main reason for the reuse of a language or a type system is that it can significantly decrease development time. Furthermore, existing languages often provide at least an initial level of quality. Thus, some of the guidelines, e.g., guidelines which target at consistency (e.g., No. 21) or claim modularity (e.g., No. 25), are met automatically. However, reusing existing languages can hinder flexibility and agility as an adaption may be hard to realize if not impossible. The same ideas apply to an improvement of the reused language itself (e.g., to meet guidelines which were not respected by the original language): the implementation of a single guideline may require a significant change of the language. Another important point is that this approach may influence the satisfiability of other guidelines. One example is No. 14 which suggests the reuse of existing notations of the domain. In case there are no languages which are similar to these notations, this guideline and language reuse are obviously contradicting. Furthermore, combining several existing languages may introduce conceptual inconsistencies, such as different styles or different underlying type systems which have to be translated into each other (cf., No. 5).

Implementing a new language from scratch in turn permits a high degree of freedom, agility, and flexibility. In this case, some guidelines can be realized more easily than in the case of reuse. However, these advantages are not for free: designing concrete and abstract syntax, context conditions, and a type system are time- and cost-intensive task. To summarize, a decision whether to reuse existing languages or to implement a new one is one of the most important and critical decisions to be made.

Another important point which was already mentioned in the introduction is that some of the presented guidelines have to be weighted according to the project settings, to the form of use, etc. One example is the expected size of the languages instances. Some DSLs serve as configuration languages and thus, typical instances consist of a small amount of lines only. Other DSLs are used to describe complex systems leading to huge instances. In the former case guidelines which target at compositionality or claim references between files (e.g., No. 19 and 25) have nearly no validity whereas in the latter example these guidelines are of high importance. However, not only the expected size of the instances can influence the weight of guidelines. Another important aspect is the intended usage of the language. Sometimes DSLs are not executable; they are designed for documentation only. In these cases, the guideline which demands to avoid inefficient elements in the language (No. 13) is of course not meaningful. However, for languages which are translated into running code, this is of high importance.

A last point we want to discuss here are the costs induced

by applying the guidelines. Some of them can be implemented easily and straightforward (e.g., distinguishability of elements or permitting comments, No. 16 and 18) whilst others require a significant amount of work (e.g., introduction of references between files including appropriate resolution mechanisms and symbol tables, No. 19). Of course, especially guidelines whose implementation is cost intensive have to be matched against project settings as described above. For small DSLs such guidelines should be ignored instead as the cost will often not amortize the improvements. However, from our experiences DSLs are often subject to changes. While growing these guidelines become more and more important. The main problem which emerges in these cases is that adding new things to a grown language (e.g., modularity) is typically more difficult and time-consuming than it would have been at the beginning. Therefore, analyzing the domain and usage scenarios as described in Guidelines 1 and 2 can prevent those unnecessary costs.

4. CONCLUSION

In this paper 26 guidelines have been discussed that should be considered while developing domain specific languages. To our experience this set of guidelines is a good basis for developing a language. For space reasons, we restricted ourselves to guidelines for designing the language itself. Other guidelines are needed for successfully integrating DSLs in a software development process, deploying it to new users, and evolving the syntax and existing models in a coherent way.

In general, a guideline should not be followed closely, but many of them are proposals as to what a language designer should consider during development. Some of the guidelines have to be discussed in certain domains, because they might not have the same relevance and as discussed many guidelines contradict each other and the language developer has to balance them appropriately.

But generally, the consideration of explicitly formulated guidelines is improving language design. We also think that it is worthwhile to develop much more detailed sets of concrete instructions for particular DSLs. We currently focus on textual languages in the spirit of Java.

Although we have compiled this list from literature and our own experience, we are sure that this list is not complete and has to be extended constantly. In addition, guidelines might change during time as developers gather more experience, tools become more elaborate, and taste changes. Maybe some guidelines are not relevant anymore in a few years, as some guidelines from the 1970's are less important today.

Acknowledgment: The work presented in this paper is partly undertaken in the MODELPLEX project. MODELPLEX is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006). Information included in this document reflects only the authors' views. The European Community is not liable for any use that may be made of the information contained herein.

5. REFERENCES

[1] Antlr Website `www.antlr.org`.

[2] GME Website `http://www.isis.vanderbilt.edu/projects/gme/`.

[3] T. Goldschmidt, S. Becker, and A. Uhl. Classification of concrete textual syntax mapping approaches. In *ECMDA-FA*, pages 169–184, 2008.

[4] J. Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall, November 2001.

[5] H. Grönniger, J. Hartmann, H. Krahn, S. Kriebel, and B. Rumpe. View-based modeling of function nets. In *Proceedings of the Object-oriented Modelling of Embedded Real-Time Systems (OMER4) Workshop, Paderborn,*, October 2007.

[6] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering*, 2007.

[7] G. Hamon and J. Rushby. An operational semantics for stateflow. In *Fundamental Approaches to Software Engineering: 7th International Conference (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243, Barcelona, Spain, March 2004. Springer-Verlag.

[8] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford University, Stanford, CA, USA, 1973.

[9] Java Code Conventions `http://java.sun.com/docs/codeconv/`.

[10] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.

[11] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of Tools Europe*, 2008.

[12] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. Technical Report SEN-E0309, Centrum voor Wiskunde en Informatica, Amsterdam, 2005.

[13] MontiCore Website `http://www.monticore.de`.

[14] R. Paige, J. Ostroff, and P. Brooke. Principles for Modeling Language Design. Technical Report CS-1999-08, York University, December 1999.

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[16] P. Pfahler and U. Kastens. Language Design and Implementation by Selection. In *Proc. 1st ACM-SIGPLAN Workshop on Domain-Specific-Languages, DSL '97*, pages 97–108, Paris, France, January 1997. Technical Report, University of Illinois at Urbana-Champaign.

[17] B. Rumpe. *Modellierung mit UML*. Springer, Berlin, May 2004.

[18] D. Spinellis. Notable Design Patterns for Domain Specific Languages. *Journal of Systems and Software*, 56(1):91–99, Feb. 2001.

[19] The Atlantic Zoo Website `http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/`.

[20] D. Wile. Lessons learned from real DSL experiments. *Science of Computer Programming*, 51(3):265–290, June 2004.

[21] D. S. Wile. Supporting the DSL Spectrum. *Computing and Information Technology*, 4:263–287, 2001.

[22] N. Wirth. On the Design of Programming Languages. In *IFIP Congress*, pages 386–393, 1974.

IEEE Software
www.computer.org/software

Worst Practices for Domain-Specific Modeling

Steven Kelly and Risto Pohjonen

Vol. 26, No. 4
July/August 2009

Worst Practices for Domain-Specific Modeling

Steven Kelly and Risto Pohjonen, *MetaCase*

Little guidance exists on creating domain-specific modeling languages. Learning what not to do—including how to deal with common pitfalls and recognizing troublesome areas—can help.

In computing's early days, language creation was a common activity, but by the millennium's end it was relegated to a few gurus. Early articles[1] citing "1,700 special programming languages" or hundreds of modeling languages were smothered by the ubiquity of languages such as Java and UML. The current decade has seen a resurgence of interest in domain-specific languages, particularly domain-specific modeling (DSM) languages. The reasons for this renewed growth include the availability of tools to create and work with such languages, and the frequency of cases in which productivity increased by a factor of 5–10 with the introduction of DSM.[2]

All too often, however, language developers have had to fly by the seats of their pants because little material is available to teach them how to create a good language. Although industrial books offer solid background on why we need such languages,[3,4] and academic research offers theories and analysis of them,[5–7] both fields mostly omit instruction on how to actually build them.

There are a few good guides to creating a DSM language, including articles[8,9] and a recent book.[2] Still, many readers are left feeling uncertain, and many languages repeat basic mistakes. Perhaps in language creation, as in music, it's easier to teach what not to do and thus help even first-timers create something acceptable. At the least, knowing what to avoid can be a valuable addition to a set of best practices, enabling language developers to recognize troublesome situations early and thus save themselves from later rework. Here, we outline the common pitfalls, focusing on language creation and use; length restrictions prevent us covering generators or wider organizational issues.

Method Overview

We've identified several worst practices during our experience over the years. To refine our categories, we analyzed 76 DSM cases. This sample is relatively broad, spanning 15 years, four continents, several tools, around 100 language creators, and projects having from three to more than 300 modelers. Among the problem domains are automotive, avionics, mobile, medical, consumer electronics, enterprise systems, system integration, and server configuration. Solution domains include assembler, Basic, C, C++, C#, Java, JavaScript, shell scripts, Python, Prolog, Matlab, SQL, and various XML schemas. That said, the sample does contain a preponderance of cases in Europe or involving MetaEdit+, which is somewhat excused by the

0740-7459/09/$25.00 © 2009 IEEE

fact that these conditions probably accounted for the majority of DSM cases worldwide.

We present the worst practices here in the order you'd encounter them over the life of a project: the initial starting conditions; the domain concept sources; the resulting language; the language's notation; and the language's use. We also list the percentage of cases in which we observed the practice. Because a single case might exhibit zero or many worst practices, percentages might not sum to 100 percent. Finally, we changed some details in example diagrams to protect the identities and rights of those involved.

Initial Conditions

Even before language creation begins, wrong attitudes and decisions can have a serious effect on later success.

Only Gurus Allowed

Believing that only gurus can build languages (4 percent) or that "I'm smart and don't need help" (12 percent)

Decades of experience with theoretical fundamentals, software systems, and language creation might be helpful when developing general-purpose languages. However, such a background isn't the key success factor when developing DSM languages. Because DSM languages try to solve fewer problems than general-purpose languages, they're typically simpler to create. They're not, however, simplistic; they require in-depth understanding and experience with the problem domain. So, appropriate domain expertise is more important than knowledge of language theory.

The other extreme to avoid is trying to do everything yourself, ignoring other people's expertise on how to make good languages. Although it's good for organizations to view their own resources as the key element for developing their DSM language, excessive complacence and a "not invented here" attitude can prove counterproductive. The cruel truth is that, without help, everyone's first language—like everyone's first program—is unlikely to be a masterpiece.

Lack of Domain Understanding

Insufficiently understanding the problem domain (17 percent) or the solution domain (5 percent)

Creating a DSM language requires a good understanding of the problem domain. Normally, this shouldn't be a problem, but occasionally companies make the mistake of delegating the task to a summer intern, or seasoned developers take it on and fail to lift their noses above the level of the code. The language must also set a reasonable boundary around the kinds of applications to be built, sparing at least a thought for future expansion.

Other possible problems when assembling domain concepts into a language include a lack of conceptual or abstract thinking skills or a lack of experience in building nontrivial systems. Such skills can come from fields other than programming. However, programming is perhaps the best teacher because it offers a good vocabulary for principles such as *DRY* (don't repeat yourself; that is, avoid duplicating code or data) and modularization (aim for high cohesion and low coupling between system parts). These principles are at least as necessary when building a language as they are when building an application.

Although creating a DSM language should focus on the problem domain, inexperience in the solution domain can cause problems later. The best DSM language creator is an experienced developer who focuses only on the problem domain, but lets his solution domain experience inform his choices among otherwise equally viable solutions.

Analysis Paralysis

Wanting the language to be theoretically complete, with its implementation assured (8 percent)

The motivation for this kind of mistake is rather obvious: fear. For most of us humans, it's rational to be cautious when entering unfamiliar territory, such as creating a language for the first time. Another form of this problem is a desire to solve every possible problem: that is, a tool isn't useful unless you can use it for everything.

DSM isn't about achieving perfection, just something that works in practice. It will always be possible to imagine a case that the language can't handle. The important questions are how often such cases occur in practice, and how well the language deals with common cases. To avoid analysis paralysis, concentrate on the core cases and build a prototype language for them.

The Source for Language Concepts

The first step in building a DSM language is identifying its concepts. The problem domain is the ideal source; relying too much on secondary sources is a recipe for trouble.

UML: New Wine in Old Wineskins

Extending a large, general-purpose modeling language (5 percent)

Although it's obviously tempting to build on an established language's constructs and semantics, such languages are typically too generic and broad

Because DSM languages try to solve fewer problems than general-purpose languages, they're typically simpler to create.

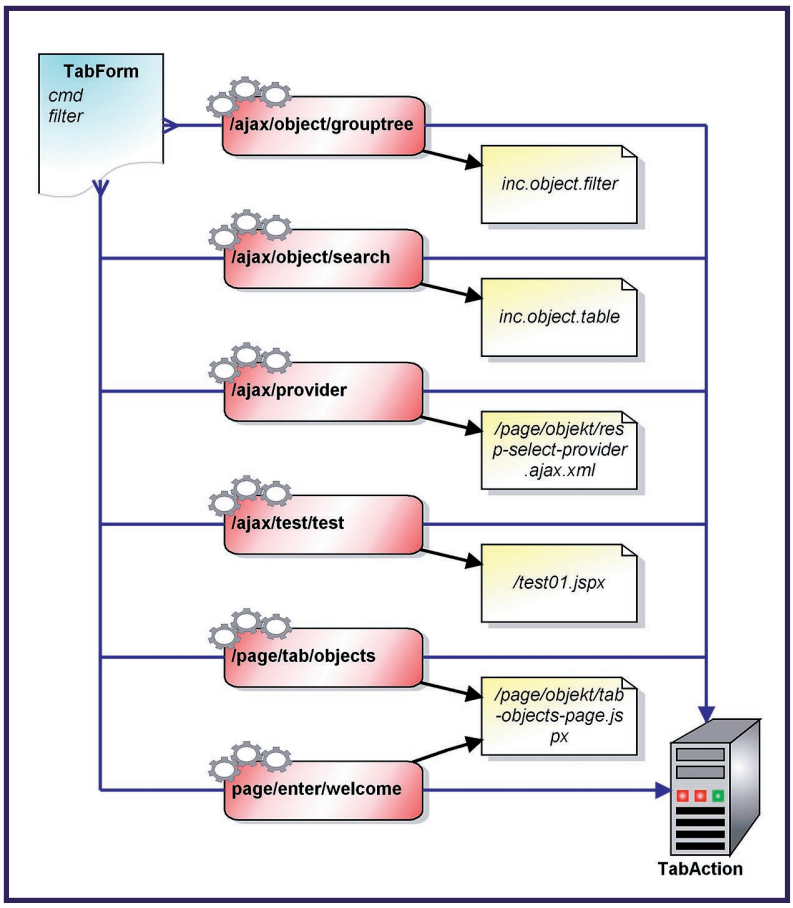


Figure 1. Focusing on framework code. Overemphasizing the target framework or component library can result in low-level details and unnecessary duplication.

for any specific domain. Stripping off parts of the original language and adding new concepts and semantics is often more work than simply starting from scratch. That said, it's obviously good to reuse the basic ideas and concepts of established languages, such as states, data flow, control flow, and inheritance.

In theory, the opposite is also possible: an existing language might be too small or narrow. In practice, however, this seems uncommon and is easier to correct by extending the existing concepts.

3GL: Visual Programming

Duplicating the concepts and semantics of traditional programming languages (7 percent)

Although incorporating programming language concepts such as choices or loops in DSM languages can be useful, you shouldn't let them become the core concepts at the expense of those in the problem domain. The peril in this case is to end up with generic visual programming instead of DSM, leading to a language with a poor level of abstraction. Visual programming languages of this type often have lower expressive power and are more difficult to use than the manual code they're designed to replace.

Code: The Library Is the Language

Focusing the language on the current code's technical details (32 percent)

Although you should derive the modeling language concepts primarily from the problem domain, some solution domain influence is acceptable. However, if the language overemphasizes the target framework or component library, it can drag the abstraction level down toward the code level, preventing retargeting to other platforms. This directly opposes DSM's idea of achieving the best possible level of abstraction for software development. Solution-domain-based languages often expose the implementation details and repetition common in code. Figure 1 shows an example of both: each object pair in the middle could be replaced by a single object, with the implementation details abstracted out.

This was the most common worst practice in our sample, which is hardly surprising when you consider the domain framework's role. At the beginning of a language development project, a framework often represents the solution domain's best existing abstraction; it's also well understood by the domain experts and familiar to the programmers. Given this, a framework is a plausible candidate for the language concepts, but it's typically best to return instead to the source: the problem domain itself.

Tool: If You Have a Hammer ...

Letting the tool's technical limitations dictate language development (14 percent)

Ensuring good tool support for a language is an important aspect of its development, but focusing on tool issues or getting trapped into seeing the world through the tool's limitations is a mistake. Different DSM tools have different emphases, and not all tools support all parts of DSM equally well. Using a poorly suited or weak tool can lead you to make decisions on the basis of what the tool supports, rather than what's needed for the problem domain or the modelers. Figure 2a shows an example where a tool led even an experienced developer to create a language for menu structures that's hard to read and use; Figure 2b would be clearer. Also, practices you learn as workarounds for weaknesses in one tool can all too easily be carried over when you work with another tool that's stronger in that area.

Similarly, people often get carried away with a

tool's new or cool features at the expense of getting the language's substance right. A sound foundation has more effect on a language's usefulness and success than do the latest bells and whistles. Also, don't feel obliged to use all tool features: just because a tool supports something doesn't necessarily mean it's a good idea.

The Resulting Language

Building a language is a balancing act between a number of forces, both technical and psychological.

Too Generic/Too Specific

Creating a language with a few generic concepts (21 percent) or too many specific concepts (8 percent), or a language that can create only a few models (7 percent)

Finding the proper generic-specific balance is a key success factor in DSM development—and is thus a rather common place to make mistakes. Developers often create a language that's too generic for its domain, with concepts and semantics that are either too few, too generic, or both. In Figure 3, for example, adding the concepts of "lights" and "heating" would improve the language. A good benchmark here is to see whether you can use your language to model in domains other than your target problem domain. If so, your language is probably too generic.

The other extreme is a language with too many concepts, which are probably too narrow semantically or overlap. This creates problems during language deployment and use; overly complex languages are difficult to learn, master, and maintain.

An interesting variant on the theme of genericity is a language that enables users to create only a few potential models. DSM solutions are mass-production environments first and foremost; if users can't create many applications, building the language might be a waste of effort.

Misplaced Emphasis

Too strongly emphasizing a particular domain feature (12 percent)

By definition, DSM languages should have a strong emphasis on the domain concepts. Unfortunately, language developers can stretch this good practice too far by focusing on a particular feature or concept at the expense of others. This is especially troublesome if that concept has little or no value for the DSM solution. Typically, such a situation arises when you let too many stakeholders influence the language development. It's good to listen to different voices to understand the domain and the prospective language usage, but you should always retain a clear vision of the language's "big picture" and objectives.

Similarly, some developers might be tempted to put every domain element into the language,

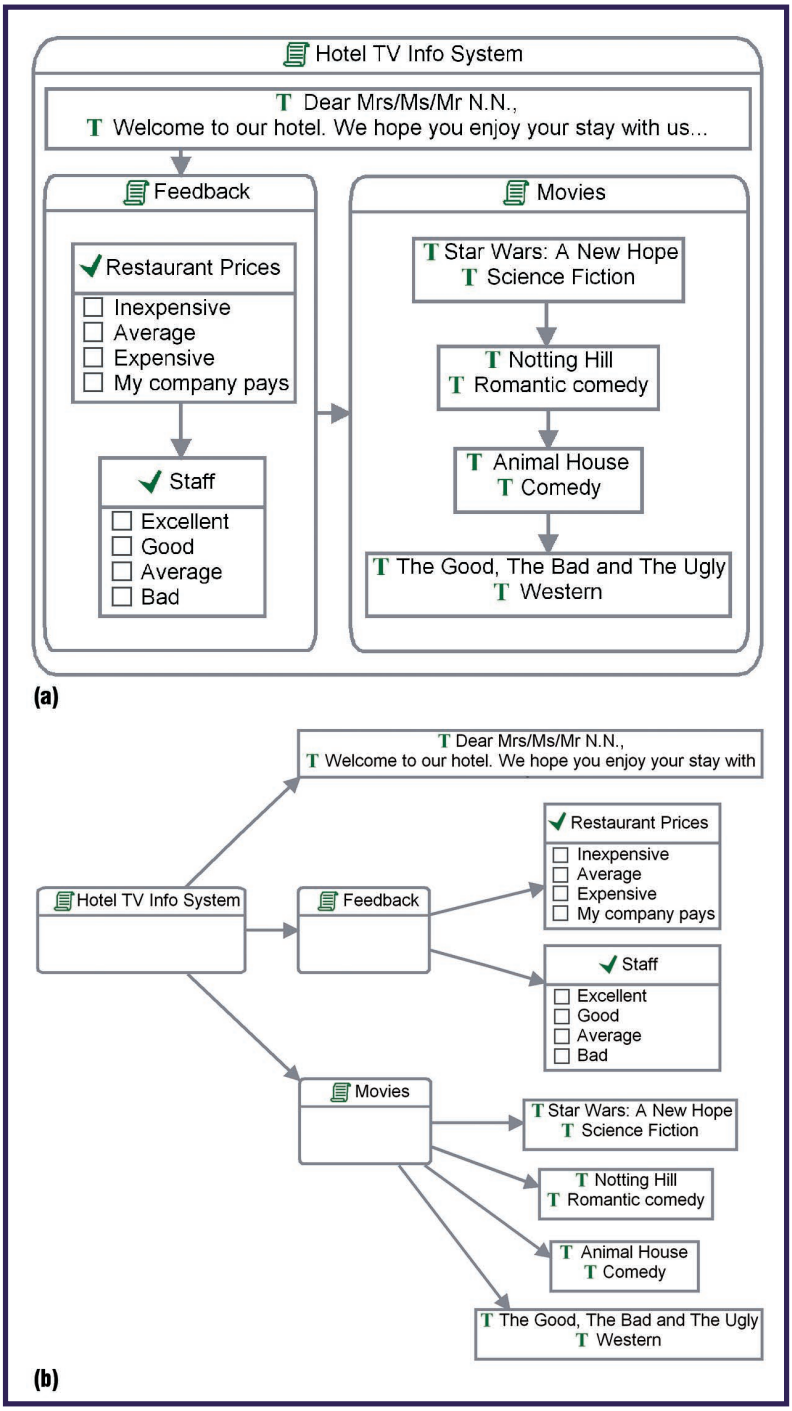


Figure 2. Tool choice and outcomes. (a) A tool focused on strong containment leads to an odd, labor-intensive model structure. (b) Replacing the visual containment with relationships makes the menu structure clearer.

