Marco Kuhrmann, Georg Kalus, Alexander Knapp

Rapid Prototyping for Domain-specific Languages

From Stakeholder Analyses to Modelling Tools

Today, modelling is a widely acisecepted technique in Software Engineering (SE). Many problems can be expressed using general-purpose modelling languages such as the UML. For more specific problems, the definition of a specialised domain-specific language (DSL) may be required. The definition of a domain-specific language is a time-consuming task that requires knowledge in (modelling) language design, deep understanding of the domain and, to be useful and usable, user assistance and tool support. In this paper, we present an approach to derive a domain-specific language from the description of instances of the domain under consideration: Stakeholders describe model instances from which the metamodel (the DSL) and a suitable modelling tool are derived automatically. We describe a tool that we used to experiment with this approach, its current state and the future work.

1 Introduction

A 'model' has several advantages over free-from sketches, as it has some degree of syntax and semantics and that it can be used to generate or derive other artefacts from it. However, defining a modelling language and corresponding modelling tools is laborious. If models are used mainly to clarify a particular domain, i.e., while analysing a customer's domain or a project's requirements, defining an appropriate modelling language is often not worth the effort. In addition, stakeholders not familiar with (or not interested in) modelling languages may not see the immediate benefit of the investment. A statement by a tool vendor highlights this problem: 'Nobody wants to perform *real* modelling, but only drawing pictures…' And in fact, tools such as Microsoft Visio or Omni Group's OmniGraffle, and even PowerPoint can be regarded as some of the most popular general-purpose 'modelling' tools.

On the other side of the spectrum, powerful modelling tools were developed: The Unified Modeling Language (UML) became a standardised modelling language and notation, and various UML dialects, such as SPEM (OMG 2008) or BPMN (OMG 2010), were created. Also, for certain domains specialised and comprehensive modelling approaches (including formalisms, notations, and tools) were developed, e.g., ARIS (Davis, R. 2010) or Focus (Schätz, B. 2001). Such dialects and specialised modelling approaches are well suited for their particular domain.

A popular approach for creating precise, specialised and easy-to-understand modelling languages are *domain-specific languages* (DSL). A domain-specific language facilitates (1) easy communication by using well-known and accepted domain objects in an appropriate notation, while keeping the (2) precision that enables further processing of the domain models, e.g., to generate code, data models, and so on. A domain-specific language is usually designed according to specific domain requirements and, therefore, a concrete domain-specific language is difficult to apply in other environments than the one it was developed for. The development of a specialised domain-specific language that may be used in just one project therefore may not be economically feasible as long as rapid language development — similar to rapid prototyping — is not well supported for domain-specific language development.

1.1 Problem Statement

Providing stakeholders, i.e., analysts, designers, and other project roles with appropriate modelling languages and tools beyond standard solutions is a challenging and costly undertaking. Domain-specific languages are a way to define special-purpose modelling languages. Current tools to develop DSLs, such as Eclipse EMF/GMF, Meta-Case, or the Visual Studio DSL-Tools require deep understanding of the tool itself and conceptual and technical knowledge. For specific problems, the effort necessary to develop a suitable domain-specific language therefore often seems too high compared to the potential benefit. The benefits of domain-specific languages could, however, be leveraged if there were means to quickly and iteratively develop a domain-specific language, similar to the rapid prototyping development paradigm.

1.2 Contribution

At the ICSE 2011 workshop on 'Flexible Modeling Tools' (Kuhrmann 2011) we discussed an early idea about how to make the language development process for domain-specific languages easier. The core of this idea was to interactively develop a domain-specific language by deriving it from an exemplary instance, which was 'drawn' during a stakeholder workshop.

In the paper at hands we take this idea one step further by reporting on an implementation of a DSL-based platform which generates a concrete domain-specific language from an instance-model scribbled on a 'virtual white board'. The instance modeler works in a drag-and-drop-style, similar to free-form drawing tools such as Microsoft Visio. The result is more than just pictures, but a concrete modelling language.

1.3 Outline

The remainder of the paper is organised as follows: In Sect. 2 we discuss related work, especially with regard to modelling of domain-specific languages and corresponding tools. In Sect. 3 we briefly describe the 'traditional' domain-specific language development process using our DSL-platform. In Sect. 4 we present our understanding of instance modelling and its impact to domain-specific language design. The presented approach is applied to a small case study in Sect. 5. Continuing with Sect. 6, we summarise the extended domain-specific language development method that facilitates rapid, instance-based language design. We conclude the paper in Sect. 7 and formulate the need for further research tasks.