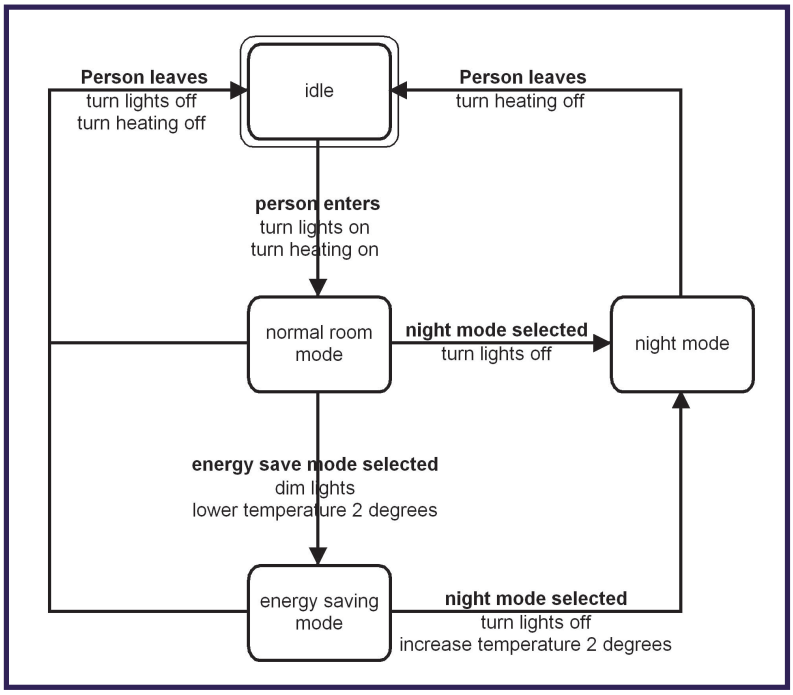


Figure 3. Insufficient concepts. This language has too few concepts, and they're too generic for this domain. Adding explicit concepts for "lights" and "heating" would improve the language considerably.

forgetting the importance of deciding what not to incorporate. Many DSM cases are essentially software product lines, and their languages should model variability—you can omit any commonalities among all products, handling them instead in the generators or domain framework.[4]

Sacred at Birth

Viewing the initial language version as unalterable (12 percent)

This rather common mistake occurs for several reasons. Most of us don't like the idea of "build one to throw away" and are thus reluctant to discard or radically modify our first draft. People often view language creation as a waterfall process, neglecting its iterative nature and the need for prototyping. This mistake can also result from spacing development milestones too far apart. In this case, language creators often invest too much effort into a development step without testing the language in real life, which makes it difficult to step back if needed. Tool support plays an important role here: inflexible tools often lead to extra work in rebuilding models when the modeling language changes.

Language evolution is inevitable, and modifying a language is easier when only a few people know it and only a few models exist. The language is also less proven at this stage, so there will be more flaws and more room for improvement.

Language Notation

A poorly chosen concrete syntax will drive users away, stopping them from using even the most wonderful language.

Predetermined Paradigm

Choosing the wrong representational paradigm on the basis of a blinkered view (7 percent)

Many people approach DSM with a fixed idea of how to represent systems, such as through text or graphical diagrams. Although 75 percent of the general population reportedly prefer visual rather than textual representations,[10] a higher proportion of developers might be predisposed to choose text given its traditional prevalence in programming. 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.

This error is almost certainly underreported in our sample because, of the available tools, MetaEdit+ supports the widest variety of representational paradigms. Also, developers who prefer text might have self-selected themselves out of the sample by using a simpler, purely textual editor.

Simplistic Symbols

Using symbols that are too simple or similar (25 percent) or downright ugly (5 percent)

One of the most common failure areas is in the language's notation—its symbols or icons. Unlike more abstract or general-purpose languages, DSM languages can often find familiar, intuitive representations directly from the problem domain. All too often, however, the symbols for different language concepts are just boxes with the concepts' names as labels. People recognize things by their shapes, not by labels (if you doubt this, stick the label "lemon" on a banana and see how people react). Also, symbols differing in color alone are suboptimal: the brain views color change primarily as a different version of the same thing, not as a completely different thing. Figure 4 shows an example of both mistakes.

Alan Blackwell has shown that the best symbols are pictograms, not simpler geometric shapes or more complex bitmap or photographic representations.[11] Although our sample contained no cases with overly complex bitmap symbols, you should avoid these as well—bitmaps scale poorly (particularly with aspect-ratio changes) and have little room for text or other contents.

Symbols have an aesthetic role, and few people are fortunate enough to have both the abstract thinking that language design requires and the artistic skills needed to create great symbols. Not surprisingly, the few truly ugly languages in our sample encountered significant opposition from users. Take such opposition seriously: find someone with decent graphic design skills to improve your symbols.

Language Use

All too often, language creators forget that languages are made to be used and to serve their users. Percentages here are only of those languages that have already seen significant use by people other than their creators.

Ignoring the Use Process

Failing to consider the language's real-life usage (42 percent)

It's notoriously hard to predict how people will use a new system or how group members' individual efforts will interact when brought together. Language developers ignore this topic at their peril: To have any value, the language and its use process must serve the modelers. This category involves five areas of concern.

First, generally, multiple people will use a DSM language to make multiple models. To avoid having modelers reenter or copy-and-paste the same information multiple times, plan for reuse and referencing among models in advance. Models that interconnect should do so with minimal coupling. Data duplication and a lack of modularization invariably lead to maintenance nightmares. Figure 5 shows a particularly unpleasant example: The user copied the whole model to achieve a variant without the small time-out object on the left. Instead, the language could have offered concepts for reusing models or made the generator or framework ignore time-out objects on platforms that don't support them.

Second, semiautomated model transformations help users create more data quickly, but with poor long-term results. Unlike full transformations, users must maintain the extra data by editing generated source code or model transformation results as in MDA (model-driven architecture). Anything that transformations can create automatically can be created at generation time, avoiding the maintenance burden and letting transformations change freely over time.

Third, language creators often try to prevent modeler error by creating myriad strongly enforced rules that serve only to annoy, preventing modelers from breaking the rules even temporarily while they're changing their models.

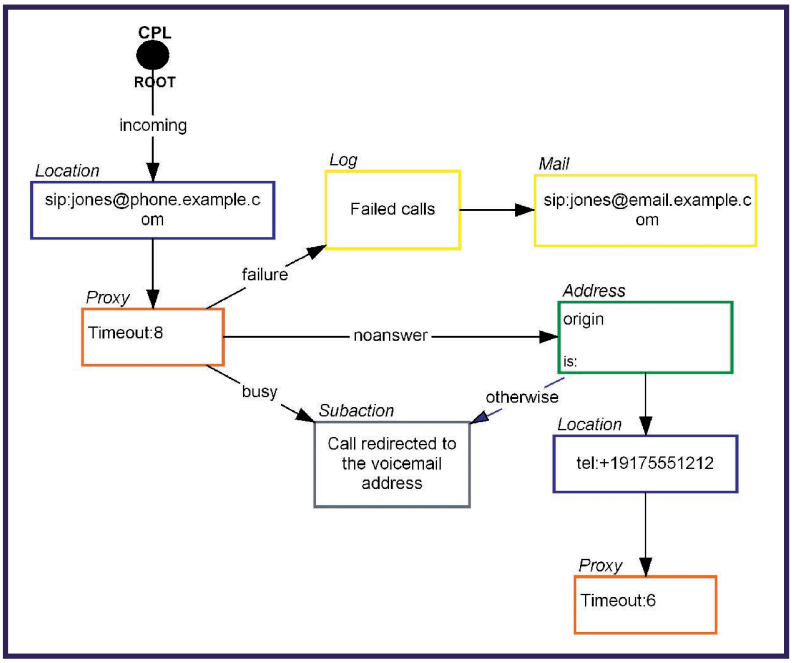


Figure 4. Inadequate symbol differentiation. Symbols differing in only color and label are insufficient. Research shows that the best symbols are pictograms rather than simple geometric shapes or photorealistic bitmaps.

Fourth, unsurprisingly, developers using DSM often uncritically apply processes that have evolved to support source-code-based development. Many such practices are simply crutches and bandages evolved to fix problems inherent in source code and its single-user editing. Repository-based multiuser editing works much better for models, as does a proper modularization and division of labor.

Finally, debugging DSM models at the source-code level is a bad idea if the structure of models and source code differ significantly. When the model-to-code mapping is unclear, it's hard to know where to insert a breakpoint in generated code. When the code-to-model mapping is unclear, it's hard to correct a bug found during debugging. It's better to have running code call back to the modeling tool to highlight the current symbol, and let the modelers set breakpoints there.

No Training

Assuming everyone understands the language like its creator (21 percent)

Although the use of familiar domain concepts makes DSM languages easier to learn, it doesn't mean users will immediately understand them completely. Language creators often overlook this fact and become disconnected from the modelers. The

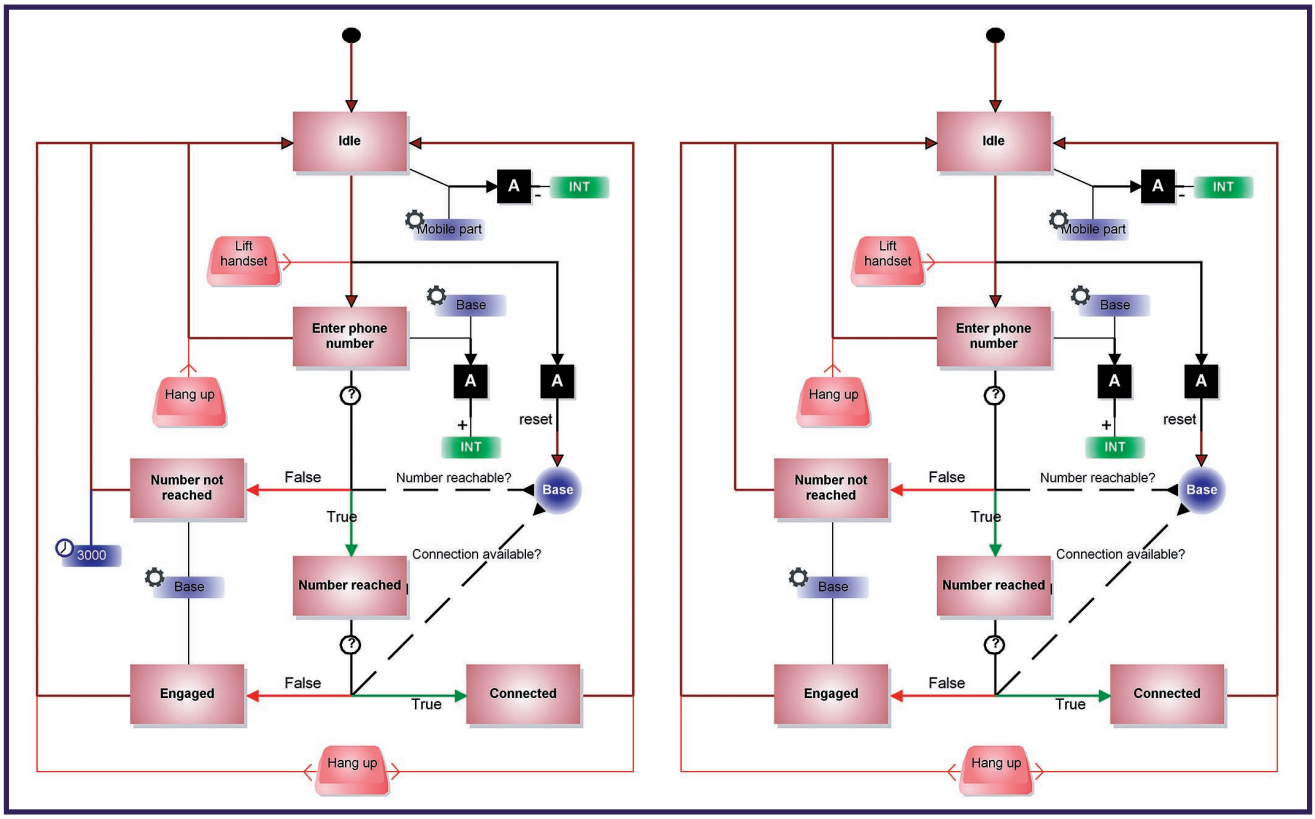


Figure 5. Poor planning for reuse of models. The modeler in this case had to copy the entire diagram to account for a minor variation: the small time-out object on the left.

task of language creation doesn't stop when everything works: you must create documentation and training materials and communicate them to users. DSM research indicates that failures here lead to problems and long-term resistance, even when support later improves.[12] As with any project, it's worthwhile to involve users early, both to get practical feedback and to achieve smooth acceptance.

Post-adoption Stagnation

Letting the language stagnate after successful adoption (37 percent)

Successful adoption of a DSM language implies many models and modelers. The greater the number of models and modelers, the harder changing the language is. Although the best tools can automatically update models when the language changes, you can't automatically update the modelers' brains.

Fortunately, our experience indicates that the problem domain changes that affect a deployed language tend to be additive—that is, they involve new concepts or concept extensions that both modelers and tools adopt with relative ease. To avoid language stagnation, you should make such changes promptly rather than postpone them. You should also avoid passing off language maintenance to someone unsuited to the task.

After several years, a problem domain might change sufficiently to create problems. (However, this situation is rare.) Trying to shoehorn such changes into the old language might not work, and a massive update of the modeling language and all models might be impractical. Another option is to create a new language for the new domain: the better fit can create increased productivity that often balances out the cost, just as it did when creating the first language.

Preliminary Analysis

Our sample covered 76 cases, mostly of companies using MetaCase as consultants or tool providers; in some cases MetaCase was not involved but we have been able to discuss the case with participants. In all, 7 percent of the cases were carried out by MetaCase alone, 57 percent by the customer with consultancy from MetaCase, and 36 percent with no consultancy from MetaCase. In 15 percent of cases, participants used a tool other than MetaEdit+ (at least initially).

In assessing cases by worst practices, we agreed on landmark cases to determine the watershed—

for example, to be counted as "ugly," symbols had to be at least as ugly as case X. We normalized worst practices to questions with either a simple yes-or-no answer or a three-point scale, such as *too generic*, *acceptable*, and *too specific*.

As a preliminary analysis, we calculated the correlation among practices, given below as Pearson's coefficient, r, expressed as a percentage. All correlations below are statistically significant (n = 76, α = 0.05, one-tailed, $|r| \geq .190$), but the relationship's direction and its possible causality are our own interpretation.

The single largest factor that led to a language not being used was when organizations gave the language design task to someone with insufficient experience in the problem domain (26 percent).

Basing the language on code led developers to try to take everything into consideration (33 percent). This desire for theoretical completeness was often accompanied by ascetic symbols (28 percent). Using code as a basis also led to stagnation (37 percent).

If the language developer didn't accept help initially, the language was likely to become sacred (24 percent). Sacred languages were likely to stagnate (31 percent). However, sacred languages were also more likely to be used in practice (35 percent)—perhaps because their developers loved them and pushed for their use.

Using a poor tool required extra effort, so developers were less willing to change their languages and those languages thus became sacred (31 percent). Poor tools also led to languages whose abstraction level was no higher than programming languages (34 percent), while poor facilities for defining symbols led to ugly notation (41 percent). A lack of attention to symbols correlated with insufficient training (47 percent), showing a consistent disregard for users.

Examining our sample cases in relation to the initial set of worst practices helped us tighten up the boundaries between practices and identify some extra facets. We were surprised by the rarity of certain practices—including using existing languages as sources for concepts and making the initial language draft sacred. However, we often try to warn customers about such issues early, and they probably avoided them as a result. It would be interesting and instructive to repeat the analysis for cases with other tools or extend the preliminary analysis with more detail on the division of labor and the language developers' relative experience. The most important result, however, would be if our honesty about these failings in our own cases could help others avoid falling into the same traps.

About the Authors

Steven Kelly is chief technology officer of MetaCase and has more than 15 years' experience building domain-specific modeling tools and languages. He has a PhD in information systems from Jyväskylä University. Contact him at stevek@metacase.com.

Risto Pohjonen is a domain-specific modeling (DSM) consultant and developer at MetaCase, with over 10 years' experience building DSM tools and languages. Contact him at rise@metacase.com.

Acknowledgments

We thank all the MetaCase staff, particularly Juha-Pekka Tolvanen and Janne Luoma, and all the people we've worked with on these cases.

References

1. *Computer Software Issues, An American Mathematical Association Prospectus*, July 1965, quoted in P.J. Landin, "The Next 700 Programming Languages," *Comm. ACM*, vol. 9, no. 3, 1966, pp. 157–166.
2. S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation,* John Wiley & Sons, 2008.
3. J. Greenfield and K. Short, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools,* John Wiley & Sons, 2004.
4. D. Weiss and C.T.R. Lai, *Software Product-Line Engineering*, Addison Wesley Longman, 1999.
5. J. Ralyté, S. Brinkkemper, and B. Henderson-Sellers, eds., *Situational Method Engineering: Fundamentals and Experiences*, Springer, 2007.
6. D. Spinellis, "Notable Design Patterns for Domain Specific Languages," *J. Systems and Software,* vol. 56, no. 1, 2001, pp. 91–99.
7. G. Costagliola et al., "A Classification Framework to Support the Design of Visual Languages," *J. Visual Languages and Computing*, vol. 13, no. 6, 2002, pp. 573–600.
8. D. Roberts and R. Johnson, "Evolve Frameworks into Domain-Specific Languages," *Proc. 3rd Int'l Conf. Pattern Languages*, 1996; www.cs.wustl.edu/~schmidt/PLoP-96/roberts.ps.gz.
9. M. Voelter and J. Bettin, "Patterns for Model-Driven Software Development"; www.voelter.de/data/pub/MDDPatterns.pdf.
10. *Train the Trainer*, Int'l Assoc. Information Technology Trainers, 2001; http://itrain.org/pdf/itrain_ttt_course_outlines.pdf.
11. A. Blackwell, *Metaphor in Diagrams*, PhD thesis, Darwin College, Univ. of Cambridge, 1998.
12. J. Ruuska, "Factors Influencing CASE Tool User Satisfaction: An Empirical Study in a Large Telecommunications Company," master's thesis, Dept. of Computer Sciences, Univ. of Tampere, 2001 (in Finnish).

When and How to Develop Domain-Specific Languages

MARJAN MERNIK

University of Maribor

JAN HEERING

CWI

AND

ANTHONY M. SLOANE

Macquarie University

Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application. DSL development is hard, requiring both domain knowledge and language development expertise. Few people have both. Not surprisingly, the decision to develop a DSL is often postponed indefinitely, if considered at all, and most DSLs never get beyond the application library stage.

Although many articles have been written on the development of particular DSLs, there is very limited literature on DSL development methodologies and many questions remain regarding when and how to develop a DSL. To aid the DSL developer, we identify patterns in the decision, analysis, design, and implementation phases of DSL development. Our patterns improve and extend earlier work on DSL design patterns. We also discuss domain analysis tools and language development systems that may help to speed up DSL development. Finally, we present a number of open problems.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Specialized Application Languages*

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Domain-specific language, application language, domain analysis, language development system

---

Authors' addresses: M. Mernik, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia; email: marjan.mernik@uni-mb.si; J. Heering, Department of Software Engineering, CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands; email: Jan.Heering@cwi.nl; A.M. Sloane, Department of Computing, Macquarie University, Sydney, NSW 2109, Australia; email: asloane@ics.mq.edu.au.

1. INTRODUCTION

1.1. General

Many computer languages are *domain-specific* rather than general purpose. Domain-specific languages (DSLs) are also called application-oriented [Sammet 1969], special purpose [Wexelblat 1981, p. xix], specialized [Bergin and Gibson 1996, p. 17], task-specific [Nardi 1993], or application [Martin 1985] languages. So-called *fourth-generation languages* (4GLs) [Martin 1985] are usually DSLs for database applications. *Little languages* are small DSLs that do not include many features found in general-purpose programming languages (GPLs) [Bentley 1986, p. 715].

DSLs trade generality for expressiveness in a limited domain. By providing notations and constructs tailored toward a particular application domain, they offer substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs. Also, by reducing the amount of domain and programming expertise needed, DSLs open up their application domain to a larger group of software developers compared to GPLs. Some widely used DSLs with their application domains are listed in Table I. The third column gives the language level of each DSL as given in Jones [1996]. Language level is related to productivity as shown in Table II, also from Jones [1996]. Apart from these examples, the benefits of DSLs have often been observed in practice and are supported by quantitative results such as those reported in Herndon and Berzins [1988]; Batory et al. [1994]; Jones [1996]; Kieburtz et al. [1996]; and Gray and Karsai [2003], but their quantitative validation in general as well as in particular cases, is hard and an important open problem. Therefore, the treatment of DSL development in this article will be largely qualitative.

The use of DSLs is by no means new. APT, a DSL for programming numerically-controlled machine tools, was developed in 1957–1958 [Ross 1981]. BNF, the well-known syntax specification formalism, dates back to 1959 [Backus 1960]. Domain-specific visual languages (DSVLs), such as visual languages for hardware description and protocol specification, are important but beyond the scope of this survey.

We will not give a definition of what constitutes an application domain and what does not. Some consider Cobol to be a DSL for business applications, but others would argue this is pushing the notion of application domain too far. Leaving matters of definition aside, it is natural to think of DSLs in terms of a gradual scale with very specialized DSLs such as BNF on the left and GPLs such as C++ on the right. (The language level measure of Jones [1996] is one attempt to quantify this scale.) On this scale, Cobol would be somewhere between BNF and C++ but much closer to the latter. Similarly, it is hard to tell if command languages like the Unix shell or scripting languages like Tcl are DSLs. Clearly, domain-specificity is a matter of degree.

In combination with an *application library*, any GPL can act as a DSL. The library's Application Programmers Interface (API) constitutes a domain-specific vocabulary of class, method, and function names that becomes available by object creation and method invocation to any GPL program using the library. This being the case, why were DSLs developed in the first place? Simply because they can offer domain-specificity in better ways.

—Appropriate or established *domain-specific notations* are usually beyond the limited user-definable operator notation offered by GPLs. A DSL offers appropriate domain-specific notations from the start. Their importance should not be underestimated as they are directly related to the productivity improvement associated with the use of DSLs.

—Appropriate *domain-specific constructs and abstractions* cannot always be mapped in a straightforward way to functions or objects that can be put in a library. Traversals and error handling are typical examples [Bonachea et al. 1999; Gray and Karsai 2003; Bruntink

Table I. Some Widely Used Domain-Specific Languages

DSL	Application Domain	Level	
BNF	Syntax specification	n.a.	
Excel	Spreadsheets	57	(version 5)
HTML	Hypertext web pages	22	(version 3.0)
LaTeX	Typesetting	n.a.	
Make	Software building	15	
MATLAB	Technical computing	n.a.	
SQL	Database queries	25	
VHDL	Hardware design	17	
Java	General-purpose	6	(comparison only)

Table II. Language Level vs. Productivity as Measured in Function Points (FP)

Level	Productivity Average per Staff Month (FP)
1–3	5–10
4–8	10–20
9–15	16–23
16–23	15–30
24–55	30–50
> 55	40–100

et al. 2005]. A GPL in combination with an application library can only express these constructs indirectly or in an awkward way. Again, a DSL would incorporate domain-specific constructs from the start.

—Use of a DSL offers possibilities for *analysis*, *verification*, *optimization*, *parallelization*, and *transformation* in terms of DSL constructs that would be much harder or unfeasible if a GPL had been used because the GPL source code patterns involved are too complex or not well defined.

—Unlike GPLs, DSLs *need not be executable*. There is no agreement on this in the DSL literature. For instance, the importance of nonexecutable DSLs is emphasized in Wile [2001], but DSLs are required to be executable in van Deursen et al. [2000]. We discuss DSL executability in Section 1.2.

Despite their shortcomings, application libraries are formidable competitors to DSLs. It is probably fair to say that most DSLs never get beyond the application library stage. These are sometimes called *domain-specific embedded languages* (DSELs) [Hudak 1996]. Even with improved DSL development tools, application libraries will remain the most cost effective solution in many cases, the more so since the advent of component technologies such as COM and CORBA [Szyperski 2002] has further complicated the relative merits of DSLs and application libraries. For instance, Microsoft Excel's macro language is a DSL for spreadsheet applications which adds programmability to Excel's fundamental interactive mode. Using COM, Excel's implementation has been restructured into an application library of COM components, thereby opening it up to GPLs such as C++, Java, and Basic which can access it through its COM interfaces. This process of componentization is called *automation* [Chappell 1996]. Unlike the Excel macro language, which by its very nature is limited to Excel functionality, GPLs are not. They can be used to write applications transcending Excel's boundaries by using components from other automated programs and COM libraries in addition to components from Excel itself.

In the remainder of this section, we discuss DSL executability (Section 1.2), DSLs as enablers of reuse (Section 1.3), the scope of this article (Section 1.4), and DSL literature (Section 1.5).

1.2. Executability of DSLs

DSLs are executable in various ways and to various degrees even to the point of being nonexecutable. Accordingly, depending on the character of the DSL in question, the corresponding programs are often more properly called specifications, definitions, or descriptions. We identify some points on the DSL executability scale.

—DSL with well-defined execution semantics (e.g., Excel macro language, HTML).

—Input language of an *application generator* [Cleaveland 1988; Smaragdakis and Batory 2000]. Examples are ATMOL [van Engelen 2001], a DSL for atmospheric modeling, and Hancock [Bonachea et al. 1999], a DSL for customer profiling. Such languages are also executable, but they usually have a more declarative character and less well-defined execution semantics as far as the details of the generated applications are concerned. The application generator is a compiler for the DSL in question.

—DSL not primarily meant to be executable but nevertheless useful for application generation. The syntax specification formalism BNF is an example of a DSL with a purely declarative character that can also act as an input language for a parser generator.

—DSL not meant to be executable. Examples are domain-specific data structure representations [Wile 2001]. Just like their executable relatives, such nonexecutable DSLs may benefit from various kinds of tool support such as specialized editors, prettyprinters, consistency checkers, analyzers, and visualizers.

1.3. DSLs as Enablers of Reuse

The importance of DSLs can also be appreciated from the wider perspective of the construction of large software systems. In this context, the primary contribution of DSLs is to enable reuse of software artifacts [Biggerstaff 1998]. Among the types of artifacts that can be reused via DSLs are language grammars, source code, software designs, and domain abstractions. Later sections provide many examples of DSLs; here we mention a few from the perspective of reuse.

In his definitive survey of reuse Krueger [1992] categorizes reuse approaches along the following dimensions: abstracting, selecting, specializing, and integrating. In particular, he identifies application generators as an important reuse category. As already noted, application generators often use a DSL as their input language, thereby enabling programmers to reuse semantic notions embodied in the DSL without having to perform a detailed domain analysis themselves. Examples include BDL [Bertrand and Augeraud 1999] that generates software to control concurrent objects and Teapot [Chandra et al. 1999] that produces implementations of cache coherence protocols. Krueger identifies definition of domain coverage and concepts as a difficult challenge for implementors of application generators. We identify patterns for domain analysis in this article.

DSLs also play a role in other reuse categories identified by Krueger [1992]. For example, software architectures are commonly reused when DSLs are employed because the application generator or compiler follows a standard design when producing code from a DSL input. For example, GAL [Thibault et al. 1999] enables reuse of a standard architecture for video device drivers. DSLs implemented as application libraries clearly enable reuse of source code. Prominent examples are Haskell-based DSLs such as Fran [Elliott 1999]. DSLs can also be used for formal specification of software schemas. For example, Nowra [Sloane 2002] specifies software manufacturing processes and SSC [Buffenbarger and Gruell 2001] deals with subsystem composition.

Reuse may involve exploitation of an existing language grammar. For example, Hancock [Bonachea et al. 1999] piggybacks on C, while SWUL [Bravenboer and Visser 2004] extends Java. Moreover, the success of XML for DSLs is largely based on reuse of its grammar for specific domains. Less formal language grammars may also be reused when notations used by domain experts, but not yet available in a computer language, are realized in a DSL. For example, Hawk [Launchbury et al. 1999] uses a textual form of an existing visual notation.

1.4. Scope of This Article

There are no easy answers to the "when and how" question in the title of this article. The previously mentioned benefits of DSLs do not come free.

—DSL development is hard, requiring both domain and language development expertise. Few people have both.

—DSL development techniques are more varied than those for GPLs, requiring careful consideration of the factors involved.

—Depending on the size of the user community, development of training material, language support, standardization, and maintenance may become serious and time-consuming issues.

These are not the only factors complicating the decision to develop a new DSL. Initially, it is often far from evident that a DSL might be useful or that developing a new one might be worthwhile. This may become clear only after a sizable investment in domain-specific software development using a GPL has been made. The concepts underlying a suitable DSL may emerge only after a lot of GPL programming has been done. In such cases, DSL development may be a key step in software reengineering or software evolution [Bennett and Rajlich 2000].

To aid the DSL developer, we provide a systematic survey of the many factors involved by identifying patterns in the decision, analysis, design, and implementation phases of DSL development (Section 2). Our patterns improve and extend earlier work on DSL design patterns, in particular [Spinellis 2001]. This is discussed in Section 2.6. The DSL development process can be facilitated by using domain analysis tools and language development systems. These are surveyed in Section 3. Finally, conclusions and open problems are presented in Section 4.

1.5. Literature

We give some general pointers to the DSL literature; more specific references are given at appropriate points throughout this article rather than in this section. Until recently, DSLs received relatively little attention in the computer science research community, and there are few books on the subject. We mention Martin [1985], an exhaustive account of 4GLs; Biggerstaff and Perlis [1989], a two-volume collection of articles on software reuse including DSL development and program generation; Nardi [1993], focuses on the role of DSLs in end-user programming; Salus [1998], a collection of articles on little languages (not all of them DSLs); and Barron [2000], which treats scripting languages (again, not all of them DSLs). Domain analysis, program generators, generative programming techniques, and intentional programming (IP) are treated in Czarnecki and Eisenecker [2000]. Domain analysis and the use of XML, DOM, XSLT, and related languages and tools to generate programs are discussed in Cleaveland [2001]. Domain-specific language development is an important element of the software factories method [Greenfield et al. 2004].

Proceedings of recent workshops and conferences partly or exclusively devoted to DSLs are Kamin [1997]; USENIX [1997, 1999]; HICSS [2001, 2002, 2003]; Lengauer et al. [2004]. Several journals have published special issues on DSLs [Wile and Ramming 1999; Mernik and Lämmel 2001, 2002]. Many of the DSLs used as examples in this article were taken from these sources. A special issue on end-user development is the subject of Sutcliffe and Mehandjiev [2004]. A special issue on program generation, optimization, and platform adaptation is authored by Moura et al. [2005]. There are many workshops and conferences at least partly devoted to DSLs for a particular domain, for example, description of features of telecommunications and other software systems [Gilmore and Ryan 2001]. The annotated DSL bibliography [van Deursen et al. 2000] (78 items) has limited overlap with the references in this article because of our emphasis on general DSL development issues.

2. DSL PATTERNS

2.1. Pattern classification

The following are DSL development phases: *decision*, *analysis*, *design*, *implementation*, and *deployment*. In practice,

Table III. Decision Patterns

Pattern	Description
Notation	Add new or existing domain notation Important subpatterns: • Transform visual to textual notation • Add user-friendly notation to existing API
AVOPT	Domain-specific Analysis, Verification, Optimization, Parallelization, and Transformation
Task automation	Eliminate repetitive tasks
Product line	Specify member of software product line
Data structure representation	Facilitate data description
Data structure traversal	Facilitate complicated traversals
System front-end	Facilitate system configuration
Interaction	Make interaction programmable
GUI construction	Facilitate GUI construction

DSL development is not a simple sequential process, however. The decision process may be influenced by preliminary analysis which, in turn, may have to supply answers to unforeseen questions arising during design, and design is often influenced by implementation considerations.

We associate classes of patterns with each of the development phases except deployment which is beyond the scope of this article. The decision phase corresponds to the "when" part of DSL development, the other phases to the "how" part. *Decision patterns* are common situations that potential developers may find themselves in for which successful DSLs have been developed in the past. In such situations, use of an existing DSL or development of a new one is a serious option. Similarly, *analysis patterns*, *design patterns*, and *implementation patterns* are common approaches to, respectively, domain analysis, DSL design, and DSL implementation. Patterns corresponding to different DSL development phases are independent. For a particular decision pattern, virtually any analysis or design pattern can be chosen, and the same is true for design and implementation patterns. Patterns in the same class, on the other hand, need not be independent but may have some overlap.

We discuss each development phase and the associated patterns in a separate section. Inevitably, there may be some patterns we have missed.

2.2. Decision

Deciding in favor of a new DSL is usually not easy. The investment in DSL development (including deployment) has to pay for itself by more economical software development and/or maintenance later on. As mentioned in Section 1.1, a quantitative treatment of the trade-offs involved is difficult. In practice, short-term considerations and lack of expertise may easily cause indefinite postponement of the decision. Obviously, adopting an existing DSL is much less expensive and requires much less expertise than developing a new one. Finding out about available DSLs may be hard, since DSL information is scattered widely and often buried in obscure documents. Adopting DSLs that are not well publicized might be considered too risky, anyway.

To aid in the decision process, we identify the decision patterns, shown in Table III. Underlying them are general, interrelated concerns such as:

—improved software economics,

—enabling of software development by users with less domain and programming expertise, or even by end-users with some domain, but virtually no programming expertise [Nardi 1993; Sutcliffe and Mehandjiev 2004].

The patterns in Table III may be viewed as more concrete and specific subpatterns of these general concerns. We briefly discuss each decision pattern in turn. Examples for each pattern are given in Table IV.

Table IV. Examples for the Decision Patterns in Table III

Pattern	DSL	Application Domain
Notation	MSC [SDL Forum 2000]	Telecom system specification
• Visual-to-textual	Hawk [Launchbury et al. 1999]	Microarchitecture design
	MSF [Gray and Karsai 2003]	Tool integration
	Verischemelog [Jennings and Beuscher 1999]	Hardware design
• API-to-DSL	SPL [Xiong et al. 2001]	Digital signal processing
	SWUL [Bravenboer and Visser 2004]	GUI construction
AVOPT	AL [Guyer and Lin 1999]	Software optimization
	ATMOL [van Engelen 2001]	Atmospheric modeling
	BDL [Bertrand and Augeraud 1999]	Coordination
	ESP [Kumar et al. 2001]	Programmable devices
	OWL-Light [Dean et al. 2003]	Web ontology
	PCSL [Bruntink et al. 2005]	Parameter checking
	PLAN-P [Thibault et al. 1998]	Network programming
	Teapot [Chandra et al. 1999]	Cache coherence protocols
Task automation	Facile [Schnarr et al. 2001]	Computer architecture
	JAMOOS [Gil and Tsoglin 2001]	Language processing
	lava [Sirer and Bershad 1999]	Software testing
	PSL-DA [Fertalj et al. 2002]	Database applications
	RoTL [Mauw et al. 2004]	Traffic control
	SHIFT [Antoniotti and Göllü 1997]	Hybrid system design
	SODL [Mernik et al. 2001]	Network applications
Product line	GAL [Thibault et al. 1999]	Video device drivers
Data structure representation	ACML [Gondow and Kawashima 2002]	CASE tools
	ASDL [Wang et al. 1997]	Language processing
	DiSTiL [Smaragdakis and Batory 1997]	Container data structures
	FIDO [Klarlund and Schwartzbach 1999]	Tree automata
Data structure traversal	ASTLOG [Crew 1997]	Language processing
	Hancock [Bonachea et al. 1999]	Customer profiling
	S-XML [Clements et al. 2004; Felleisen et al. 2004]	XML processing
	TVL [Gray and Karsai 2003]	Tool integration
System front-end	Nowra [Sloane 2002]	Software configuration
	SSC [Buffenbarger and Gruell 2001]	Software composition
Interaction	CHEM [Bentley 1986]	Drawing chemical structures
	FPIC [Kamin and Hyatt 1997]	Picture drawing
	Fran [Elliott 1999]	Computer animation
	Mawl [Atkins et al. 1999]	Web computing
	Service Combinators [Cardelli and Davies 1999]	Web computing
GUI construction	AUI [Schneider and Cordy 2002]	User interface construction
	HyCom [Risi et al. 2001]	Hypermedia applications

Table V. Analysis Patterns

Pattern	Description
Informal	The domain is analyzed in an informal way.
Formal	A domain analysis methodology is used.
Extract from code	Mining of domain knowledge from legacy GPL code by inspection or by using software tools, or a combination of both.

Notation. The availability of appropriate (new or existing) domain-specific notations is the decisive factor in this case. Two important subpatterns are:

—*Transform visual to textual notation.* There are many benefits to making an existing visual notation available in textual form such as easier composition of large programs or specifications, and enabling of the AVOPT decision pattern discussed next.

—*Add user-friendly notation to an existing API* or turn an API into a DSL.

AVOPT. Domain-specific analysis, verification, optimization, parallelization, and transformation of application programs written in a GPL are usually not feasible because the source code patterns involved are too complex or not well defined. Use of an appropriate DSL makes these operations possible. With continuing developments in chip-level multiprocessing (CMP), domain-specific parallelization will become steadily more important [Kuck 2005]. This pattern overlaps with most of the others.

Task automation. Programmers often spend time on GPL programming tasks that are tedious and follow the same pattern. In such cases, the required code can be generated automatically by an application generator (compiler) for an appropriate DSL.

Product line. Members of a software product line [Weiss and Lay 1999] share a common architecture and are developed from a common set of basic elements. Use of a DSL may often facilitate their specification. This pattern has considerable overlap with both the task automation and system front-end patterns.

Data structure representation. Data-driven code relies on initialized data structures whose complexity may make them difficult to write and maintain. Such structures are often more easily expressed using a DSL.

Data structure traversal. Traversals over complicated data structures can often be expressed better and more reliably in a suitable DSL.

System front-end. A DSL-based front-end may often be used for handling a system's configuration and adaptation.

Interaction. Text- or menu-based interaction with application software often has to be supplemented with an appropriate DSL for the specification of complicated or repetitive input. For example, Excel's interactive mode is supplemented with the Excel macro language to make Excel programmable.

GUI construction. This is often done using a DSL.

2.3. Analysis

In the analysis phase of DSL development, the problem domain is identified and domain knowledge is gathered. Inputs are various sources of explicit or implicit domain knowledge, such as technical documents, knowledge provided by domain experts, existing GPL code, and customer surveys. The output of domain analysis varies widely but consists basically of domain-specific terminology and semantics in more or less abstract form. There is a close link between domain analysis and knowledge engineering which is only beginning to be explored. Knowledge capture, knowledge representation, and ontology development [Denny 2003] are potentially useful in the analysis phase.

The analysis patterns we have identified are shown in Table V. Examples are given in Table VI. Most of the time, domain analysis is done

Table VI. Examples for the Analysis Patterns in Table V
(References and application domains are given in Table IV. The FODA and FAST domain analysis methodologies are discussed in the text.)

Pattern	DSL	Analysis Methodology
Informal	All DSLs in Table IV except:	
Formal	GAL	FAST commonality analysis
	Hancock	FAST
	RoTL	Variability analysis (close to FODA's)
	Service Combinators	FODA (only in this article—see text)
Extract from code	FPIC	Extracted by inspection from PIC implementation
	Nowra	Extracted by inspection from Odin implementation
	PCSL	Extracted by clone detection from proprietary C code
	Verischemelog	Extracted by inspection from Verilog implementation

informally, but sometimes domain analysis methodologies are used. Examples of such methodologies are DARE (Domain Analysis and Reuse Environment) [Frakes et al. 1998], DSSA (Domain-Specific Software Architectures) [Taylor et al. 1995], FAST (Family-Oriented Abstractions, Specification, and Translation) [Weiss and Lay 1999], FODA (Feature-Oriented Domain Analysis) [Kang et al. 1990], ODE (Ontology-based Domain Engineering) [Falbo et al. 2002], or ODM (Organization Domain Modeling) [Simos and Anthony 1998]. To give an idea of the scope of these methods, we explain the FODA and FAST methodologies in somewhat greater detail. Tool support for formal domain analysis is discussed in Section 3.2.

The output of formal domain analysis is a *domain model* consisting of

—a domain definition defining the scope of the domain,
—domain terminology (vocabulary, ontology),
—descriptions of domain concepts,
—feature models describing the commonalities and variabilities of domain concepts and their interdependencies.

How can a DSL be developed from the information gathered in the analysis phase? No clear guidelines exist, but some are presented in Thibault et al. [1999] and Thibault [1998]. Variabilities indicate precisely what information is required to specify an instance of a system. This information must be specified directly in, or be derivable from, a DSL program. Terminology and concepts are used to guide the development of the actual DSL constructs corresponding to the variabilities. Commonalities are used to define the execution model (by a set of common operations) and primitives of the language. Note that the execution model of a DSL is usually much richer than that for a GPL. On the basis of a single domain analysis, many different DSLs can be developed, but all share important characteristics found in the feature model.

For the sake of concreteness, we apply the FODA domain analysis methodology [Kang et al. 1990] to the service combinator DSL discussed in Cardelli and Davies [1999]. The latter's goal is to reproduce human behavior, while accessing and manipulating Web resources such as reaction to slow transmission, failures, and many simultaneous links. FODA requires construction of a feature model capturing commonalities (mandatory features) and variabilities (variable features). More specifically, such a model consists of

—a feature diagram representing a hierarchical decomposition of features and their character, that is, whether they are mandatory, alternative, or optional,
—definitions of the semantics of features,
—feature composition rules describing which combinations of features are valid or invalid,
—reasons for choosing a feature.

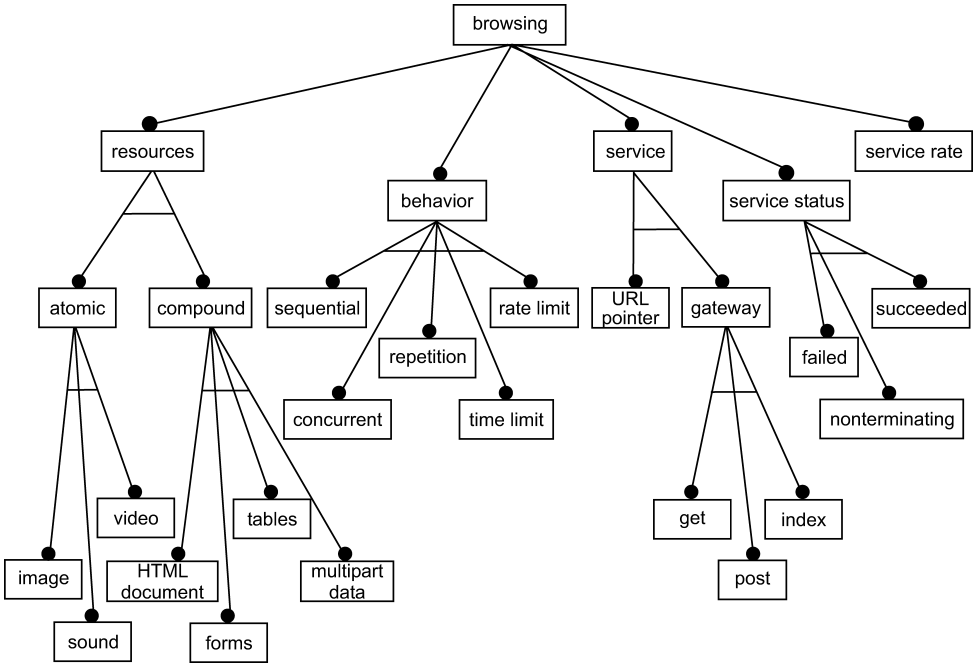


Fig. 1. Feature diagram for Web browsing.

A common feature of a concept is a feature present in all instances of the concept. All mandatory features whose parent is the concept are common features. Also, all mandatory features whose parents are common are themselves common. A variable feature is either optional or alternative (one of, more of). Nodes in the feature diagram to which these features are attached are called variation points.