2 Related Work

The field of modelling and meta-modelling is too wide to be covered exhaustively here. We therefore focus on basic concepts, current tools for meta-modelling and domain-specific language design, and new emerging ideas w.r.t. the advancement of meta-modelling.

Domain-specific Languages

We can roughly distinguish between the *general-purpose* approach, such as the UML (OMG 2011b), specific techniques for certain domains, e.g., Focus (Schätz, B. 2001), and the concept of *domain-specific languages* somewhere in between (Fowler and Parsons 2010; Kleppe, A. 2008). A domain-specific language is, essentially, a metamodel, which can be discussed from different perspectives. With regards to language design, some good definitions can be found in (Cook et al. 2007). We understand a metamodel to be a formalism to describe (domain-specific) languages, which can be understood by a computer.

Standard Meta-modelling Tools

In Eclipse-based language modelling tools (Steinberg et al. 2008), metamodels are represented by so-called Ecore models, which are based on the OMGs Meta Object Facility (MOF) hierarchy (OMG 2011a). The definition of a metamodel (a domain-specific language) is done using a UML-like notation subset. For instance, the Eclipse

Modeling Framework (EMF) provides rich support for the definition of metamodels (Steinberg et al. 2008), which is shown by many concrete EMF-based languages (e.g., the OMME tools (Folz and Jablonski 2010) and the considerable number of concrete EMF-based tools). Another example for such a tool is the Meta Case environment (MetaCase: Company's homepage and samples). Such a comprehensive support is important for language engineers to adjust all aspects of a modelling language (structure and semantics). However, for quickly capturing a domain, many of the powerful features are not required. The same can be said about the Microsoft Visual Studio DSL-Tools (Cook et al. 2007; Greenfield and Short 2004), which we used to develop PDE (Kuhrmann et al. 2010b). The Visual Studio DSL-Tools are not based on UML but also use a structured, XML-based approach to define data models and offer the possibility to add semantics and behaviour using source code.

New Ideas in Meta-modelling

The development of modelling languages and modelling environments is a widely discussed topic. Kimelman and Herschman (2011), for instance, discuss the need for ways to support formal modelling tools in a flexible manner. Similar to Cho et al. (2012), they argue the design of a modelling language should not be regarded as a Waterfall-like process, and highlight the necessity to dynamically move forth and back between informal (free) and formal models. Challenges resulting from this view are discussed by Cho et al. (2011). They discuss an approach that captures model instances by demonstration, and note that most domain-specific modelling is initially done using 'creativity' tools such as word processors, drawing tools or presentation tools. They conclude that the creation process of a domain-specific language has to take into account that (1) free form shapes have to be formalised, that (2) a metamodel needs to be formalised from model instances, and that (3) captured model instances have to be enriched by semantics.

We were facing similar challenges when designing DSL-based meta-modelling tools (Kuhrmann 2011). In Cho et al. (2012) a first implementation of the concept described in Cho et al. (2011) is presented. This implementation results, however, in a 'drawing' tool. Beyond sketches and ideas, Volz et al. (2011) present a multi-layer modelling environment that not only supports meta-modelling but also the connection of models at different levels of abstraction. They motivate their approach with the observation that users often use different tools and that a solution for bridging the gap could be to integrate all models using one modelling language and creating connections among the models.

Discussion

The design of a domain-specific language needs support in at least two areas: (1) to provide language 'end users' with a modelling tool that supports them in creating and handling concrete model instances and (2) to assist language engineers during the definition of a domain-specific language.

Almost all DSL tools address the first aspect. Eclipse EMF/GMF for instance supports textual as well as visual domain-specific languages and provides corresponding Eclipse-integrated editors. With our work on PDE we followed an alternative approach where an editor is generated from the domain-specific language. The domain-specific language is 'baked into' the resulting editor. The end users are provided with a specific modelling tool (stand-alone or IDE-hosted) according to their needs (Kuhrmann et al. 2010b).

Comprehensive support for language engineers to define a domain-specific language is only partially addressed. Eclipse, MetaEdit, and Visual Studio provide comprehensive support for the language engineers if the domain of action is known and analysed. If the language engineer should capture a domain and derive a domain-specific language, no adequate support is given — especially for scenarios such as a domain analysis workshop, which is done with the stakeholders.