In the case of our example DSL, the domain consists of resources, browsing behavior, and services (type, status, and rate). Resources can be atomic or compound, access to the resource (service) can be through a URL pointer or a gateway, and browsing behavior can be sequential, concurrent, repetitive, limited by accessing time, or rate. Service has a rate and status (succeeded, failed, or nonterminating). A corresponding feature diagram is shown in Figure 1. The first step in designing the DSL is to look into variabilities and commonalities in the feature diagram. Variable parts must be specified directly in or be derivable from DSL programs. It is clear that type of service (URL pointer or gateway) and browsing behavior have to be specified in DSL programs. Service status and service rate will be examined and computed while running a DSL program. Therefore, both will be built into the execution model. Type of resource (atomic or compound) are actually types of values that exist during the execution of a DSL program. The basic syntax proposed in Cardelli and Davies [1999]

```
S ::= url(String)       // basic services
| gateway get (String+)
| gateway post (String+)
| index(String, String)
| S1 ? S2               // sequential execution
| S1 '|' S2            // concurrent execution
| timeout(Real, S)     // timeout combinator
| limit(Real, Real, S) // rate limit combinator
| repeat(S)            // repetition
| stall                // nontermination
| fail                 // failure
```

closely resembles our feature diagram. The syntax can later be extended with abstractions and binding.

Table VII. Design Patterns

Pattern	Description
Language exploitation	DSL uses (part of) existing GPL or DSL. Important subpatterns: • Piggyback: Existing language is partially used • Specialization: Existing language is restricted • Extension: Existing language is extended
Language invention	A DSL is designed from scratch with no commonality with existing languages
Informal	DSL is described informally
Formal	DSL is described formally using an existing semantics definition method such as attribute grammars, rewrite rules, or abstract state machines

Another domain analysis methodology is FAST (Family-Oriented Abstractions, Specification, and Translation) [Coplien et al. 1998]. FAST is a software development process applying product-line architecture principles, so it relates directly to the product-line decision pattern. A common platform is specified for a family of software products. It is based on the similarities and differences between products. The FAST method consists of the following activities: domain qualification, domain engineering, application engineering, project management, and family change.

During domain engineering, the domain is analyzed and then implemented as a set of domain-specific reusable components. The purpose of domain analysis in FAST is to capture common knowledge about the domain and guide reuse of the implemented components. Domain analysis involves the following steps: decision model definition, commonality analysis, domain design, application modeling language design, creation of standard application engineering process design, and development of the application engineering design environment. An important task of domain analysis is commonality analysis which identifies useful abstractions that are common to all family members. Commonalities are the main source of reuse, thus the emphasis is on finding common parts. Besides the commonalities, variabilities are also discovered during commonality analysis. Variabilities indicate potential sources of change over the lifetime of the family. Commonalities and variabilities in FAST are specified as a structured list. For every variable property, the range of variability as well as binding time are specified. Commonality analysis is later used in designing an application modeling language (AML) which is used to generate a family member from specifications.

2.4. Design

Approaches to DSL design can be characterized along two orthogonal dimensions: the relationship between the DSL and existing languages, and the formal nature of the design description. This dichotomy is reflected in the design patterns in Table VII and the corresponding examples in Table VIII.

The easiest way to design a DSL is to base it on an existing language. Possible benefits are easier implementation (see Section 2.5) and familiarity for users, but the latter only applies if users are also programmers in the existing language which may not be the case. We identify three patterns of design based on an existing language. First, we can *piggyback* domain-specific features on part of an existing language. A related approach restricts the existing language to provide a *specialization* targeted at the problem domain. The difference between these two patterns is really a matter of how rigid the barrier is between the DSL and the rest of the existing language. Both of these approaches are often used when a notation is already widely known. For example, many DSLs contain arithmetic expressions which are usually written in the infix-operator style of mathematics.

Another approach is to take an existing language and extend it with new features

Table VIII. Examples for the Design Patterns in Table VII
(References and application domains are given in Table IV.)

Pattern	DSL
Language exploitation	
• Piggyback	ACML, ASDL, BDL, ESP, Facile, Hancock, JAMOOS, lava, Mawl, PSL-DA, SPL, SSC, Teapot
• Specialization	OWL-Light
• Extension	AUI, DiSTiL, FPIC, Fran, Hawk, HyCom, Nowra, PLAN-P, SWUL, S-XML, Verischemelog
Language invention	AL, ASTLOG, ATMOL, CHEM, GAL, FIDO, MSF, RoTL, Service Combinators, SHIFT, SODL, TVL
Informal	All DSLs in Table IV except:
Formal	ATMOL, ASTLOG, BDL, FIDO, GAL, OWL-Light, PLAN-P, RoTL, Service Combinators, SHIFT, SODL, SSC

that address domain concepts. In most applications of this pattern, the existing language features remain available. The challenge is to integrate the domain-specific features with the rest of the language in a seamless fashion.

At the other end of the spectrum is a DSL whose design bears no relationship to any existing language. In practice, development of this kind of DSL can be extremely difficult and is hard to characterize. Well-known GPL design criteria such as readability, simplicity, orthogonality, the design principles listed by Brooks [1996], and Tennent's design principles [1977] retain some validity for DSLs. However, the DSL designer has to keep in mind both the special character of DSLs as well as the fact that users need not be programmers. Since ideally the DSL adopts established notations of the domain, the designer should suppress a tendency to improve them. As stated in Wile [2004], one of the lessons learned from real DSL experiments is:

> *Lesson T2:* You are almost never designing a programming language.
> Most DSL designers come from language design backgrounds. There the admirable principles of orthogonality and economy of form are not necessarily well-applied to DSL design. Especially in catering to the pre-existing jargon and notations of the domain, one must be careful not to embellish or over-generalize the language.
> *Lesson T2 Corollary:* Design only what is necessary. Learn to recognize your tendency to over-design.

Once the relationship to existing languages has been determined, a DSL designer must turn to specifying the design before implementation. We distinguish between *informal* and *formal* designs. In an informal design the specification is usually in some form of natural language, probably including a set of illustrative DSL programs. A formal design consists of a specification written using one of the available semantic definition methods [Slonneger and Kurtz 1995]. The most widely used formal notations include regular expressions and grammars for syntax specifications, and attribute grammars, rewrite systems, and abstract state machines for semantic specification.

Clearly, an informal approach is likely to be easiest for most people. A formal approach should not be discounted, however. Development of a formal description of both syntax and semantics can bring problems to light before the DSL is actually implemented. Furthermore, formal designs can be implemented automatically by language development systems and tools, thereby significantly reducing the implementation effort (Section 3).

As mentioned in the beginning of this section, design patterns can be characterized in terms of two orthogonal dimensions: language invention or language exploitation (extension, specialization, or piggyback), and informal or formal description. Figure 2 indicates the position of the DSLs from Table VIII in the design pattern plane. We note that formal description is used more often than informal description when a DSL is designed using the language invention pattern. The opposite is true

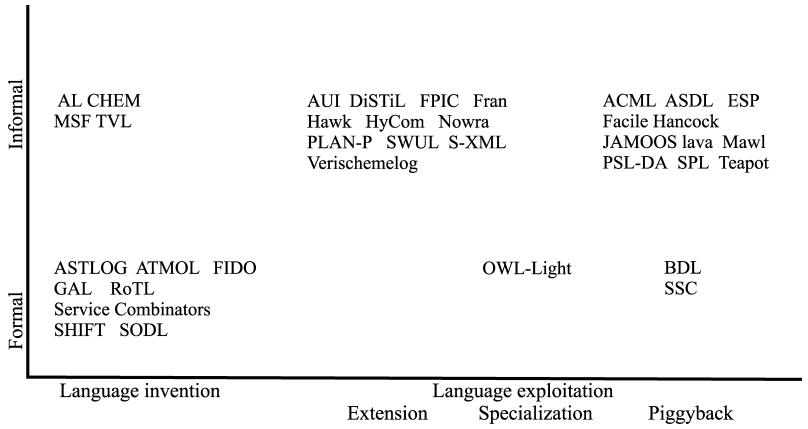


Fig. 2. The DSLs from Table VIII in the design pattern plane.

when a DSL is designed using language exploitation.

2.5. Implementation

2.5.1. Patterns. When an (executable) DSL is designed, the most suitable implementation approach should be chosen. This may be obvious, but in practice it is not, mainly because of the many DSL implementation techniques that have no useful counterpart for GPLs. These DSL-specific techniques are less well known, but can make a big difference in the total effort that has to be invested in DSL development. The implementation patterns we have identified are shown in Table IX. We discuss some of them in more detail. Examples are given in Table X.

Interpretation and compilation are as relevant for DSLs as for GPLs, even though the special character of DSLs often makes them amenable to other, more efficient implementation methods such as preprocessing and embedding. This viewpoint is at variance with Spinellis [2001], where it is argued that DSL development is radically different from GPL development since the former is usually just a small part of a project, and hence DSL development costs have to be modest. This is not always the case, however, and interpreters and compilers or application generators are widely used in practice.

Macros and subroutines are the classic language extension mechanisms used for DSL implementation. Subroutines have given rise to implementation by embedding, while macros are handled by preprocessing. A recent survey of macros is given in Braband and Schwartzbach [2002]. Macro expansion is often independent of the syntax of the base language, and the syntactical correctness of the expanded result is not guaranteed but is checked at a later stage by the interpreter or compiler. This situation is typical for preprocessors.

C++ supports a language-specific preprocessing approach, *template metaprogramming* [Veldhuizen 1995b; Veldhuizen 1995a]. It uses template expansion to achieve compile-time generation of domain-specific code. Significant mileage has been made out of template metaprogramming to develop mathematical libraries for C++ which have familiar domain notation using C++ user-definable operator notation and overloading but also achieve good performance. An example is Blitz++ [Veldhuizen 2001].

In the embedding approach, a DSL is implemented by extending an existing GPL (the host language) by defining specific abstract data types and operators. A domain-specific problem can then be described with these new constructs. Therefore, the new language has all the power of the host language, but an application

Table IX. Implementation Patterns for Executable DSLs

Pattern	Description
Interpreter	DSL constructs are recognized and interpreted using a standard fetch-decode-execute cycle. This approach is appropriate for languages having a dynamic character or if execution speed is not an issue. The advantages of interpretation over compilation are greater simplicity, greater control over the execution environment, and easier extension.
Compiler/application generator	DSL constructs are translated to base language constructs and library calls. A complete static analysis can be done on the DSL program/specification. DSL compilers are often called application generators.
Preprocessor	DSL constructs are translated to constructs in an existing language (the *base language*). Static analysis is limited to that done by the base language processor. Important subpatterns: • Macro processing: Expansion of macro definitions. • Source-to-source transformation: DSL source code is transformed (translated) into base language source code. • Pipeline: Processors successively handling sublanguages of a DSL and translating them to the input language of the next stage. • Lexical processing: Only simple lexical scanning is required, without complicated tree-based syntax analysis.
Embedding	DSL constructs are embedded in an existing GPL (the *host language*) by defining new abstract data types and operators. Application libraries are the basic form of embedding.
Extensible compiler/ interpreter	A GPL compiler/interpreter is extended with domain-specific optimization rules and/or domain-specific code generation. While interpreters are usually relatively easy to extend, extending compilers is hard unless they were designed with extension in mind.
Commercial Off-The-Shelf (COTS)	Existing tools and/or notations are applied to a specific domain.
Hybrid	A combination of the above approaches.

Table X. Examples for the Implementation Patterns in Table IX
(References and application domains are given in Table IV.)

Pattern	DSL
Interpreter	ASTLOG, Service Combinators
Compiler/application generator	AL, ATMOL, BDL, ESP, Facile, FIDO, Hancock, JAMOOS, lava, Mawl, PSL-DA, RoTL, SHIFT, SODL, SPL, Teapot
Preprocessor	
• Macro processing	S-XML
• Source-to-source transformation	ADSL, AUI, MSF, SWUL, TVL
• Pipeline	CHEM
• Lexical processing	SSC
Embedding	FPIC, Fran, Hawk, HyCom, Nowra, Verischemelog
Extensible compiler/interpreter	DiSTiL
Commercial Off-The-Shelf (COTS)	ACML, OWL-Light
Hybrid	GAL, PLAN-P

engineer can become a programmer without learning too much of it. To approximate domain-specific notations as closely as possible, the embedding approach can use any features for user-definable operator syntax the host language has to offer. For example, it is common to develop C++ class libraries where the existing operators are overloaded with domain-specific semantics. Although this technique is quite powerful, pitfalls exist in overloading familiar operators to have unfamiliar semantics. Although the host language in the embedding approach can be any general-purpose language, functional languages are often appropriate as shown by many researchers [Hudak 1998; Kamin 1998]. This is due to functional language features such as lazy evaluation, higher-order functions, and strong typing with polymorphism and overloading.

Extending an existing language implementation can also be seen as a form of embedding. The difference is usually a matter of degree. In an interpreter or compiler approach, the implementation would usually only be extended with a few features such as new data types and operators for them. For a proper embedding, the extensions might encompass full-blown domain-specific language features. In both settings, however, extending implementations is often very difficult. Techniques for doing so in a safe and modular fashion are still the subject of much research. Since compilers are particularly hard to extend, much of this work is aimed at preprocessors and extensible compilers allowing for the addition of domain-specific optimization rules and/or domain-specific code generation. We mention user-definable optimization rules in the CodeBoost C++ preprocessor [Bagge and Haveraaen 2003] and in the Simplicissimus GCC compiler plug-in [Schupp et al. 2001], the IBM Montana extensible C++ programming environment [Soroker et al. 1997], the user-definable optimization rules in the GHC Haskell compiler [Peyton Jones et al. 2001], and the exploitation of domain-specific semantics of application libraries in the Broadway compiler [Guyer and Lin 2005]. Some extensible compilers such as OpenC++ [Chiba 1995], support a *metaobject protocol*. This is an object-oriented interface for specifying language extensions and transformations [Kiczales et al. 1991].

The COTS-based approach builds a DSL around existing tools and notations. Typically this approach involves applying existing functionality in a restricted way, according to domain rules. For example, the general-purpose Powerpoint tool has been applied in a domain-specific setting for diagram editing [Wile 2001]. The current prominence of XML-based DSLs is another instance of this approach [Gondow and Kawashima 2002; Parigot 2004]. For an XML-based DSL, grammar is described using a DTD or XML scheme where nonterminals are analogous to elements and terminals to data content. Productions are like element definitions where the element name is the left-hand side and the content model is the right-hand side. The start symbol is analogous to the document element in a DTD. Using a DOM parser or SAX (Simple API for XML) tool, parsing comes for free. Since the parse tree can be encoded in XML as well, XSLT transformations can be used for code generation. Therefore, XML and XML tools can be used to implement a programming language compiler [Germon 2001].

Many DSL endeavors apply a number of these approaches in a hybrid fashion. Thus the advantages of different approaches can be exploited. For instance, embedding can be combined with user-defined domain-specific optimization in an extensible compiler, and the interpreter and compiler approach become indistinguishable in some settings (see next section).

2.5.2. Implementation Trade-Offs. Advantages of the interpreter and compiler or application generator approaches are:

—DSL syntax can be close to the notations used by domain experts,

—good error reporting is possible,

—domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) is possible.

Some of its disadvantages are:

—the development effort is large because a complex language processor must be implemented,

—the DSL is more likely to be designed from scratch, often leading to incoherent designs compared with exploitation of an existing language,

—language extension is hard to realize because most language processors are not designed with extension in mind.

However, these disadvantages can be minimized or eliminated altogether when a language development system or toolkit is used so that much of the work of the language processor construction is automated. This presupposes a formal approach to DSL design and implementation. Automation support is discussed further in Section 3.

We now turn to the embedded approach. Its advantages are:

—development effort is modest because an existing implementation can be reused,

—it often produces a more powerful language than other methods since many features come for free,

—host language infrastructure can be reused (development and debugging environments: editors, debuggers, tracers, profilers, etc.),

—user training costs might be lower since many users may already know the host language.

Disadvantages of the embedded approach are:

—syntax is far from optimal because most languages do not allow arbitrary syntax extension,

—overloading existing operators can be confusing if the new semantics does not have the same properties as the old,

—bad error reporting because messages are in terms of host language concepts instead of DSL concepts,

—domain-specific optimizations and transformations are hard to achieve so efficiency may be affected, particularly when embedding in functional languages [Kamin 1998; Sloane 2002].

Advocates of the embedded approach often criticize DSLs implemented by the interpreter or compiler approach in that too much effort is put into syntax design, whereas the language semantics tends to be poorly designed and cannot be easily extended with new features [Kamin 1998]. However, the syntax of a DSL is extremely important and should not be underestimated. It should be as close as possible to the notation used in a domain.

In the functional setting, and in particular if Haskell is used, some of these shortcomings can be reduced by using *monads* to modularize the language implementation [Hudak 1998]. Domain-specific optimizations can be achieved using approaches such as user-defined transformation rules in the GHC compiler [Peyton Jones et al. 2001] or a form of whole-program transformation called *partial evaluation* [Jones et al. 1993; Consel and Marlet 1998]. In C++, template metaprogramming can be used, and user-defined domain-specific optimization is supported by various preprocessors and compilers. See the references in Section 2.5.1.

The decision diagram on how to proceed with DSL implementation (Figure 3) shows when a particular implementation approach is more appropriate. If the DSL is designed from scratch with no commonality with existing languages (invention pattern), the recommended approach is to implement it by embedding, unless domain-specific analysis, verification, optimization, parallelization, or transformation (AVOPT) is required, a domain-specific notation must be strictly obeyed, or the user community is expected to be large.

If the DSL incorporates (part of) an existing language, one would like to reuse (the corresponding part of) the existing language's implementation as well. Apart from this, various implementation

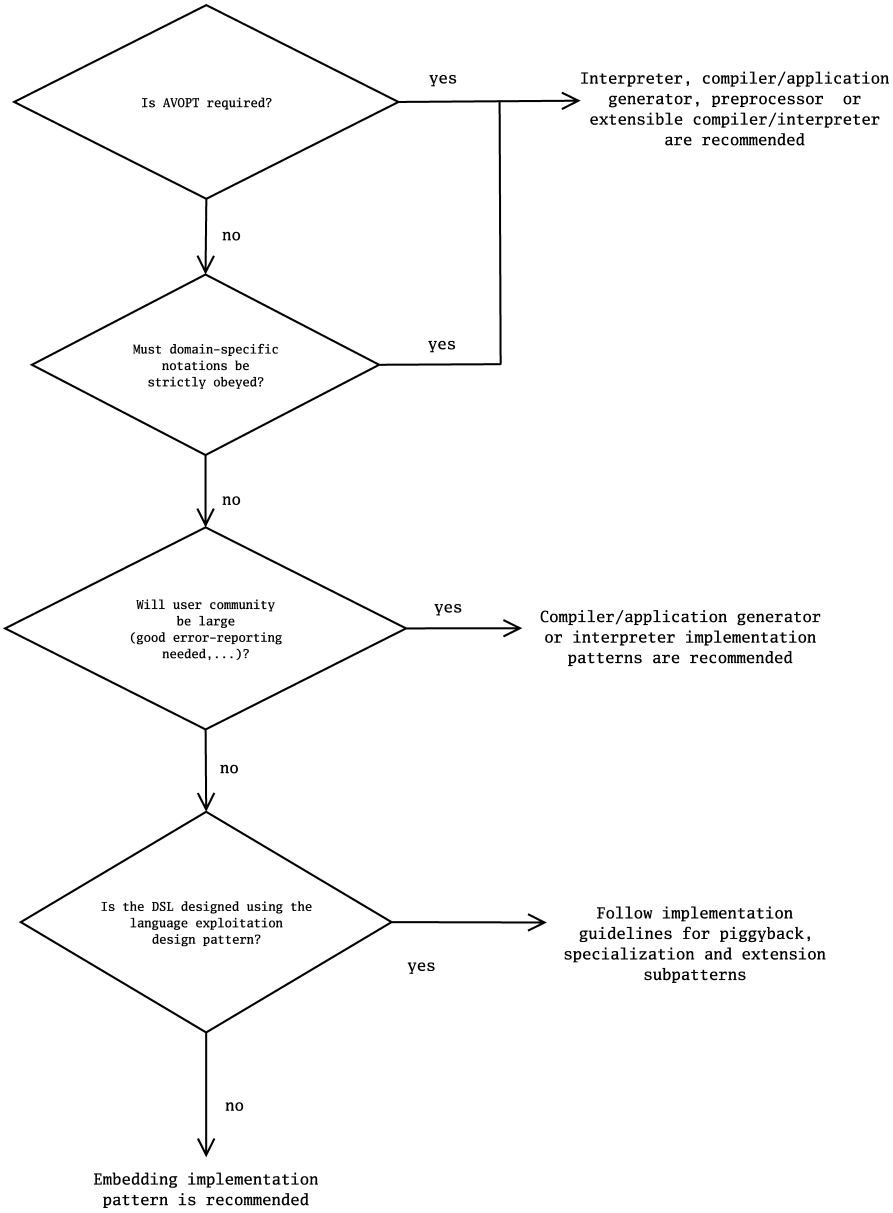


Fig. 3. Implementation guidelines.

Table XI. Pattern Classification Proposed by Spinellis [2001]

Pattern Class	Description
Creational pattern	DSL creation
Structural pattern	Structure of system involving a DSL
Behavioral pattern	DSL interactions

patterns may apply, depending on the language exploitation subpattern used. A piggyback or specialization design can be implemented using an interpreter, compiler or application generator, or preprocessor, but embedding or use of an extensible compiler or interpreter are not suitable,

Table XII. Creational Patterns

Pattern	Description
Language extension	DSL extends existing language with new datatypes, new semantic elements, and/or new syntax.
Language specialization	DSL restricts existing language for purposes of safety, static checking, and/or optimization.
Source-to-source transformation	DSL source code is transformed (translated) into source code of existing language (the base language).
Data structure representation	Data-driven code relies on initialized data structures whose complexity may make them difficult to write and maintain. These structures are often more easily expressed using a DSL.
Lexical processing	Many DSLs may be designed in a form suitable for recognition by simple lexical scanning.

Table XIII. Structural Patterns

Pattern	Description
Piggyback	DSL has elements, for instance, expressions in common with existing language. DSL processor passes those elements to existing language processor.
System front-end	A DSL based front-end may often be used for handling a system's configuration and adaptation.

Table XIV. Behavioral Patterns

Pattern	Description
Pipeline	Pipelined processors successively handling sublanguages of a DSL and translating them to input language of next stage.

although specialization can be done using an extensible compiler/interpreter in some languages (Smalltalk, for instance). In the case of piggyback, a preprocessor transforming the DSL to the language it piggybacks on is best from the viewpoint of implementation reuse, but preprocessing has serious shortcomings in other respects. A language extension design can be implemented using all of the previously mentioned implementation patterns. From the viewpoint of implementation reuse, embedding and use of an extensible compiler/interpreter are particularly attractive in this case.

If more than one implementation pattern applies, the one having the highest ratio of benefit (see discussion in this section) to implementation effort is optimal, unless, as in the language invention case, AVOPT is required, a domain-specific notation must be strictly obeyed, or the user community is expected to be large. As already mentioned, a compiler or application generator scores the worst in terms of implementation effort. Less costly are (in descending order) the interpreter, preprocessing, extensible compiler or interpreter, and embedding. On the other hand, a compiler or application generator and interpreter score best as far as benefit to DSL users is concerned. Less benefit is obtained from (in descending order) extensible compiler or interpreter, embedding, and preprocessing. In practice, such a cost-benefit analysis is rarely performed, and the decision is driven only by implementor experience. Of course, the latter should be taken into account, but it is not the only relevant factor.

2.6. Comparison With Other Classifications

We start by comparing our patterns with those proposed in Spinellis [2001]. Closely following Gamma et al. [1995], Spinellis distinguishes three classes of DSL patterns as shown in Table XI. The specific patterns for each class are summarized in Tables XII, XIII, and XIV. Most patterns are creational. The piggyback pattern might be classified as creational as well since it is very similar to language extension. This would leave only a single pattern in each of the other two categories.

First, it should be noted that Spinellis's [2001] patterns do not include traditional GPL design and implementation techniques, while ours do, since we consider them to be as relevant for DSLs as for GPLs. Second, Spinellis's

Table XV. Correspondence of Spinellis's [2001] Patterns With Ours
(Since our patterns have a wider scope, many of them have no counterpart in Spinellis's classification. These are not shown in the right-hand column.)

Spinellis's Pattern	Our Pattern
Creational: language extension	Design: language exploitation (extension)
Creational: language specialization	Design: language exploitation (specialization)
Creational: source-to-source transformation	Implementation: preprocessing (source-to-source transformation)
Creational: data structure representation	Decision: data structure representation
Creational: lexical processing	Implementation: preprocessing
Structural: piggyback	Design: language exploitation (piggyback)
Structural: system front-end	Decision: system front-end
Behavioral: pipeline	Implementation: preprocessing (pipeline)

classification does not correspond in an obvious way to our classification in decision, analysis, design, and implementation patterns. The latter are all basically creational, but cover a wider range of creation-related activities than Spinellis's patterns.

The correspondence of Spinellis's [2001] patterns with ours is shown in Table XV. Since our patterns have a wider scope, many of them have no counterpart in Spinellis's classification. These are not shown in the right-hand column. We have retained the terminology used by Spinellis whenever appropriate.

Another classification of DSL development approaches is given in Wile [2001], namely, full language design, language extension, and COTS-based approaches. Since each approach has its own pros and cons, the author discusses them with respect to three kinds of issues, DSL-specific, GPL support, and pragmatic support issues. Finally, the author shows how a hybrid development approach can be used.

3. DSL DEVELOPMENT SUPPORT

3.1. Design and Implementation Support

As we have seen, DSL development is hard, requiring both domain knowledge and language development expertise. The development process can be facilitated by using a *language development system* or *toolkit*. Some systems and toolkits that have actually been used for DSL development are listed in Table XVI. They have widely different capabilities and are in many different stages of development but are based on the same general principle: they generate tools from language descriptions [Heering and Klint 2000]. The tools generated may vary from a consistency checker and interpreter to an integrated development environment (IDE), consisting of a syntax-directed editor, a prettyprinter, an (incremental) consistency checker, analysis tools, an interpreter or compiler/application generator, and a debugger for the DSL in question (assuming it is executable). As noted in Section 1.2, nonexecutable DSLs may also benefit from various kinds of tool support such as syntax-directed editors, prettyprinters, consistency checkers, and analyzers. These can be generated in the same way.

Some of these systems support a specific DSL design methodology, while others have a largely methodology-independent character. For instance, Sprint [Consel and Marlet 1998] assumes an interpreter for the given DSL and then uses partial evaluation to remove the interpretation overhead by automatically transforming a DSL program into a compiled program. Other systems, such as ASF+SDF [van den Brand et al. 2001], DMS [Baxter et al. 2004], and Stratego [Visser 2003], would not only allow an interpretive definition of the DSL, but would also accept a transformational or translational one. On the other hand, they might not support partial evaluation of a DSL interpreter given a specific program.

The input into these systems is a description of various aspects of the DSL that are developed in terms of specialized metalanguages. Depending on the type of DSL, some important language

Table XVI. Some Language Development Systems and Toolkits That Have Been Used for DSL Development

System	Developer
ASF+SDF [van den Brand et al. 2001]	CWI/University of Amsterdam
AsmL [Glässer et al. 2002]	Microsoft Research, Redmond
DMS [Baxter et al. 2004]	Semantic Designs, Inc.
Draco [Neighbors 1984]	University of California, Irvine
Eli [Gray et al. 1992]	University of Colorado, University of Paderborn, Macquarie University
Gem-Mex [Anlauff et al. 1999]	University of L'Aquila
InfoWiz [Nakatani and Jones 1997]	Bell Labs/AT&T Labs
JTS [Batory et al. 1998]	University of Texas at Austin
Khepera [Faith et al. 1997]	University of North Carolina
Kodiyak [Herndon and Berzins 1988]	University of Minnesota
LaCon [Kastens and Pfahler 1998]	University of Paderborn (LaCon uses Eli as back-end—see above)
LISA [Mernik et al. 1999]	University of Maribor
`metafront` [Braband et al. 2003]	University of Aarhus
Metatool [Cleaveland 1988]	Bell Labs
POPART [Wile 1993]	USC/Information Sciences Institute
SmartTools [Attali et al. 2001]	INRIA Sophia Antipolis
`smgn` [Kienle and Moore 2002]	Intel Compiler Lab/University of Victoria
SPARK [Aycock 2002]	University of Calgary
Sprint [Consel and Marlet 1998]	LaBRI/INRIA
Stratego [Visser 2003]	University of Utrecht
TXL [Cordy 2004]	University of Toronto/Queen's University at Kingston

Table XVII. Development Support Provided by Current Language Development Systems and Toolkits for DSL Development Phases/Pattern Classes

Development phase/ Pattern class	Support Provided
Decision	None
Analysis	Not yet integrated—see Section 3.2
Design	Weak
Implementation	Strong

aspects are syntax, prettyprinting, consistency checking, analysis, execution, translation, transformation, and debugging. It so happens that the metalanguages used for describing these aspects are themselves DSLs for the particular aspect in question. For instance, DSL syntax is usually described in something close to BNF, the de facto standard for syntax specification (Table I). The corresponding tool generated by the language development system is a parser.

Although the various specialized metalanguages used for describing language aspects differ from system to system, they are often (but not always) rule based. For instance, depending on the system, the consistency of programs or scripts may have to be checked in terms of *attributed syntax rules* (an extension of BNF), *conditional rewrite rules*, or *transition rules*. See, for instance, Slonneger and Kurtz [1995] for further details.

The level of support provided by these systems in various phases of DSL development is summarized in Table XVII. Their main strength lies in the implementation phase. Support of DSL design tends to be weak. Their main assets are the metalanguages they support and, in some cases, a meta-environment to aid in constructing and debugging language descriptions but they have little built-in knowledge of language concepts or design rules. Furthermore, to the best of our knowledge, none of them provides any support in the analysis or decision phase. Analysis support tools are discussed in Section 3.2.

Examples of DSL development using the systems in Table XVI are given in Table XVIII. They cover a wide range of application domains and implementation patterns. The Box prettyprinting metalanguage is an example of a DSL developed with a language development system (in this case ASF+SDF) for later use as one of the metalanguages of the system itself. This is common practice. The

Table XVIII. Examples of DSL Development using the Systems in Table XVI

System Used	DSL	Application Domain
ASF+SDF	Box [van den Brand and Visser 1996]	Prettyprinting
	Risla [van Deursen and Klint 1998]	Financial products
AsmL	UPnP [UPnP 2003]	Networked device protocol
	XLANG [Thatte 2001]	Business protocols
DMS	(Various) [Baxter et al. 2004]	Program transformation
	(Various) [Baxter et al. 2004]	Factory control
Eli	Maptool [Kadhim and Waite 1996]	Grammar mapping
	(Various) [Pfahler and Kastens 2001]	Class generation
Gem-Mex	Cubix [Kutter et al. 1998]	Virtual data warehousing
JTS	Jak [Batory et al. 1998]	Syntactic transformation
LaCon	(Various) [Kastens and Pfahler 1998]	Data model translation
LISA	SODL [Mernik et al. 2001]	Network applications
SmartTools	LML [Parigot 2004]	GUI programming
	BPEL [Courbis and Finkelstein 2004]	Business process description
`smgn`	Hoof [Kienle and Moore 2002]	Compiler IR specification
	IMDL [Kienle and Moore 2002]	Software reengineering
SPARK	Guide [Levy 1998]	Web programming
	CML2 [Raymond 2001]	Linux kernel configuration
Sprint	GAL [Thibault et al. 1999]	Video device drivers
	PLAN-P [Thibault et al. 1998]	Network programming
Stratego	Autobundle [de Jonge 2002]	Software building
	CodeBoost [Bagge and Haveraaen 2003]	Domain-specific C++ optimization

metalanguages for syntax, prettyprinting, attribute evaluation, and program transformation used by DMS were all implemented using DMS, and the Jak transformational metalanguage for specifying the semantics of a DSL or domain-specific language extension in the Jakarta Tool Suite (JTS) was also developed using JTS.

3.2. Analysis Support

The language development toolkits and systems discussed in the previous section do not provide support in the analysis phase of DSL development. Separate frameworks and tools for this have been or are being developed, however. Some of them are listed in Table XIX. We have included a short description of each entry, largely taken from the reference given for it. The fact that a framework or tool is listed does not necessarily mean it is in use or even exists.

As noted in Section 2.3, the output of domain analysis consists basically of domain-specific terminology and semantics in more or less abstract form. It may range from a feature diagram (see FDL entry in Table XIX) to a domain implementation consisting of a set of domain-specific reusable components (see DARE entry in Table XIX), or a theory in the case of scientific domains. An important issue is how to link formal domain analysis with DSL design and implementation. The possibility of linking DARE directly to the Metatool metagenerator (i.e., application generator) [Cleaveland 1988] is mentioned in Frakes [1998].

4. CONCLUSIONS AND OPEN PROBLEMS

DSLs will never be a solution to all software engineering problems, but their application is currently unduly limited by a lack of reliable knowledge available to (potential) DSL developers. To help remedy this situation, we distinguished five phases of DSL development and identified patterns in each phase, except deployment. These are summarized in Table XX. Furthermore, we discussed language development systems and toolkits that can be used to facilitate the development process especially its later phases.

Our survey also showed many opportunities for further work. As indicated in Table XVII, for instance, there are serious gaps in the DSL development support chain. More specifically, some of the issues needing further attention follow.

Decision. Can useful computer-aided decision support be provided? If so,

Table XIX. Some Domain Analysis Frameworks and Tools

Analysis Framework or Tool	Description
Ariadne [Simos and Anthony 1998]	ODM support framework enabling domain practitioners to collaboratively develop and evolve their own semantic models, and to compose and customize applications incorporating these models as first-class architectural elements.
DARE [Frakes et al. 1998]	Supports the capture of domain information from experts, documents, and code in a domain. Captured domain information is stored in a domain book that will typically contain a generic architecture for the domain and domain-specific reusable components.
DOMAIN [Tracz and Coglianese 1995]	DSSA [Taylor et al. 1995] support framework consisting of a collection of structured editors and a hypertext/media engine that allows the user to capture, represent, and manipulate various types of domain knowledge in a hyper-web. DOMAIN supports a "scenario-based" approach to domain analysis. Users enter scenarios describing the functions performed by applications in the domain of interest. The text in these scenarios can then be used (in a semi-automated manner) to develop a domain dictionary, reference requirements, and domain model, each of which are supported by their own editor.
FDL [van Deursen and Klint 2002]	The Feature Description Language (FDL) is a textual representation of feature diagrams, which are a graphical notation for expressing assertions (propositions, predicates) about systems in a particular application domain. These were introduced in the FODA [Kang et al. 1990] domain analysis methodology. (FDL is an example of the visual-to-textual transformation subpattern in Table III.)
ODE editor [Falbo et al. 2002]	Ontology editor supporting ODE—see also [Denny 2003].

Table XX. Summary of DSL Development Phases and Corresponding Patterns

Development Phase	Pattern
Decision (Section 2.2)	Notation
	AVOPT
	Task automation
	Product line
	Data structure representation
	Data structure traversal
	System front-end
	Interaction
	GUI construction
Analysis (Section 2.3)	Informal
	Formal
	Extract from code
Design (Section 2.4)	Language exploitation
	Language invention
	Informal
	Formal
Implementation (Section 2.5)	Interpreter
	Compiler/application generator
	Preprocessor
	Embedding
	Extensible compiler/ interpreter
	COTS
	Hybrid

its integration in existing language development systems or toolkits (Table XVI) might yield additional advantages.

Analysis. Further development and integration of domain analysis support tools. As noted in Section 2.3, there is a close link with knowledge engineering. Existing knowledge engineering tools and frameworks may be useful directly or act as inspiration for further developments in this area. An important issue is how to link formal domain analysis with DSL design and implementation.

Design and Implementation. How can DSL design and implementation be made easier for domain experts not versed in GPL development? Some approaches (not mutually exclusive) are:

—Building DSLs in an incremental, modular, and extensible way from parameterized language building blocks. This

is of particular importance for DSLs since they change more frequently than GPLs [Bosch and Dittrich; Wile 2001]. Progress in this direction is being made [Anlauff et al. 1999; Consel and Marlet 1998; Hudak 1998; Mernik et al. 2000].

—A related issue is how to combine different parts of existing GPLs and DSLs into a new DSL. For instance, in the Microsoft .NET framework, many GPLs are compiled to the Common Language Runtime (CLR) [Gough 2002]. Can this be helpful in including selected parts of GPLs into a new DSL?

—Provide pattern aware development support. The Sprint system [Consel and Marlet 1998], for instance, provides partial evaluation support for the interpreter pattern (see Section 3.1). Other patterns might benefit from specialized support as well. Embedding support is discussed separately in the next paragraph.

—Reduce the need for learning some of the specialized metalanguages of language development systems by supporting description by example (DBE) of selected language aspects like syntax or prettyprinting. The user-friendliness of DBE is due to the fact that examples of intended behavior do not require a specialized metalanguage, or possibly only a small part of it. Grammar inference from example sentences, for instance, may be viable especially since many DSLs are small. This is certainly no new idea [Crespi-Reghizzi et al. 1973; Nardi 1993], but it remains to be realized. Some preliminary results are reported in Črepinšek et al. [2005].

—How can DSL development tools generated by language development systems and toolkits be integrated with other software development tools? Using a COTS-based approach, XML technologies such as DOM and XML-parsers have great potential as a uniform data interchange format for CASE tools. See also Badros [2000] and Cleaveland [2001].

Embedding. GPLs should provide more powerful support for embedding DSLs, both syntactically and semantically. Some issues are:

—Embedding suffers from the very limited user-definable syntax offered by GPLs. Perhaps surprisingly, there has been no trend toward more powerful user-definable syntax in GPLs over the years. In fact, just the opposite has happened. Macros and user-definable operators have become less popular. Java has no user-definable operators at all. On the other hand, some of the language development systems in Table XVI, such as ASF+SDF and to some extent Stratego, support metalanguages featuring fully general user-definable context-free syntax. Although these metalanguages cannot compete directly with GPLs as embedding hosts as far as expressiveness and efficiency are concerned, they can be used to express a source-to-source transformation to translate user-defined DSL syntax embedded in a GPL to appropriate API calls. See Bravenboer and Visser [2004] for an extensive discussion of this approach.

—Improved embedding support is not only a matter of language features, but also of language implementation and, in particular, of preprocessors or extensible compilers allowing the addition of domain-specific optimization rules and/or domain-specific code generation. See the references given in Section 2.5.1 and Granicz and Hickey [2003] and Saraiva and Schneider [2003]. Alternatively, the GPL itself might feature domain-specific optimization rules as a special kind of compiler directive. Such compiler extension makes the embedding process significantly more complex, however, and its cost-benefit ratio needs further scrutiny.

Estimation. Last but not least: In this article, our approach toward DSL development has been qualitative. Can the costs and benefits of DSLs be reliably quantified?

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for many useful comments. Arie van Deursen kindly gave us permission to use the source of the annotated DSL bibliography [van Deursen et al. 2000].

REFERENCES

ANLAUFF, M., KUTTER, P. W., AND PIERANTONIO, A. 1999. Tool support for language design and prototyping with Montages. In *Compiler Construction (CC'99)*, S. Jähnichen, Ed. Lecture Notes in Computer Science, vol. 1575. Springer-Verlag, 296–299.

ANTONIOTTI, M. AND GÖLLÜ, A. 1997. SHIFT and SMART-AHS: A language for hybrid system engineering modeling and simulation. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 171–182.

ATKINS, D., BALL, T., BRUNS, G., AND COX, K. 1999. Mawl: A domain-specific language for form-based services. *IEEE Trans. Softw. Eng. 25*, 3 (May/June), 334–346.

ATTALI, I., COURBIS, C., DEGENNE, P., FAU, A., PARIGOT, D., AND PASQUIER, C. 2001. SmartTools: A generator of interactive environments tools. In *Compiler Construction: 10th International Conference (CC'01)*, R. Wilhelm, Ed. Lecture Notes in Computer Science, vol. 2027. Springer-Verlag, 355–360.

AYCOCK, J. 2002. The design and implementation of SPARK, a toolkit for implementing domain-specific languages. *J. Comput. Inform. Tech. 10*, 1, 55–66.

BACKUS, J. W. 1960. The syntax and semantics of the proposed International Algebraic Language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing, UNESCO, Paris, 1959*. Oldenbourg, Munich and Butterworth, London, 125–132.

BADROS, G. 2000. JavaML: A markup language for Java source code. In *Proceedings of the 9th International World Wide Web Conference*. http://www9.org/w9cdrom/start.html.

BAGGE, O. S. AND HAVERAAEN, M. 2003. Domain-specific optimisation with user-defined rules in CodeBoost. In *Proceedings of the 4th International Workshop on Rule-Based Programming (RULE'03)*, J.-L. Giavitto and P.-E. Moreau, Eds. Electronic Notes in Theoretical Computer Science, vol. 86(2). Elsevier. http://www.sciencedirect.com/.

BARRON, D. W. 2000. *The World of Scripting Languages*. John Wiley.

BATORY, D., LOFASO, B., AND SMARAGDAKIS, Y. 1998. JTS: Tools for implementing domain-specific languages. In *Proceedings of the 5th International Conference on Software Reuse (JCSR'98)*, P. Devanbu and J. Poulin, Eds. IEEE Computer Society, 143–153.

BATORY, D., THOMAS, J., AND SIRKIN, M. 1994. Reengineering a complex application using a scalable data structure compiler. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. 111–120.

BAXTER, I. D., PIDGEON, C., AND MEHLICH, M. 2004. DMS: Program transformation for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. IEEE Computer Society, 625–634.

BENNETT, K. H. AND RAJLICH, V. T. 2000. Software maintenance and evolution: A roadmap. In *The Future of Software Engineering*, A. Finkelstein, Ed. ACM Press, 73–87.

BENTLEY, J. L. 1986. Programming pearls: Little languages. *Comm. ACM 29,* 8 (August), 711–721.

BERGIN, T. J. AND GIBSON, R. G., Eds. 1996. *History of Programming Languages II*. ACM Press.

BERTRAND, F. AND AUGERAUD, M. 1999. BDL: A specialized language for per-object reactive control. *IEEE Trans. Softw. Eng. 25*, 3, 347–362.

BIGGERSTAFF, T. J. 1998. A perspective of generative reuse. *Annals Softw. Eng. 5*, 169–226.

BIGGERSTAFF, T. J. AND PERLIS, A. J., Eds. 1989. *Software Reusability*. ACM Press/Addison-Wesley. Vol. I: Concepts and Models, Vol. II: Applications and Experience.

BONACHEA, D., FISHER, K., ROGERS, A., AND SMITH, F. 1999. Hancock: A language for processing very large-scale data. In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, 163–176.

BOSCH, J. AND DITTRICH, Y. Domain-specific languages for a changing world. http://www.cs.rug.nl/bosch/articles.html.

BRABAND, C. AND SCHWARTZBACH, M. 2002. Growing languages with metamorphic syntax macros. *ACM SIGPLAN Notices 37*, 3 (March), 31–40.

BRABAND, C., SCHWARTZBACH, M. I., AND VANGGAARD, M. 2003. The `metafront` system: Extensible parsing and transformation. In *Proceedings of the 3rd Workshop on Language Descriptions, Tools, and Applications (LDTA'03)*, B. R. Bryant and J. Saraiva, Eds. Electronic Notes in Theoretical Computer Science, vol. 82(3). Elsevier. http://www.sciencedirect.com/.

BRAVENBOER, M. AND VISSER, E. 2004. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, D. C. Schmidt, Ed. ACM, 365–383.

BROOKS, JR., F. P. 1996. Language design as design. In *History of Programming Languages II*. T. J. Bergin and R. C. Gibson Eds. ACM Press, 4–15.

BRUNTINK, M., VAN DEURSEN, A., AND TOURWÉ, T. 2005. Isolating idiomatic crosscutting concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*. IEEE Computer Society, 37–46.