3 PDE Language & Tool Development

A result of the research described here is the Process Development Environment (PDE). The platform is published as an Open Source project and can be accessed at `http://pde.codeplex.com`. PDE provides an infrastructure for the design of domain-specific languages in general, and process languages and process authoring tools in particular. The core functionality is based on the Microsoft Visual Studio DSL-Tools (Cook et al. 2007). PDE adds several features, such as:

- Improved visual design
- Model visualisation
- Metamodel modularisation
- Hooks for validation functions

3.1 The PDE Platform

The PDE framework consists (Fig. 2) of two parts: The first part is PDELanguage, which is an extension of the Visual Studio DSL-Tools. The second part is PDE Framework that serves as the shell for the editors that are generated from a domain-specific language. The PDE Framework consists of a ToolFramework and a concrete PDE-based language that is an instance of the PDELanguage (Fig. 1). Such a language consists of a ViewModel and a DomainModel. The DomainModel is the executable language in which functions, such as validation or serialisation are configured. A concrete PDE-based language might contain different view models, e.g., a tree view, a graphical editing pane, or a property grid. The ToolFramework is a comprehensive *Windows Presentation Foundation*-based application frame, which uses the .NET Framework. Using the MEF interfaces (Managed Extensibility Framework 2010) the application frame is extendable, i.e., by new views extending the ViewModel or additional functionality provided by separate plug-ins.

A concrete language (a metamodel) is a domain-specific language, which is based on the PDE extension of the Visual Studio DSL-Tools. The PDE Language, which is itself a 'meta-meta model[1]' (Fig. 1), is the basis for the metamodel, which is merged with the PDE Tool Framework into a concrete modelling tool (stand-alone or hosted in Microsoft Visual Studio) for language engineers or modelers respectively. A comprehensive description of PDE can be found in (Kuhrmann et al. 2010a).

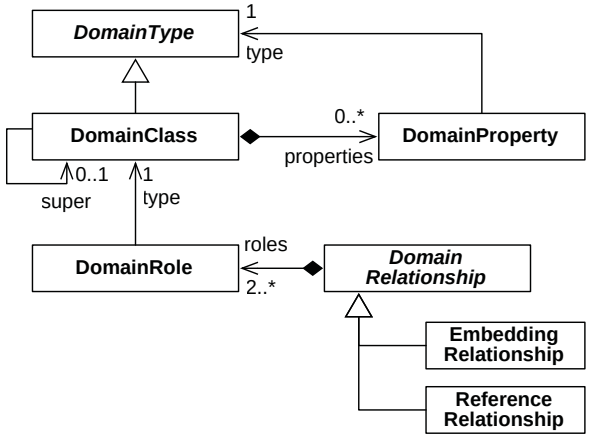

Figure 1: Concept meta model of PDE-based languages.

3.2 Creating a PDE-based Language

In the following we describe the typical language development process using PDE. This process is usually gone through only once per metamodel. However, if the metamodel is updated, parts of the process have to be repeated to incorporate the changes. Figure 3 shows the (classic) language development process consisting of up to five steps.

Step 0 In the step 'PDE Language Development' the PDE Language itself can be manipulated or extended. This step influences the behaviour of the whole platform. It should not be done without deep knowledge of the platform. For a language engineer who just wants to define a new domain-specific language for a customer, this step is usually unnecessary.

[1]Note that the meaning of meta-meta model is aligned with the definition provided by the MOF (OMG 2011a).

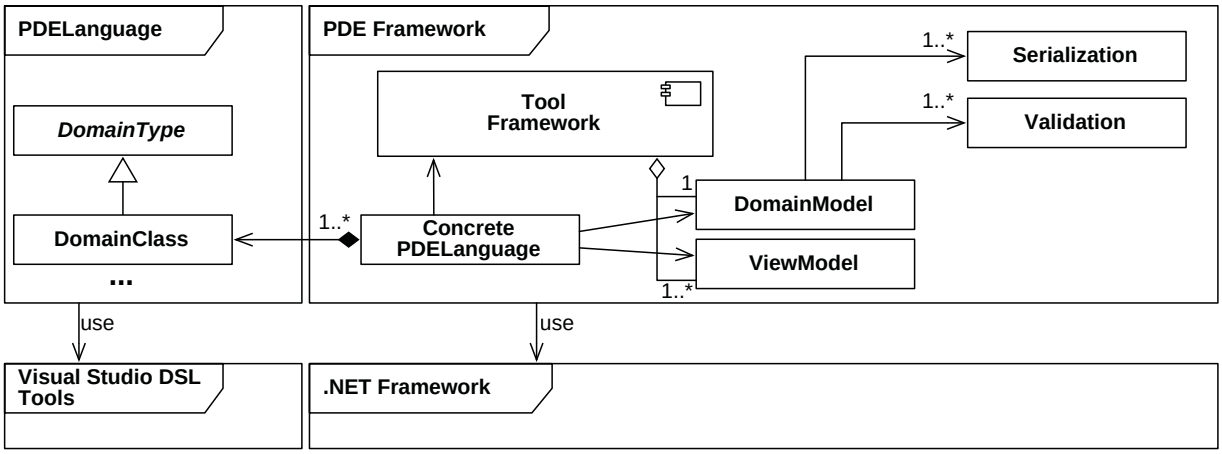

Figure 2: Architecture of the PDE platform (simplified).