BUFFENBARGER, J. AND GRUELL, K. 2001. A language for software subsystem composition. In *IEEE Proceedings of the 34th Hawaii International Conference on System Sciences*.

CARDELLI, L. AND DAVIES, R. 1999. Service combinators for web computing. *IEEE Trans. Softw. Eng. 25*, 3 (May/June), 309–316.

CHANDRA, S., RICHARDS, B., AND LARUS, J. R. 1999. Teapot: A domain-specific language for writing cache coherence protocols. *IEEE Trans. Softw. Eng. 25*, 3 (May/June), 317–333.

CHAPPELL, D. 1996. *Understanding ActiveX and OLE*. Microsoft Press.

CHIBA, S. 1995. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*. ACM, 285–299.

CLEAVELAND, J. C. 1988. Building application generators. *IEEE Softw. 5*, 4, 25–33.

CLEAVELAND, J. C. 2001. *Program Generators Using Java and XML*. Prentice-Hall.

CLEMENTS, J., FELLEISEN, M., FINDLER, R., FLATT, M., AND KRISHNAMURTHI, S. 2004. Fostering little languages. *Dr. Dobb's J. 29*, 3 (March), 16–24.

CONSEL, C. AND MARLET, R. 1998. Architecturing software using a methodology for language development. In *Principles of Declarative Programming (PLILP'98/ALP'98)*, C. Palamidessi, H. Glaser, and K. Meinke, Eds. Lecture Notes in Computer Science, vol. 1490. Springer-Verlag, 170–194.

COPLIEN, J., HOFFMAN, D., AND WEISS, D. 1998. Commonality and variability in software engineering. *IEEE Softw. 15*, 6, 37–45.

CORDY, J. R. 2004. TXL—A language for programming language tools and applications. In *Proceedings of the 4th Workshop on Language Descriptions, Tools, and Applications (LDTA'04)*, G. Hedin and E. van Wyk, Eds. Electronic Notes in Theoretical Computer Science, vol. 110. Elsevier, 3–31. http://www.sciencedirect.com/.

COURBIS, C. AND FINKELSTEIN, A. 2004. Towards an aspect weaving BPEL engine. In *Proceedings of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Y. Coady and D. H. Lorenz, Eds. Tech. rep. NU-CCIS-04-04, College of Computer and Information Science, Northeastern University, Boston, MA.

ČREPINŠEK, M., MERNIK, M., JAVED, F., BRYANT, B. R., AND SPRAGUE, A. 2005. Extracting grammar from programs: evolutionary approach. *ACM SIGPLAN Notices 40*, 4 (April), 39–46.

CRESPI-REGHIZZI, S., MELKANOFF, M. A., AND LICHTEN, L. 1973. The use of grammatical inference for designing programming languages. *Comm. ACM 16*, 83–90.

CREW, R. F. 1997. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 229–242.

CZARNECKI, K. AND EISENECKER, U. 2000. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley.

DE JONGE, M. 2002. Source tree composition. In *Software Reuse: Methods, Techniques, and Tools: 7th International Conference (ICSR-7)*, C. Gacek, Ed. Lecture Notes in Computer Science, vol. 2319. Springer-Verlag, 17–32.

DEAN, M., SCHREIBER, G., VAN HARMELEN, F., HENDLER, J., HORROCKS, I., MCGUINNESS, D. L., PATEL-SCHNEIDER, P. F., AND STEIN, L. A. 2003. OWL Web Ontology Language Reference. Working draft, W3C (March). http://www.w3.org/TR/2003/WD-owl-ref-20030331/.

DENNY, M. 2003. Ontology building: A survey of editing tools. Tech. rep., XML.com. http://www.xml.com/lpt/a/2002/11/06/ontologies.html.

ELLIOTT, C. 1999. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Trans. Softw. Eng. 25*, 3 (May/June), 291–308.

FAITH, R. E., NYLAND, L. S., AND PRINS, J. F. 1997. Khepera: A system for rapid implementation of domain specific languages. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 243–255.

FALBO, R. A., GUIZZARDI, G., AND DUARTE, K. C. 2002. An ontological approach to domain engineering. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*. ACM, 351–358.

FELLEISEN, M., FINDLER, R., FLATT, M., AND KRISHNAMURTHI, S. 2004. Building little languages with macros. *Dr. Dobb's J. 29*, 4 (April), 45–49.

FERTALJ, K., KALPIČ, D., AND MORNAR, V. 2002. Source code generator based on a proprietary specification language. In *Proceedings of the 35th Hawaii International Conference on System Sciences*.

FRAKES, W. 1998. Panel: Linking domain analysis with domain implementation. In *Proceedings of the 5th International Conference on Software Reuse*. IEEE Computer Society, 348–349.

FRAKES, W., PRIETO-DIAZ, R., AND FOX, C. 1998. DARE: Domain analysis and reuse environment. *Annals of Software Engineering 5*, 125–141.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GERMON, R. 2001. Using XML as an intermediate form for compiler development. In *XML Conference Proceedings*. http://www.idealliance.org/papers/xml2001/index.html.

GIL, J. AND TSOGLIN, Y. 2001. JAMOOS—A domain-specific language for language processing. *J. Comput. Inform. Tech. 9*, 4, 305–321.

GILMORE, S. AND RYAN, M., Eds. 2001. *Language Constructs for Describing Features—Proceedings of the FIREworks Workshop*. Springer-Verlag.

GLÄSSER, U., GUREVICH, Y., AND VEANES, M. 2002. An abstract communication model. Tech. rep. MSR-TR-2002-55. Microsoft Research, Redmond, WA.

GONDOW, K. AND KAWASHIMA, H. 2002. Towards ANSI C program slicing using XML. In *Proceedings of the 2nd Workshop on Language Descriptions, Tools, and Applications (LDTA'02)*, M. G. J. van den Brand and R. Lämmel, Eds. Electronic Notes in Theoretical Computer Science, vol. 65(3). Elsevier. http://www.sciencedirect.com/.

GOUGH, J. 2002. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall.

GRANICZ, A. AND HICKEY, J. 2003. Phobos: Extending compilers with executable language definitions. In *Proceedings of the 36th Hawaii International Conference on System Sciences*.

GRAY, J. AND KARSAI, G. 2003. An examination of DSLs for concisely representing model traversals and transformations. In *Proceedings of the 36th Hawaii International Conference on System Sciences*.

GRAY, R. W., LEVI, S. P., HEURING, V. P., SLOANE, A. M., AND WAITE, W. M. 1992. Eli: A complete, flexible compiler construction system. *Comm. ACM 35*, 2 (Feb.), 121–130.

GREENFIELD, J., SHORT, K., COOK, S., KENT, S., AND CRUPI, J. 2004. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley.

GUYER, S. Z. AND LIN, C. 1999. An annotation language for optimizing software libraries. In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, 39–52.

GUYER, S. Z. AND LIN, C. 2005. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. In *Proceedings of IEEE, 93*, 2, 342–357.

HEERING, J. AND KLINT, P. 2000. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices 35*, 3 (March) 39–48.

HERNDON, R. M. AND BERZINS, V. A. 1988. The realizable benefits of a language prototyping language. *IEEE Trans. Softw. Eng. 14*, 803–809.

HICSS 2001. *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS'34)*. IEEE.

HICSS 2002. *Proceedings of the 35th Hawaii International Conference on System Sciences (HICSS'35)*. IEEE.

HICSS 2003. *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'36)*. IEEE.

HUDAK, P. 1996. Building domain-specific embedded languages. *ACM Comput. Surv. 28*, 4 (Dec).

HUDAK, P. 1998. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse (JCSR'98)*, P. Devanbu and J. Poulin, Eds. IEEE Computer Society, 134–142.

JENNINGS, J. AND BEUSCHER, E. 1999. Verischemelog: Verilog embedded in Scheme. In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*. 123–134.

JONES, C. 1996. SPR Programming Languages Table Release 8.2, http://www.theadvisors.com/langcomparison.htm. (Accessed April 2005). Later release not available at publication.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

KADHIM, B. M. AND WAITE, W. M. 1996. Maptool—Supporting modular syntax development. In *Compiler Construction (CC'96)*, T. Gyimóthy, Ed. Lecture Notes in Computer Science, vol. 1060. Springer-Verlag, 268–280.

KAMIN, S., Ed. 1997. *DSL'97—1st ACM SIGPLAN Workshop on Domain-Specific Languages in Association with POPL'97*. University of Illinois Computer Science Report.

KAMIN, S. 1998. Research on domain-specific embedded languages and program generators. *Electro. Notes Theor. Comput. Sci. 14*. http://www.sciencedirect.com/.

KAMIN, S. AND HYATT, D. 1997. A special-purpose language for picture-drawing. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 297–310.

KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. S. 1990. Feature-oriented domain analysis (FODA) feasibility study. Tech. rep. CMU/SEI-90-TR-21. Software Engineering Institute, Carnegie Mellon University.

KASTENS, U. AND PFAHLER, P. 1998. Compositional design and implementation of domain-specific languages. In *IFIP TC2 WG 2.4 Working Conference on System Implementation 2000: Languages, Methods and Tools*, R. N. Horspool, Ed. Chapman and Hall, 152–165.

KICZALES, G., DES RIVIERES, J., AND BOBROW, D. G. 1991. *The Art of the Metaobject Protocol*. MIT Press.

KIEBURTZ, R. B., MCKINNEY, L., BELL, J. M., HOOK, J., KOTOV, A., LEWIS, J., OLIVA, D. P., SHEARD, T., SMITH, I., AND WALTON, L. 1996. A software engineering experiment in software component generation. In *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)*. IEEE, 542–552.

KIENLE, H. M. AND MOORE, D. L. 2002. `smgn`: Rapid prototyping of small domain-specific languages. *J. Comput. Inform. Tech. 10*, 1, 37–53.

KLARLUND, N. AND SCHWARTZBACH, M. 1999. A domain-specific language for regular sets of strings and trees. *IEEE Trans. Softw. Eng. 25*, 3 (May/June), 378–386.

KRUEGER, C. W. 1992. Software reuse. *ACM Computing Surveys 24*, 2 (June), 131–183.

KUCK, D. J. 2005. Platform 2015 software: Enabling innovation in parallelism for the next decade. *Technology@Intel Magazine*. http://www.intel.com/technology/magazine/computing/Parallelism-0405.htm .

KUMAR, S., MANDELBAUM, Y., YU, X., AND LI, K. 2001. ESP: A language for programmable devices. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, 309–320.

KUTTER, P. W., SCHWEIZER, D., AND THIELE, L. 1998. Integrating domain specific language design in the software life cycle. In *Applied Formal Methods—FM-Trends 98*, D. Hutter et al., Eds. Lecture Notes in Computer Science, vol. 1641. Springer-Verlag, 196–212.

LAUNCHBURY, J., LEWIS, J. R., AND COOK, B. 1999. On embedding a microarchitectural design language within Haskell. *ACM SIGPLAN Notices 34*, 9 (Sept.), 60–69.

LENGAUER, C., BATORY, D., CONSEL, C., AND ODERSKY, M., Eds. 2004. *Domain-Specific Program Generation*. Lecture Notes in Computer Science, vol. 3016. Springer-Verlag.

LEVY, M. R. 1998. Web programming in Guide. *Softw. Pract. Exper. 28*, 1581–1603.

MARTIN, J. 1985. *Fourth-Generation Languages*. Vol. I: Principles, Vol II: Representative 4GLs. Prentice-Hall.

MAUW, S., WIERSMA, W., AND WILLEMSE, T. 2004. Language-driven system design. *Int. J. Softw. Eng. Knowl. Eng. 14*, 1–39.

MERNIK, M. AND LÄMMEL, R. 2001. Special issue on domain-specific languages, Part I. *J. Comput. Inform. Techn. 9*, 4.

MERNIK, M. AND LÄMMEL, R. 2002. Special issue on domain-specific languages, Part II. *J. Comput. Inform. Techn. 10*, 1.

MERNIK, M., LENIČ, M., AVDIČAUŠEVIĆ, E., AND ŽUMER, V. 2000. Multiple attribute grammar inheritance. *Informatica 24*, 3 (Sept.), 319–328.

MERNIK, M., NOVAK, U., AVDIČAUŠEVIĆ, E., LENIČ, M., AND ŽUMER, V. 2001. Design and implementation of simple object description language. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC'01)*. ACM, 590–594.

MERNIK, M., ŽUMER, V., LENIČ, M., AND AVDIČAUŠEVIĆ, E. 1999. Implementation of multiple attribute grammar inheritance in the tool LISA. *ACM SIGPLAN Notices 34*, 6 (June), 68–75.

MOURA, J. M. F., PÜSCHEL, M., PADUA, D., AND DONGARRA, J. 2005. Special issue on program generation, optimization, and platform adaptation. *Proceedings of the IEEE 93*, 2.

NAKATANI, L. AND JONES, M. 1997. Jargons and infocentrism. *1st Acm SIGPLAN Workshop on Domain-Specific Languages*. 59–74. http://www-sal.cs.uiuc.edu/ kamin/dsl/papers/nakatani.ps.

NARDI, B. A. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press.

NEIGHBORS, J. M. 1984. The Draco approach to constructing software from reusable components. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept.), 564–574.

PARIGOT, D. 2004. Towards domain-driven development: The SmartTools software factory. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. ACM, 37–38.

PEYTON JONES, S., TOLMACH, A., AND HOARE, T. 2001. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proceedings of the Haskell Workshop*.

PFAHLER, P. AND KASTENS, U. 2001. Configuring component-based specifications for domain-specific languages. In *Proceedings of the 34th Hawaii International Conference on System Sciences*.

RAYMOND, E. S. 2001. The CML2 language: Python implementation of a constraint-based interactive configurator. In *Proceeding of the 9th International Python Conference*. 135–142. http://www.catb.org/ esr/cml2/cml2-paper.html.

RISI, W., MARTINEZ-LOPEZ, P., AND MARCOS, D. 2001. Hycom: A domain specific language for hypermedia application development. In *Proceedings of the 34th Hawaii International Conference on System Sciences*.

ROSS, D. T. 1981. Origins of the APT language for automatically programmed tools. *History of Programming Languages*, R. L. Wexelblat Ed. Academic Press. 279–338.

SALUS, P. H., Ed. 1998. *Little Languages*. Handbook of Programming Languages, vol. III. MacMillan.

SAMMET, J. E. 1969. *Programming Languages: History and Fundamentals*. Prentice-Hall.

SARAIVA, J. AND SCHNEIDER, S. 2003. Embedding domain specific languages in the attribute grammar formalism. In *Proceedings of the 36th Hawaii International Conference on System Sciences*.

SCHNARR, E., HILL, M. D., AND LARUS, J. R. 2001. Facile: A language and compiler for high-performance processor simulators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, 321–331.

SCHNEIDER, K. A. AND CORDY, J. R. 2002. AUI: A programming language for developing plastic interactive software. In *Proceedings of the 35th Hawaii International Conference on System Sciences*.

SCHUPP, S., GREGOR, D. P., MUSSER, D. R., AND LIU, S. 2001. User-extensible simplification—Type-based optimizer generators. In *Compiler Construction (CC'01)*, R. Wilhelm, Ed. Lecture

Notes in Computer Science, vol. 2027. Springer-Verlag, 86–101.

SDL FORUM. 2000. MSC-2000: Interaction for the new millenium. http://www.sdl-forum.org/MSC2000present/index.htm.

SIMOS, M. AND ANTHONY, J. 1998. Weaving the model web: A multi-modeling approach to concepts and features in domain engineering. In *Proceedings of the 5th International Conference on Software Reuse*. IEEE Computer Society, 94–102.

SIRER, E. G. AND BERSHAD, B. N. 1999. Using production grammars in software testing. In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*. 1–14.

SLOANE, A. M. 2002. Post-design domain-specific language embedding: A case study in the software engineering domain. In *Proceedings of the 35th Hawaii International Conference on System Sciences*.

SLONNEGER, K. AND KURTZ, B. L. 1995. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley.

SMARAGDAKIS, Y. AND BATORY, D. 1997. DiSTiL: A transformation library for data structures. In *Proceedings of the USENIX Conference on Domain-Specific Languages*. 257–270.

SMARAGDAKIS, Y. AND BATORY, D. 2000. Application generators. In *Wiley Encyclopedia of Electrical and Electronics Engineering Online*, J. Webster, Ed. John Wiley.

SOROKER, D., KARASICK, M., BARTON, J., AND STREETER, D. 1997. Extension mechanisms in Montana. In *Proceedings of the 8th Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE'97)*. IEEE Computer Society, 119–128.

SPINELLIS, D. 2001. Notable design patterns for domain-specific languages. *J. Syst. Softw. 56*, 91–99.

SUTCLIFFE, A. AND MEHANDJIEV, N. 2004. Special issue on End-User Development. *Comm. ACM 47*, 9.

SZYPERSKI, C. 2002. *Component Software—Beyond Object-Oriented Programming*, 2nd Ed. Addison-Wesley/ACM Press.

TAYLOR, R. N., TRACZ, W., AND COGLIANESE, L. 1995. Software development using domain-specific software architectures. *ACM SIGSOFT Software Engineering Notes 20*, 5, 27–37.

TENNENT, R. D. 1977. Language design methods based on semantic principles. *Acta Inf. 8*, 97–112.

THATTE, S. 2001. XLANG: Web services for business process design. Tech. rep. Microsoft. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/.

THIBAULT, S. A. 1998. Domain-specific languages: Conception, implementation and application. Ph.D. thesis, University of Rennes.

THIBAULT, S. A., CONSEL, C., AND MULLER, G. 1998. Safe and efficient active network programming. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*. IEEE Computer Society, 135–143.

THIBAULT, S. A., MARLET, R., AND CONSEL, C. 1999. Domain-specific languages: From design to implementation—Application to video device drivers generation. *IEEE Trans. Softw. Eng. 25*, 3, (May/June), 363–377.

TRACZ, W. AND COGLIANESE, L. 1995. DOMAIN (DOmain Model All INtegrated)—a DSSA domain analysis tool. Tech. rep. ADAGE-LOR-94-11. Loral Federal Systems.

UPnP 2003. Universal Plug and Play Forum. http://www.upnp.org/.

USENIX 1997. *Proceedings of the USENIX Conference on Domain-Specific Languages*.

USENIX 1999. *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages (DSL'99)*.

VAN DEN BRAND, M. G. J., VAN DEURSEN, A., HEERING, J., DE JONG, H. A., DE JONGE, M., KUIPERS, T., KLINT, P., MOONEN, L., OLIVER, P. A., SCHEERDER, J., VINJU, J. J., VISSER, E., AND VISSER, J. 2001. The ASF+SDF Meta-Environment: A component-based language development environment. In *Compiler Construction (CC'01)*, R. Wilhelm, Ed. Lecture Notes in Computer Science, vol. 2027. Springer-Verlag, 365–370. http://www.cwi.nl/projects/MetaEnv.

VAN DEN BRAND, M. G. J. AND VISSER, E. 1996. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. Method. 5*, 1–41.

VAN DEURSEN, A. AND KLINT, P. 1998. Little languages: Little maintenance? *J. Softw. Maintenance 10*, 75–92.

VAN DEURSEN, A. AND KLINT, P. 2002. Domain-specific language design requires feature descriptions. *J. Comput. Inform. Tech. 10*, 1, 1–17.

VAN DEURSEN, A., KLINT, P., AND VISSER, J. 2000. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices 35*, 6 (June), 26–36.

VAN ENGELEN, R. 2001. ATMOL: A domain-specific language for atmospheric modeling. *J. Comput. Inform. Techn. 9*, 4, 289–303.

VELDHUIZEN, T. L. 1995a. Expression templates. *C++ Report 7*, 5 (June) 26–31.

VELDHUIZEN, T. L. 1995b. Using C++ template metaprograms. *C++ Report 7*, 4 (May) 36–43.

VELDHUIZEN, T. L. 2001. Blitz++ User's Guide. Version 1.2 http://www .oonumerics.org/blitz/ manual/blitz.ps.

VISSER, E. 2003. Stratego—Strategies for program transformation. http://www.stratego-language.org.

WANG, D. C., APPEL, A. W., KORN, J. L., AND SERRA, C. S. 1997. The Zephyr abstract syntax description language. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 213–28.

WEISS, D. AND LAY, C. T. R. 1999. *Software Product Line Engineering*. Addison-Wesley.

WEXELBLAT, R. L., Ed. 1981. *History of Programming Languages*. Academic Press.

WILE, D. S. 1993. *POPART: Producer of Parsers and Related Tools*. USC/Information Sciences Institute. http:// mr.teknowledge.com /wile/popart.html.

WILE, D. S. 2001. Supporting the DSL spectrum. *J. Comput. Inform. Techn. 9*, 4, 263–287.

WILE, D. S. 2004. Lessons learned from real DSL experiments. *Sci. Comput. Program. 51*, 265–290.

WILE, D. S. AND RAMMING, J. C. 1999. Special issue on Domain-Specific Languages. *IEEE Trans. Softw. Eng. SE-25*, 3 (May/June).

XIONG, J., JOHNSON, J., JOHNSON, R. W., AND PADUA, D. A. 2001. SPL: A language and compiler for DSP algorithms. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, 298–308.

Received September 2003; revised May 2005; accepted December 2005

Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, 2003, pp. 726-736.

Writing Good Software Engineering Research Papers

Minitutorial

Mary Shaw
Carnegie Mellon University
mary.shaw@cs.cmu.edu

Abstract

Software engineering researchers solve problems of several different kinds. To do so, they produce several different kinds of results, and they should develop appropriate evidence to validate these results. They often report their research in conference papers. I analyzed the abstracts of research papers submitted to ICSE 2002 in order to identify the types of research reported in the submitted and accepted papers, and I observed the program committee discussions about which papers to accept. This report presents the research paradigms of the papers, common concerns of the program committee, and statistics on success rates. This information should help researchers design better research projects and write papers that present their results to best advantage.

Keywords: research design, research paradigms, validation, software profession, technical writing

1. Introduction

In software engineering, research papers are customary vehicles for reporting results to the research community. In a research paper, the author explains to an interested reader what he or she accomplished, and how the author accomplished it, and why the reader should care. A good research paper should answer a number of questions:

- What, precisely, was your contribution?
  - What question did you answer?
  - Why should the reader care?
  - What larger question does this address?
- What is your new result?
  - What new knowledge have you contributed that the reader can use elsewhere?
  - What previous work (yours or someone else's) do you build on? What do you provide a superior alternative to?
  - How is your result different from and better than this prior work?
  - What, precisely and in detail, is your new result?
- Why should the reader believe your result?
  - What standard should be used to evaluate your claim?
  - What concrete evidence shows that your result satisfies your claim?

If you answer these questions clearly, you'll probably communicate your result well. If in addition your result represents an interesting, sound, and significant contribution to our knowledge of software engineering, you'll have a good chance of getting it accepted for publication in a conference or journal.

Other fields of science and engineering have well-established research paradigms. For example, the experimental model of physics and the double-blind studies of medicines are understood, at least in broad outline, not only by the research community but also by the public at large. In addition to providing guidance for the design of research in a discipline, these paradigms establish the scope of scientific disciplines through a social and political process of "boundary setting" [5].

Software engineering, however, has not yet developed this sort of well-understood guidance. I previously [19, 20] discussed early steps toward such understanding, including a model of the way software engineering techniques mature [17, 18] and critiques of the lack of rigor in experimental software engineering [1, 22, 23, 24, 25]. Those discussions critique software engineering research reports against the standards of classical paradigms. The discussion here differs from those in that this discussion reports on the types of papers that are accepted in practices as good research reports. Another current activity, the Impact Project [7] seeks to trace the influence of software engineering research on practice. The discussion here focuses on the paradigms rather than the content of the research

This report examines how software engineers answer the questions above, with emphasis on the design of the research project and the organization of the report. Other sources (e.g., [4]) deal with specific issues of technical writing. Very concretely, the examples here come from the papers submitted to ICSE 2002 and the program committee review of those papers. These examples report research results in software engineering. Conferences often include other kinds of papers, including experience reports, materials on software engineering education, and opinion essays.

2. What, precisely, was your contribution?

Before reporting what you did, explain what problem you set out to solve or what question you set out to answer —and why this is important.

2.1 What kinds of questions do software engineers investigate?

Generally speaking, software engineering researchers seek better ways to develop and evaluate software. Development includes all the synthetic activities that involve creating and modifying the software, including the code, design documents, documentation, etc. Evaluation includes all the analytic activities associated with predicting, determining, and estimating properties of the software systems, including both functionality and extra-functional properties such as performance or reliability.

Software engineering research answers questions about methods of development or analysis, about details of designing or evaluating a particular instance, about generalizations over whole classes of systems or techniques, or about exploratory issues concerning existence or feasibility. Table 1 lists the types of research questions that are asked by software engineering research papers and provides specific question templates.

Table 1. Types of software engineering research questions

Type of question	Examples
Method or means of development	How can we do/create/modify/evolve (or automate doing) X? What is a better way to do/create/modify/evolve X?
Method for analysis or evaluation	How can I evaluate the quality/correctness of X? How do I choose between X and Y?
Design, evaluation, or analysis of a particular instance	How good is Y? What is property X of artifact/method Y? What is a (better) design, implementation, maintenance, or adaptation for application X? How does X compare to Y? What is the current state of X / practice of Y?
Generalization or characterization	Given X, what will Y (necessarily) be? What, exactly, do we mean by X? What are its important characteristics? What is a good formal/empirical model for X? What are the varieties of X, how are they related?
Feasibility study or exploration	Does X even exist, and if so what is it like? Is it possible to accomplish X at all?

The first two types of research produce methods of development or of analysis that the authors investigated in one setting, but that can presumably be applied in other settings. The third type of research deals explicitly with some particular system, practice, design or other instance of a system or method; these may range from narratives about industrial practice to analytic comparisons of alternative designs. For this type of research the instance itself should have some broad appeal—an evaluation of Java is more likely to be accepted than a simple evaluation of the toy language you developed last summer. Generalizations or characterizations explicitly rise above the examples presented in the paper. Finally, papers that deal with an issue in a completely new way are sometimes treated differently from papers that improve on prior art, so "feasibility" is a separate category (though no such papers were submitted to ICSE 2002).

Newman's critical comparison of HCI and traditional engineering papers [12] found that the engineering papers were mostly incremental (improved model, improved technique), whereas many of the HCI papers broke new ground (observations preliminary to a model, brand new technique). One reasonable interpretation is that the traditional engineering disciplines are much more mature than HCI, and so the character of the research might reasonably differ [17, 18]. Also, it appears that different disciplines have different expectations about the "size" of a research result—the extent to which it builds on existing knowledge or opens new questions. In the case of ICSE, the kinds of questions that are of interest and the minimum interesting increment may differ from one area to another.

2.2 Which of these are most common?

The most common kind of ICSE paper reports an improved method or means of developing software—that is, of designing, implementing, evolving, maintaining, or otherwise operating on the software system itself. Papers addressing these questions dominate both the submitted and the accepted papers. Also fairly common are papers about methods for reasoning about software systems, principally analysis of correctness (testing and verification). Analysis papers have a modest acceptance edge in this very selective conference.

Table 2 gives the distribution of submissions to ICSE 2002, based on reading the abstracts (not the full papers—but remember that the abstract tells a reader what to expect from the paper). For each type of research question, the table gives the number of papers submitted and accepted, the percentage of the total paper set of each kind, and the acceptance ratio within each type of question. Figures 1 and 2 show these counts and distributions.

Table 2. Types of research questions represented in ICSE 2002 submissions and acceptances

Type of question	Submitted	Accepted	Ratio Acc/Sub
Method or means of development	142(48%)	18 (42%)	(13%)
Method for analysis or evaluation	95 (32%)	19 (44%)	(20%)
Design, evaluation, or analysis of a particular instance	43 (14%)	5 (12%)	(12%)
Generalization or characterization	18 (6%)	1 (2%)	(6%)
Feasibility study or exploration	0 (0%)	0 (0 %)	(0%)
TOTAL	298(100.0%)	43 (100.0%)	(14%)

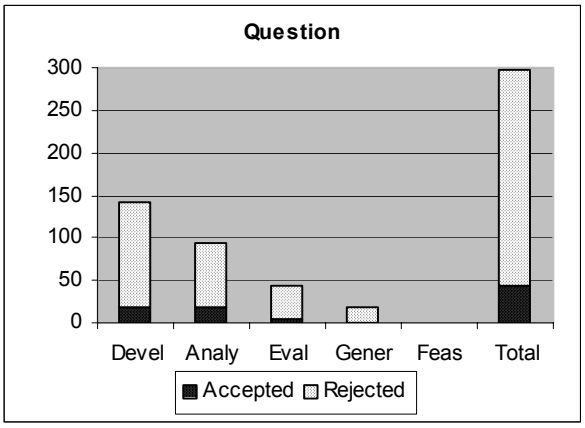


Figure 1. Counts of acceptances and rejections by type of research question

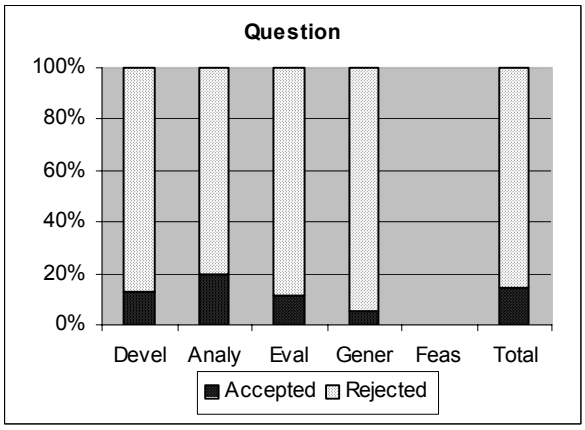


Figure 2. Distribution of acceptances and rejections by type of research question

2.3 What do program committees look for?

Acting on behalf of prospective readers, the program committee looks for a clear statement of the specific problem you solved—the question about software development you answered—and an explanation of how the answer will help solve an important software engineering problem. You'll devote most of your paper to describing your result, but you should begin by explaining what question you're answering and why the answer matters.

If the program committee has trouble figuring out whether you developed a new evaluation technique and demonstrated it on an example, or applied a technique you reported last year to a new real-world example, or evaluated the use of a well-established evaluation technique, you have not been clear.

3. What is your new result?

Explain precisely what you have contributed to the store of software engineering knowledge and how this is useful beyond your own project.

3.1 What kinds of results do software engineers produce?

The tangible contributions of software engineering research may be procedures or techniques for development or analysis; they may be models that generalize from specific examples, or they may be specific tools, solutions, or results about particular systems. Table 3 lists the types of research results that are reported in software engineering research papers and provides specific examples.

3.2 Which of these are most common?

By far the most common kind of ICSE paper reports a new procedure or technique for development or analysis. Models of various degrees of precision and formality were also common, with better success rates for quantitative than for qualitative models. Tools and notations were well represented, usually as auxiliary results in combination with a procedure or technique. Table 4 gives the distribution of submissions to ICSE 2002, based on reading the abstracts (but not the papers), followed by graphs of the counts and distributions in Figures 3 and 4.

Table 3. Types of software engineering research results	
Type of result	**Examples**
Procedure or technique	New or better way to do some task, such as design, implementation, maintenance, measurement, evaluation, selection from alternatives; includes techniques for implementation, representation, management, and analysis; a technique should be operational—not advice or guidelines, but a procedure
Qualitative or descriptive model	Structure or taxonomy for a problem area; architectural style, framework, or design pattern; non-formal domain analysis, well-grounded checklists, well-argued informal generalizations, guidance for integrating other results, well-organized interesting observations
Empirical model	Empirical predictive model based on observed data
Analytic model	Structural model that permits formal analysis or automatic manipulation
Tool or notation	Implemented tool that embodies a technique; formal language to support a technique or model (should have a calculus, semantics, or other basis for computing or doing inference)
Specific solution, prototype, answer, or judgment	Solution to application problem that shows application of SE principles – may be design, prototype, or full implementation; careful analysis of a system or its development, result of a specific analysis, evaluation, or comparison
Report	Interesting observations, rules of thumb, but not sufficiently general or systematic to rise to the level of a descriptive model.

Table 4. Types of research results represented in ICSE 2002 submissions and acceptances			
Type of result	**Submitted**	**Accepted**	**Ratio Acc/Sub**
Procedure or technique	152(44%)	28 (51%)	18%
Qualitative or descriptive model	50 (14%)	4 (7%)	8%
Empirical model	4 (1%)	1 (2%)	25%
Analytic model	48 (14%)	7 (13%)	15%
Tool or notation	49 (14%)	10 (18%)	20%
Specific solution, prototype, answer, or judgment	34 (10%)	5 (9%)	15%
Report	11 (3%)	0 (0%)	0%
TOTAL	348(100.0%)	55 (100.0%)	16%

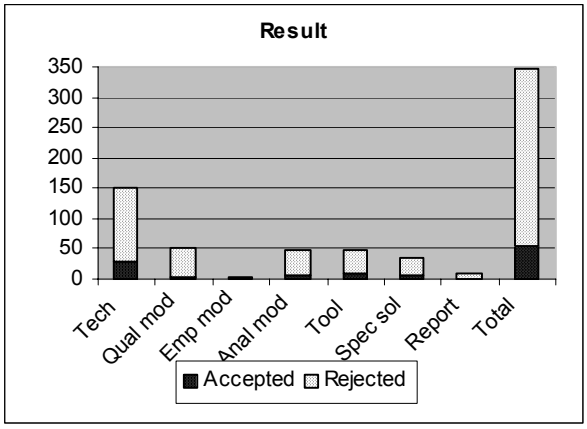


Figure 3. Counts of acceptances and rejections by type of result

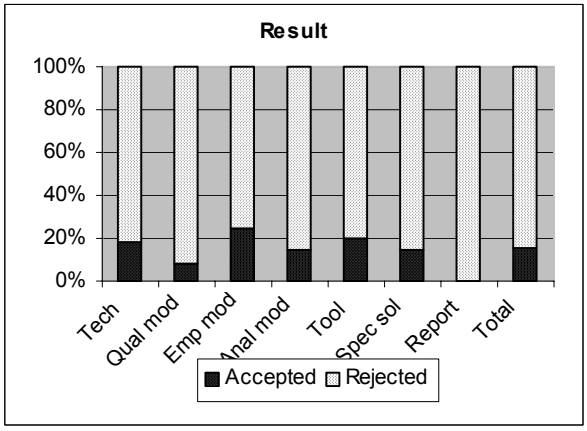


Figure 4. Distribution of acceptances and rejections by type of result

The number of results is larger than the number of papers because 50 papers included a supporting result, usually a tool or a qualitative model.

Research projects commonly produce results of several kinds. However, conferences, including ICSE, usually impose strict page limits. In most cases, this provides too little space to allow full development of more than one idea, perhaps with one or two supporting ideas. Many authors present the individual ideas in conference papers, and then synthesize them in a journal article that allows space to develop more complex relations among results.

3.3 What do program committees look for?

The program committee looks for interesting, novel, exciting results that significantly enhance our ability to develop and maintain software, to know the quality of the software we develop, to recognize general principles about software, or to analyze properties of software.

You should explain your result in such a way that someone else could use your ideas. Be sure to explain what's novel or original – is it the idea, the application of the idea, the implementation, the analysis, or what?

Define critical terms precisely. Use them consistently. The more formal or analytic the paper, the more important this is.

Here are some questions that the program committee may ask about your paper:

What, precisely, do you claim to contribute?

Does your result fully satisfy your claims? Are the definitions precise, and are terms used consistently?

Authors tend to have trouble in some specific situations. Here are some examples, with advice for staying out of trouble:

- *If your result ought to work on large systems*, explain why you believe it scales.
- *If you claim your method is "automatic",* using it should not require human intervention. If it's automatic when it's operating but requires manual assistance to configure, say so. If it's automatic except for certain cases, say so, and say how often the exceptions occur.
- *If you claim your result is "distributed",* it probably should not have a single central controller or server. If it does, explain what part of it is distributed and what part is not.
- *If you're proposing a new notation for an old problem,* explain why your notation is clearly superior to the old one.
- *If your paper is an "experience report",* relating the use of a previously-reported tool or technique in a practical software project, be sure that you explain what idea the reader can take away from the paper to use in other settings. If that idea is increased confidence in the tool or technique, show how your experience should increase the reader's confidence for applications beyond the example of the paper.

What's new here?

The program committee wants to know what is novel or exciting, and why. What, specifically, is the contribution? What is the increment over earlier work by the same authors? by other authors? Is this a sufficient increment, given the usual standards of subdiscipline?

Above all, the program committee also wants to know what you actually contributed to our store of knowledge about software engineering. Sure, you wrote this tool and tried it out. But was your contribution the technique that is embedded in the tool, or was it making a tool that's more effective than other tools that implement the technique, or was it showing that the tool you described in a previous paper actually worked on a practical large-scale problem? It's better for you as the author to explain than for the program committee to guess. Be clear about your claim …

Awful	▼	• I completely and generally solved … (unless you actually did!)
Bad	▼	• I worked on galumphing. (or studied, investigated, sought, explored)
Poor	▼	• I worked on improving galumphing. (or contributed to, participated in, helped with)
Good	▲	• I showed the feasibility of composing blitzing with flitzing. • I significantly improved the accuracy of the standard detector. (or proved, demonstrated, created, established, found, developed)
Better	▲	• I automated the production of flitz tables from specifications. • With a novel application of the blivet transform, I achieved a 10% increase in speed and a 15% improvement in coverage over the standard method.

Use verbs that show results and achievement, not just effort and activity.

"***Try*** not. ***Do,*** or ***do*** not. There is no ***try***." -- Yoda .

What has been done before? How is your work different or better?

What existing technology does your research build on? What existing technology or prior research does your research provide a superior alternative to? What's new here compared to your own previous work? What alternatives have other researchers pursued, and how is your work different or better?

As in other areas of science and engineering, software engineering knowledge grows incrementally. Program committees are very interested in your interpretation of prior work in the area. They want to know how your work is related to the prior work, either by building on it or by providing an alternative. If you don't explain this, it's hard for the program committee to understand how you've added to our store of knowledge. You may also damage your credibility if the program committee can't tell whether you know about related work.

Explain the relation to other work clearly …

Awful	▼	The galumphing problem has attracted much attention [3,8,10,18,26,32,37]
Bad	▼	Smith [36] and Jones [27] worked on galumphing.
Poor	▼	Smith [36] addressed galumphing by blitzing, whereas Jones [27] took a flitzing approach.
Good	▲	Smith's blitzing approach to galumphing [36] achieved 60% coverage [39]. Jones [27] achieved 80% by flitzing, but only for pointer-free cases [16].
Better	▲	Smith's blitzing approach to galumphing [36] achieved 60% coverage [39]. Jones [27] achieved 80% by flitzing, but only for pointer-free cases [16]. We modified the blitzing approach to use the kernel representation of flitzing and achieved 90% coverage while relaxing the restriction so that only cyclic data structures are prohibited.

What, precisely, is the result?

Explain what your result is and how it works. Be concrete and specific. Use examples.

If you introduce a new model, be clear about its power. How general is it? Is it based on empirical data, on a formal semantics, on mathematical principles? How formal is it—a qualitative model that provides design guidance may be as valuable as a mathematical model of some aspect of correctness, but they will have to satisfy different standards of proof. Will the model scale up to problems of size appropriate to its domain?

If you introduce a new metric, define it precisely. Does it measure what it purports to measure and do so better than the alternatives? Why?

If you introduce a new architectural style, design pattern, or similar design element, treat it as if it were a new generalization or model. How does it differ from the alternatives? In what way is it better? What real problem does it solve? Does it scale?

If your contribution is principally the synthesis or integration of other results or components, be clear about why the synthesis is itself a contribution. What is novel, exciting, or nonobvious about the integration? Did you generalize prior results? Did you find a better representation? Did your research improve the individual results or components as well as integrating them? A paper that simply reports on using numerous elements together is not enough, even if it's well-engineered. There must be an idea or lesson or model that the reader can take from the paper and apply to some other situation.

If your paper is chiefly a report on experience applying research results to a practical problem, say what the reader can learn from the experience. Are your conclusions strong and well-supported? Do you show comparative data and/or statistics? An anecdotal report on a single project is usually not enough. Also, if your report mixes additional innovation with validation through experience, avoid confusing your discussion of the innovation with your report on experience. After all, if you changed the result before you applied it, you're evaluating the changed result. And if you changed the result while you were applying it, you may have confounded the experiences with the two versions.

If a tool plays a featured role in your paper, what is the role of the tool? Does it simply support the main contribution, or is the tool itself a principal contribution, or is some aspect of the tool's use or implementation the main point? Can a reader apply the idea without the tool? If the tool is a central part of result, what is the technical innovation embedded in the tool or its implementation?

If a system implementation plays a featured role in your paper, what is the role of the implementation? Is the system sound? Does it do what you claim it does? What ideas does the system demonstrate?

- *If the implementation illustrates an architecture or design strategy*, what does it reveal about the architecture? What was the design rationale? What were the design tradeoffs? What can the reader apply to a different implementation?
- *If the implementation demonstrates an implementation technique*, how does it help the reader use the technique in another setting?
- *If the implementation demonstrates a capability or performance improvement*, what concrete evidence does it offer to support the claim?
- *If the system is itself the result*, in what way is it a contribution to knowledge? Does it, for example, show you can do something that no one has done before (especially if people doubted that this could be done)?

4. Why should the reader believe your result?

Show evidence that your result is valid—that it actually helps to solve the problem you set out to solve.

4.1. What kinds of validation do software engineers do?

Software engineers offer several kinds of evidence in support of their research results. It is essential to select a form of validation that is appropriate for the type of research result and the method used to obtain the result. As an obvious example, a formal model should be supported by rigorous derivation and proof, not by one or two simple examples. On the other hand, a simple example derived from a practical system may play a major role in validating a new type of development method. Table 5 lists the types of research validation that are used in software engineering research papers and provides specific examples. In this table, the examples are keyed to the type of result they apply to.

Table 5. Types of software engineering research validation

Type of validation	Examples
Analysis	I have analyzed my result and find it satisfactory through rigorous analysis, e.g. … For a formal model … rigorous derivation and proof For an empirical model … data on use in controlled situation For a controlled experiment … carefully designed experiment with statistically significant results
Evaluation	Given the stated criteria, my result... For a descriptive model … adequately describes phenomena of interest … For a qualitative model … accounts for the phenomena of interest… For an empirical model … is able to predict … because …, or … generates results that fit actual data … Includes feasibility studies, pilot projects
Experience	My result has been used on real examples by someone other than me, and the evidence of its correctness/usefulness/effectiveness is … For a qualitative model … narrative For an empirical model or tool … data, usually statistical, on practice For a notation or technique … comparison of systems in actual use
Example	Here's an example of how it works on For a technique or procedure …a "slice of life" example based on a real system … For a technique or procedure …a system that I have been developing … For a technique or procedure … a toy example, perhaps motivated by reality The "slice of life" example is most likely to be convincing, especially if accompanied by an explanation of why the simplified example retains the essence of the problem being solved. Toy or textbook examples often fail to provide persuasive validation, (except for standard examples used as model problems by the field).
Persuasion	I thought hard about this, and I believe passionately that ... For a technique … if you do it the following way, then … For a system … a system constructed like this would … For a model … this example shows how my idea works Validation purely by persuasion is rarely sufficient for a research paper. Note, though, that if the original question was about feasibility, a working system, even without analysis, can suffice
Blatant assertion	No serious attempt to evaluate result. This is highly unlikely to be acceptable

4.2 Which of these are most common?

Alas, well over a quarter of the ICSE 2002 abstracts give no indication of how the paper's results are validated, if at all. Even when the abstract mentions that the result was applied to an example, it was not always clear whether the example was a textbook example, or a report on use in the field, or something in between.

The most successful kinds of validation were based on analysis and real-world experience. Well-chosen examples were also successful. Persuasion was not persuasive, and narrative evaluation was only slightly more successful. Table 6 gives the distribution of submissions to ICSE 2002, based on reading the abstracts (but not the papers), followed by graphs of the counts and distributions. Figures 5 and 6 show these counts and distributions.