Step 1 The step 'Implement a Metamodel' involves the implementation of the metamodel (the domain-specific language) within the specification of the PDE Language. Elements and relationships of the domain-specific language are transformed into *domain classes* and *domain relationships* of the concrete metamodel. Additionally, the domain-specific language can be extended to provide multiple views (graphical notations) to present different aspects of the model.

Step 2 In the step 'Transform' the transformation process of the platform utilises T4 (Sych 2007) templates to generate the source code representing the domain-specific language for integration in the PDE Editor Framework.

Step 3 The third step covers two aspects: the extension (step 'Extend') and the customisation (step 'Customise') of the transformed domain-specific language. In this step the generated code can be extended or customised for different purposes, i.e., serialisation methods can be overridden to define a custom serialisation format or new validation methods can be provided to check for specific constraints. New views etc. can also be provided in this step, i.e., sophisticated views that combine certain aspects of the underlying model to ease the modelling or to foster the discussion with stakeholders.

Step 4 Step 4 is the last step in which the final editor is created. Depending on the audience of the editor and the initial PDE-template, either a stand-alone tool is created, or an editor, which is hosted in the Visual Studio environment.

The language development process is designed to allow short development cycles if a domain-specific language needs to be changed and evaluated (as discussed by Kuhrmann et al. (2010b) and demanded by Cho et al. (2012)). Besides the above listed steps there is a standard path for the language development consisting of the steps 1, 2, and 4.

Step 3 can be skipped if the features that are initially provided by the platform fit (almost) all requirements. In this step additional components can be introduced to the tool, i.e., specialised visualisation components or additional logic.

Besides this static binding of additional components, PDE also provides a MEF-based (Managed Extensibility Framework 2010) plugin interface to discover and load separately developed plugins at runtime, e.g., additional logic for runtime validation, or features that cannot or can only

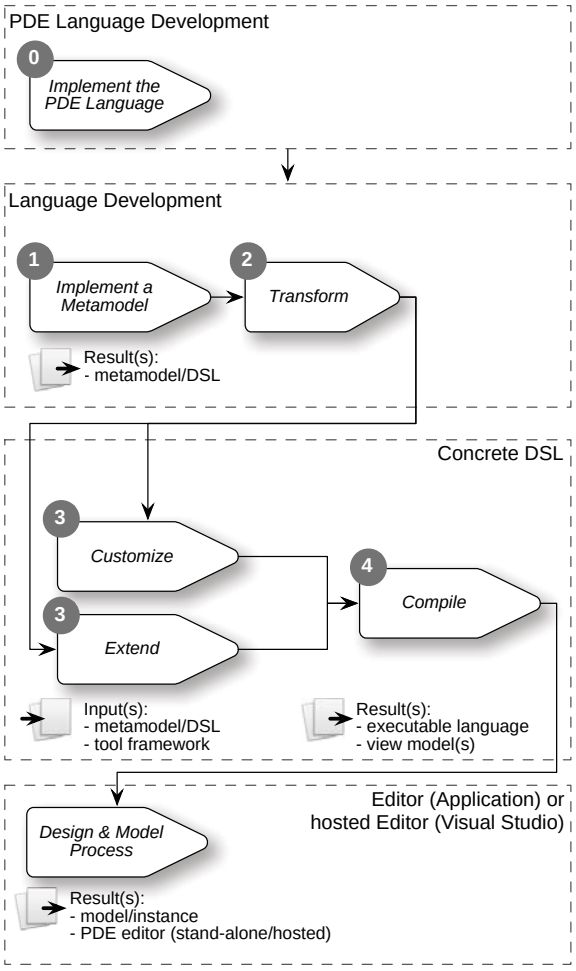

Figure 3: The language development process for creating a new domain-specific language using PDE.

hardly be expressed in the domain-specific language itself.