Table 6. Types of research validation represented in ICSE 2002 submissions and acceptances			
Type of validation	**Submitted**	**Accepted**	**Ratio Acc/Sub**
Analysis	48 (16%)	11 (26%)	23%
Evaluation	21 (7%)	1 (2%)	5%
Experience	34 (11%)	8 (19%)	24%
Example	82 (27%)	16 (37%)	20%
Some example, can't tell whether it's toy or actual use	6 (2%)	1 (2%)	17%
Persuasion	25 (8%)	0 (0.0%)	0%
No mention of validation in abstract	84 (28%)	6 (14%)	7%
TOTAL	300(100.0%)	43 (100.0%)	14%

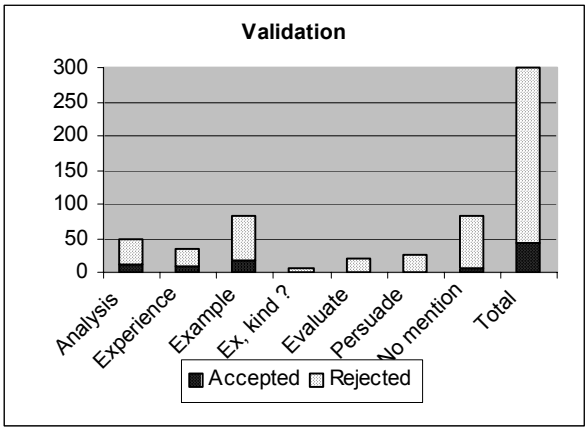


Figure 5. Counts of acceptances and rejections by type of validation

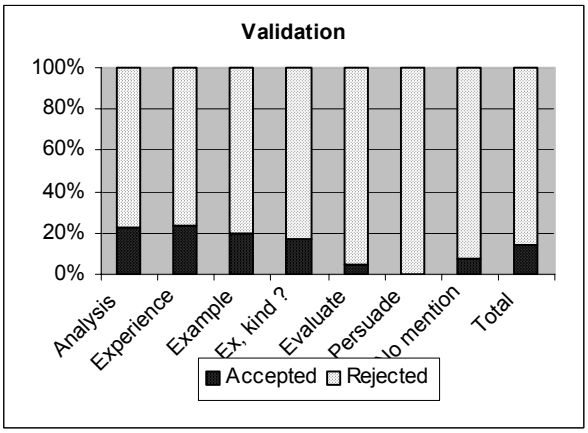


Figure 6. Distribution of acceptances and rejections by type of validation

4.3 What do program committees look for?

The program committee looks for solid evidence to support your result. It's not enough that your idea works for you, there must also be evidence that the idea or the technique will help someone else as well.

The statistics above show that analysis, actual experience in the field, and good use of realistic examples tend to be the most effective ways of showing why your result should be believed. Careful narrative, qualitative analysis can also work if the reasoning is sound.

Why should the reader believe your result?

Is the paper argued persuasively? What evidence is presented to support the claim? What kind of evidence is offered? Does it meet the usual standard of the subdiscipline?

Is the kind of evaluation you're doing described clearly and accurately? "Controlled experiment" requires more than data collection, and "case study" requires more than anecdotal discussion. Pilot studies that lay the groundwork for controlled experiments are often not publishable by themselves.

Is the validation related to the claim? If you're claiming performance improvement, validation should analyze performance, not ease of use or generality. And conversely.

Is this such an interesting, potentially powerful idea that it should get exposure despite a shortage of concrete evidence?

Authors tend to have trouble in some specific situations. Here are some examples, with advice for staying out of trouble:

- *If you claim to improve on prior art,* compare your result objectively to the prior art.
- *If you used an analysis technique,* follow the rules of that analysis technique. If the technique is not a common one in software engineering (e.g., meta-analysis, decision theory, user studies or other behavioral analyses), explain the technique and standards of proof, and be clear about your adherence to the technique.
- *If you offer practical experience as evidence for your result,* establish the effect your research has. If at all possible, compare similar situations with and without your result.

- *If you performed a controlled experiment,* explain the experimental design. What is the hypothesis? What is the treatment? What is being controlled? What data did you collect, and how did you analyze it? Are the results significant? What are the potentially confounding factors, and how are they handled? Do the conclusions follow rigorously from the experimental data?
- *If you performed an empirical study,* explain what you measured, how you analyzed it, and what you concluded. What data did you collect, and how? How is the analysis related to the goal of supporting your claim about the result? Do not confuse correlation with causality.
- *If you use a small example for explaining the result,* provide additional evidence of its practical use and scalability.

5. How do you combine the elements into a research strategy?

It is clear that not all combinations of a research question, a result, and a validation strategy lead to good research. Software engineering has not developed good general guidance on this question.

Tables 1, 3, and 5 define a 3-dimensional space. Some portions of that space are densely populated: One common paradigm is to find a better way to perform some software development or maintenance task, realize this in a concrete procedure supported by a tool, and evaluate the effectiveness of this procedure and tool by determining how its use affects some measure (e.g., error rates) of quality. Another common paradigm is to find a better way to evaluate a formalizable property of a software system, develop a formal model that supports inference, and to show that the new model allows formal analysis or proof of the properties of interest.

Clearly, the researcher does not have free choice to mix and match the techniques—validating the correctness of a formal model through field study is as inappropriate as attempting formal verification of a method based on good organization of rules of thumb.

Selecting a type of result that will answer a given question usually does not seem to present much difficulty, at least for researchers who think carefully about the choice. Blindly adopting the research paradigm someone used last year for a completely different problem is a different case, of course, and it can lead to serious misfits.

Choosing a good form of validation is much harder, and this is often a source of difficulty in completing a successful paper. Table 6 shows some common good matches. This does not, unfortunately, provide complete guidance.

When I advise PhD students on the validation section of their theses, I offer the following heuristic: Look carefully at the short statement of the result—the principal claim of the thesis. This often has two or three clauses (e.g., I found an efficient and complete method …"); if so, each presents a separate validation problem. Ask of each clause whether it is a global statement ("always", "fully"), a qualified statement ("a 25% improvement", "for noncyclic structures…"), or an existential statement {"we found an instance of"). Global statements often require analytic validation, qualified statements can often be validated by evaluation or careful examination of experience, and existential statements can sometimes be validated by a single positive example. A frequent result of this discussion is that students restate the thesis claims to reflect more precisely what the theses actually achieve. If we have this discussion early enough in the thesis process, students think about planning the research with demonstrable claims in mind.

Concretely, Table 7 shows the combinations that were represented among the accepted papers at ICSE 2002, omitting the 7 for which the abstracts were unclear about validation:

Table 7. Paradigms of ICSE2002 acceptances

Question	Result	Validation	#
Devel method	Procedure	Analysis	2
Devel method	Procedure	Experience	3
Devel method	Procedure	Example	3
Devel method	Qual model	Experience	2
Devel method	Analytic model	Experience	2
Devel method	Notation or tool	Experience	1
Analysis method	Procedure	Analysis	5
Analysis method	Procedure	Evaluation	1
Analysis method	Procedure	Experience	2
Analysis method	Procedure	Example	6
Analysis method	Analytic model	Experience	1
Analysis method	Analytic model	Example	2
Analysis method	Tool	Analysis	1
Eval of instance	Specific analysis	Analysis	3
Eval of instance	Specific analysis	Example	2

6. Does the abstract matter?

The abstracts of papers submitted to ICSE convey a sense of the kinds of research submitted to the conference. Some abstracts were easier to read and (apparently) more informative than others. Many of the clearest abstracts had a common structure:

- Two or three sentences about the current state of the art, identifying a particular problem
- One or two sentences about what this paper contributes to improving the situation

- One or two sentences about the specific result of the paper and the main idea behind it
- A sentence about how the result is demonstrated or defended

Abstracts in roughly this format often explained clearly what readers could expect in the paper.

Acceptance rates were highest for papers whose abstracts indicate that analysis or experience provides evidence in support of the work. Decisions on papers were made on the basis of the whole papers, of course, not just the abstracts—but it is reasonable to assume that the abstracts reflect what's in the papers.

Whether you like it or not, people judge papers by their abstracts and read the abstract in order to decide whether to read the whole paper. It's important for the abstract to tell the story. Don't assume, though, that simply adding a sentence about analysis or experience to your abstract is sufficient; the paper must deliver what the abstract promises

7. Questions you might ask about this report

7.1. Is this a sure-fire recipe?

No, not at all. First, it's not a recipe. Second, not all software engineers share the same views of interesting and significant research. Even if your paper is clear about what you've done and what you can conclude, members of a program committee may not agree about how to interpret your result. These are usually honest technical disagreements, and committee members will try hard to understand what you have done. You can help by explaining your work clearly; this report should help you do that.

7.2 Is ICSE different from other conferences?

ICSE recognizes several distinct types of technical papers [6]. For 2002, they were published separately in the proceedings

Several other conferences offer "how to write a paper" advice:

In 1993, several OOPSLA program committee veterans gave a panel on "How to Get a Paper Accepted at OOPSLA" [9]. This updated the 1991 advice for the same conference [14]

SIGSOFT offers two essays on getting papers accepted, though neither was actually written for a software engineering audience. They are "How to Have Your Abstract Rejected" [26] (which focuses on theoretical papers) and "Advice to Authors of Extended Abstracts", which was written for PLDI. [16].

Rather older, Levin and Reddell, the 1983 SOSP (operating systems) program co-chairs offered advice on writing a good systems paper [11]. USENIX now provides this advice to its authors. Also in the systems vein, Partridge offers advice on "How to Increase the Chances Your Paper is Accepted at ACM SIGCOMM" [15].

SIGCHI offers a "Guide to Successful Papers Submission" that includes criteria for evaluation and discussion of common types of CHI results, together with how different evaluation criteria apply for different types of results [13]. A study [8] of regional factors that affect acceptance found regional differences in problems with novelty, significance, focus, and writing quality.

In 1993, the SIGGRAPH conference program chair wrote a discussion of the selection process, "How to Get Your SIGGRAPH Paper Rejected" [10]. The 2003 SIGGRAPH call for papers [21] has a description of the review process and a frequently-asked questions section with an extensive set of questions on "Getting a Paper Accepted".

7.3. What about this report itself?

People have asked me, "what would happen if you submitted this to ICSE?" Without venturing to predict what any given ICSE program committee would do, I note that as a research result or technical paper (a "finding" in Brooks' sense [3]) it falls short in a number of ways:

- There is no attempt to show that anyone else can apply the model. That is, there is no demonstration of inter-rater reliability, or for that matter even repeatability by the same rater.
- The model is not justified by any principled analysis, though fragments, such as the types of models that can serve as results, are principled. In defense of the model, Bowker and Starr [2] show that useful classifications blend principle and pragmatic descriptive power.
- Only one conference and one program committee is reflected here.
- The use of abstracts as proxies for full papers is suspect.
- There is little discussion of related work other than the essays about writing papers for other conferences. Although discussion of related work does appear in two complementary papers [19, 20], this report does not stand alone.

On the other hand, I believe that this report does meet Brooks' standard for "rules of thumb" (generalizations, signed by the author but perhaps incompletely supported by data, judged by usefulness and freshness), and I offer it in that sense.

8. Acknowledgements

This work depended critically on access to the entire body of submitted papers for the ICSE 2002 conference,

which would not have been possible without the cooperation and encouragement of the ICSE 2002 program committee. The development of these ideas has also benefited from discussion with the ICSE 2002 program committee, with colleagues at Carnegie Mellon, and at open discussion sessions at FSE Conferences. The work has been supported by the A. J. Perlis Chair at Carnegie Mellon University.

9. References

1. Victor R. Basili. The experimental paradigm in software engineering. In *Experimental Software Engineering Issues: Critical Assessment and Future Directives*. Proc of Dagstuhl-Workshop, H. Dieter Rombach, Victor R. Basili, and Richard Selby (eds), published as *Lecture Notes in Computer Science #706*, Springer-Verlag 1993.
2. Geoffrey Bowker and Susan Leigh Star: *Sorting Things Out: Classification and Its Consequences*. MIT Press, 1999
3. Frederick P. Brooks, Jr. Grasping Reality Through Illusion—Interactive Graphics Serving Science. *Proc 1988 ACM SIGCHI Human Factors in Computer Systems Conf (CHI '88)* pp. 1-11.
4. Rebecca Burnett. *Technical Communication*. Thomson Heinle 2001.
5. Thomas F. Gieryn. *Cultural Boundaries of Science: Credibility on the line*. Univ of Chicago Press, 1999.
6. ICSE 2002 Program Committee. *Types of ICSE papers*. http://icse-conferences.org/2002/info/paperTypes.html
7. Impact Project. "Determining the impact of software engineering research upon practice. Panel summary, *Proc. 23rd International Conference on Software Engineering (ICSE 2001)*, 2001
8. Ellen Isaacs and John Tang. *Why don't more non-North-American papers get accepted to CHI?* http://acm.org/sigchi/bulletin/1996.1/isaacs.html
9. Ralph E. Johnson & panel. How to Get a Paper Accepted at OOPSLA. *Proc OOPSLA'93*, pp. 429-436, http://acm.org/sigplan/oopsla/oopsla96/how93.html
10. Jim Kajiya. How to Get Your SIGGRAPH Paper Rejected. Mirrored at http://www.cc.gatech.edu/student.services/phd/phd-advice/kajiya
11. Roy Levin and David D. Redell. How (and How Not) to Write a Good Systems Paper. *ACM SIGOPS Operating Systems Review*, Vol. 17, No. 3 (July, 1983), pages 35-40. http://ftp.digital.com/pub/DEC/SRC/other/SOSPadvice.txt
12. William Newman. A preliminary analysis of the products of HCI research, using pro forma abstracts. *Proc 1994 ACM SIGCHI Human Factors in Computer Systems Conf (CHI '94)*, pp.278-284.
13. William Newman et al. *Guide to Successful Papers Submission at CHI 2001*. http://acm.org/sigs/sigchi/chi2001/call/submissions/guide-papers.html
14. OOPSLA '91 Program Committee. How to get your paper accepted at OOPSLA. *Proc OOPSLA'91*, pp.359-363. http://acm.org/sigplan/oopsla/oopsla96/how91.html
15. Craig Partridge. How to Increase the Chances your Paper is Accepted at ACM SIGCOMM. http://www.acm.org/sigcomm/conference-misc/author-guide.html
16. William Pugh and PDLI 1991 Program Committee. *Advice to Authors of Extended Abstracts*. http://acm.org/sigsoft/conferences/pughadvice.html
17. Samuel Redwine, et al. *DoD Related Software Technology Requirements, Practices, and Prospects for the Future*. IDA Paper P-1788, June 1984.
18. S. Redwine & W. Riddle. Software technology maturation. *Proceedings of the Eighth International Conference on Software Engineering*, May 1985, pp. 189-200.
19. Mary Shaw. The coming-of-age of software architecture research. *Proc. 23rd Int'l Conf on Software Engineering (ICSE 2001)*, pp. 656-664a.
20. Mary Shaw. What makes good research in software engineering? Presented at ETAPS 02, appeared in Opinion Corner department, *Int'l Jour on Software Tools for Tech Transfer*, vol 4, DOI 10.1007/s10009-002-0083-4, June 2002.
21. SigGraph 2003 Call for Papers. http://www.siggraph.org/s2003/cfp/papers/index.html
22. W. F. Tichy, P. Lukowicz, L. Prechelt, & E. A. Heinz. "Experimental evaluation in computer science: A quantitative study." *Journal of Systems Software,* Vol. 28, No. 1, 1995, pp. 9-18.
23. Walter F. Tichy. "Should computer scientists experiment more? 16 reasons to avoid experimentation." *IEEE Computer*, Vol. 31, No. 5, May 1998
24. Marvin V. Zelkowitz and Delores Wallace. Experimental validation in software engineering. *Information and Software Technology*, Vol 39, no 11, 1997, pp. 735-744.
25. Marvin V. Zelkowitz and Delores Wallace. Experimental models for validating technology. *IEEE Computer*, Vol. 31, No. 5, 1998, pp.23-31.
26. Mary-Claire van Leunen and Richard Lipton. *How to have your abstract rejected.* http://acm.org/sigsoft/conferences/vanLeunenLipton.html

A. Installing Eclipse Tooling

Please install a new copy of Eclipse by following the instructions below. Multiple instances of Eclipse in different versions can happily co-exist in your system. We recommend that you do not mess up with existing installations of Eclipse on your PC. You would likely break the setup for your existing projects (if you already use Eclipse), and likely the resulting installation would be slightly different than assumed in the course, leading to additional incompatibilities during the course (you do not want to waste time on them).

1. Make sure that at least Java SE 6, which is recommended for Eclipse 4.3.1, is installed on your computer. More details about installing the Java virtual machine (JVM) at http://wiki.eclipse.org/Eclipse/Installation.
2. Download Eclipse Modeling Tools (Release: Kepler 4.3.1) corresponding to your operating system from http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/keplersr1.
3. Copy the downloaded ZIP archive to a folder of your choice and unzip it.
4. Run Eclipse by double clicking the Eclipse icon in the newly created folder.
5. Use the proposed workspace folder or choose any other suitable folder. (A workspace is a folder structure visible from within the IDE that is used as a root folder for your projects. Projects are entities containing source code, models, configuration files, etc.). We recommend using a separate workspace folder for course projects, if you already use Eclipse for something else.
6. Select the *Help* menu, and then *Install Modeling Components*.
7. Select *Xpand* and *Xtext*. Press *Finish*. After dependencies have been computed hit *Next* and accept the license agreements. Wait until the software packages are downloaded and installed.
8. Restart Eclipse as requested.

B. Exercises on Compilers and Programming Languages

This is a preparatory exercise for the course. As such none of the tasks is mandatory and there is no hand-in due next week.

Objectives

- To recall basics of lexical and syntactic language specifications, syntactic categories
- To recall basics of type checking

I estimate about 4 hours to complete all these tasks is sufficient, plus possible extra reading time, if you forgot the details of regular expressions, grammars, etc.

Task 1. Explain what are the languages described by the following regular expressions:

A. $(ab)^*$

B. $1(0|1)^*$

C. $(\,(a(d|c)e)\_)^+$

Task 2. Write regular expressions describing the following languages:

A. A language of comma separated words; each word consisting of one or more occurences of the letter 'a'. The language contains no other symbols.

B. A language containing four words; each word is one of the keywords: if, then, else, while

C. White space consisting of spaces (␣), tab characters(\t), and new line characters (\n).

Task 3. Write a regular expression specifying identifiers as in the following quote from the ISO C standard: *An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different.*

Task 4. Explain in English what is the language described by this grammar:

$$
\begin{aligned}
S &\rightarrow T\ U\ \text{a}\ V \\
T &\rightarrow \text{John} \mid \text{Mary} \mid \text{Alice} \\
U &\rightarrow \text{reads} \mid \text{writes} \\
V &\rightarrow \text{book} \mid \text{letter} \mid \text{poem}
\end{aligned}
$$

Task 5. Write a grammar representing the language of balanced parentheses of three kinds, so "(", "{", and "[", where they can be arbitrarily nested as long as they are always balanced with a closing parenthesis of the same kind.

Task 6. Draw an abstract syntax tree (guess!) for the following Java expression and annotate nodes with types.

```
("3" + "4").length()*5
```

Task 7. Consider the following five example programs. Which part of the compiler will report an error in each of the programs ? (1) lexer/scanner (2) parser (3) name and type analysis? Or is the program statically correct?

A.
```
class Main {
    int main() {
       return x;
    }
}
```

B.
```
class Main {
    int main() {
       int x = 0;
       if (x == 0) {
          String x = "1";
          return x;
       }
       return 1;
    }
}
```

C.
```
class Main {
    void f() {
       int 1x = 0;
    }
}
```

D.
```
class Main {
    int main() {
       int x = 0;
       if (x == 0) {
          String y = "1";
          return x;
       }
       return 1;
    }
}
```

E.
```
class Main {
    int main() {
       if (x == 0) {
          String y = "1";
          return 0;
       }
}
```

Task 8. Indicate an expression, a statement and a declaration in the program D of Task 7.

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] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *SLE*, volume 6563 of *Lecture Notes in Computer Science*, pages 102–122. Springer, 2010.

[3] 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.

[4] Frank Budinsky, David Steinber, Ed Merks, Raymond Ellersick, and Timothy J. Groose. *Eclipse Modeling Framework.* Addison-Wesley, 2004.

[5] Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain-Specific Development with Visual Studio DSL Tools.* Addison-Wesley, 2007.

[6] 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.

[7] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming. Methods, Tools, and Applications.* Addison-Wesley, 2000.

[8] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches, 2006.

[9] Martin Fowler. *Uml Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Professional, 2004.

[10] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages.* Addison-Wesley, 2011.

[11] 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.

[12] 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.

[13] Daniel Jackson. *Software Abstractions.* MIT Press, 2006.

[14] 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.

[15] 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.

[16] Kang et al. Feature-oriented domain analysis (foda) feasibility study. Technical report, CMU/SEI-90-TR-21, 1990.

[17] 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.

[18] Steven Kelly and Risto Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, 2009.

[19] Anneke G. Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Addison-Wesley, 2009.

[20] 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.

[21] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[22] Richard F. Paige, Jonathan S. Ostroff, and Phillip J. Brooke. Principles for modeling language design. *Information & Software Technology*, 42(10):665–675, 2000.

[23] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer Verlag, 2005.

[24] 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.

[25] Mary Shaw. Writing good software engineering research paper. In *ICSE*, pages 726–737. IEEE Computer Society, 2003.

[26] Alan Snyder. How to get your paper accepted at OOPSLA. In *OOPSLA*, pages 359–363, 1991.

[27] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Wiley, 2005.

[28] Markus Voelter. *DSL Engineering. Designing, implementing and using domain specific languages*. 2013.

[29] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, 2003.

[30] David S. Wile. Lessons learned from real dsl experiments. *Sci. Comput. Program.*, 51(3):265–290, 2004.

[31] Claes Wohlin, Per Runeson, Martin Host, Magnus C. Ohlsson, Bjorn Regnell, and Anders Wesslen. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.

[32] Justin Zobel. *Writing for Computer Science*. Springer Verlag, 2004.