4 Instance Modelling

The outcome of the aforementioned language development process are the modelling language (for the engineers/designers) itself and corresponding modelling tools (for engineers and authors), which are either a stand-alone or a Visual Studio-hosted tool. The experiences with assistants like the built-in one of Visual Studio showed that the PDE-based process does make the language definition easier, especially if a comprehensive editor for models based on the language should be available.

However, as the PDE Language designer is integrated with Visual Studio, its application requires some technical understanding. The language designer component is far away from being easy to understand for non-expert stakeholders and supports the definition of a domain-specific language on a fairly technical level.

4.1 Instance Modelling — The Idea

In a cooperative and iterative modelling approach creative and formal tasks overlap to a certain extent (Cho et al. 2012; Kuhrmann 2011) — especially if a new domain needs to be 'explored' in a stakeholder workshop.

Therefore, the idea of *instance modelling* can be described as follows: A stakeholder workshop to understand and capture the domain is done 'as usual' — but instead of using a classical whiteboard, a digital 'informal' modelling pane is used. This pane collects domain entities, which are represented visually, and simple associations. In the workshop, entities can be collected, structured, combined, and so on.

The goal is to express the domain using prototypical model instances as representatives for the domain under consideration. Stakeholders are able to describe the domain from their perspective and experience. Thus, instance modelling currently aims at complementing the elaboration of a new domain-specific modelling language in cases where the domain still has to be explored. Changes to existing domain-specific modelling languages in the sense of model evolution have not been considered.

4.2 The Approach

In an instance modelling workshop, a prototypical instance of the envisioned domain-specific modelling language is drawn. A domain element, like a role or an artefact, is captured by an *element* class with some *attribute* slots filled; relationships between domain elements, like a role

being responsible for an artefact, are either represented by a *reference*, expressing a directed connection, or by an *embedding*, expressing a whole-part relationship (cf. Fig. 4, step 1).

Behind the scenes, the language-modelling tool captures the model prototype and translates into (or derives) PDELanguage constructs to prepare the definition of the target domain-specific language. The domain-specific language is mostly the result of the informal design and builds the basis to rapidly create the new DSL.

PDE infers the language using the following mapping:

- Element class ↦ DomainClass
- Reference ↦ ReferenceRelationship
- Embedding ↦ EmbeddingRelationship

Furthermore, attributes associated with elements are mapped to DomainProperties; also a set of similar properties can be mapped to one domain property, e.g., a property 'Name'. Primitive types, e.g., Int or String, are directly mapped to predefined domain types of PDE.

The modelling process is triggered by user interactions that are caught by the ViewModel (Fig. 2). The editor realises, e.g., drag and drop events and the framework calls methods that, for instance, create new domain entities. Figure 4 shows an example: When dragging an image onto the modelling pane (step 2 of Fig. 4) the ViewModel catches the assigned event and creates a new instance of a DomainClass (Fig. 1). Having added the new entity to the instance model, the language engineer can edit the entity, e.g., editing name, adding attributes, create relationships to other entities, and so on.

The resulting domain-specific language can be used to create or generate various tools, in particular, for modelling. Currently, only a PDE export is implemented, but other meta-modelling tools, such as EMF/GMF, could be targeted. Therefore, an appropriate SerialisationFormatter has to be added to extend the PDE Framework (Fig. 2).

The stakeholders use the resulting modelling tools. The style of modelling, the notation and the semantics comply with the drafts made during the language creation workshops.

4.3 DSL Optimisation

So far, we only changed the 'input channel' for the design of the domain (see language development process in Fig. 3, step 1). One could say, this is just another front end to the PDE Language designer. However, we have to take into account that we consider two different ways of creating domain-specific languages:

1. The first option to create a domain-specific language—creating a language based on solid information gathered beforehand—works fine if the language engineer has knowledge about the domain. In consequence a frequent interaction with the stakeholders might be unnecessary, and the design of a domain-specific language is close to, e.g., UML architecture design.
2. The second way to create a domain-specific language is a more 'exploratory' approach. It is applied in settings, where language engineers need to get initial information about the considered domain. At this point instance modelling replaces domain analysis workshops by an interactive design of the domain, represented by exemplary instances.

Figure 4 gives an idea of the second scenario of designing a domain-specific language: A language engineer asks a stakeholder for a domain entity, i.e., a role, drags a picture that symbolises this entity onto the modelling pane, and starts to refine this entity (i.e., adding attributes) during the interview with the stakeholder. From a technical point of view, this way is rather 'pragmatic' and does not result in an optimal domain-specific language.

Therefore, optimisation is performed before finalising the language. PDE analyses those captured domain entities, derives domain types, and analyses them for optimisation opportunities, e.g.,

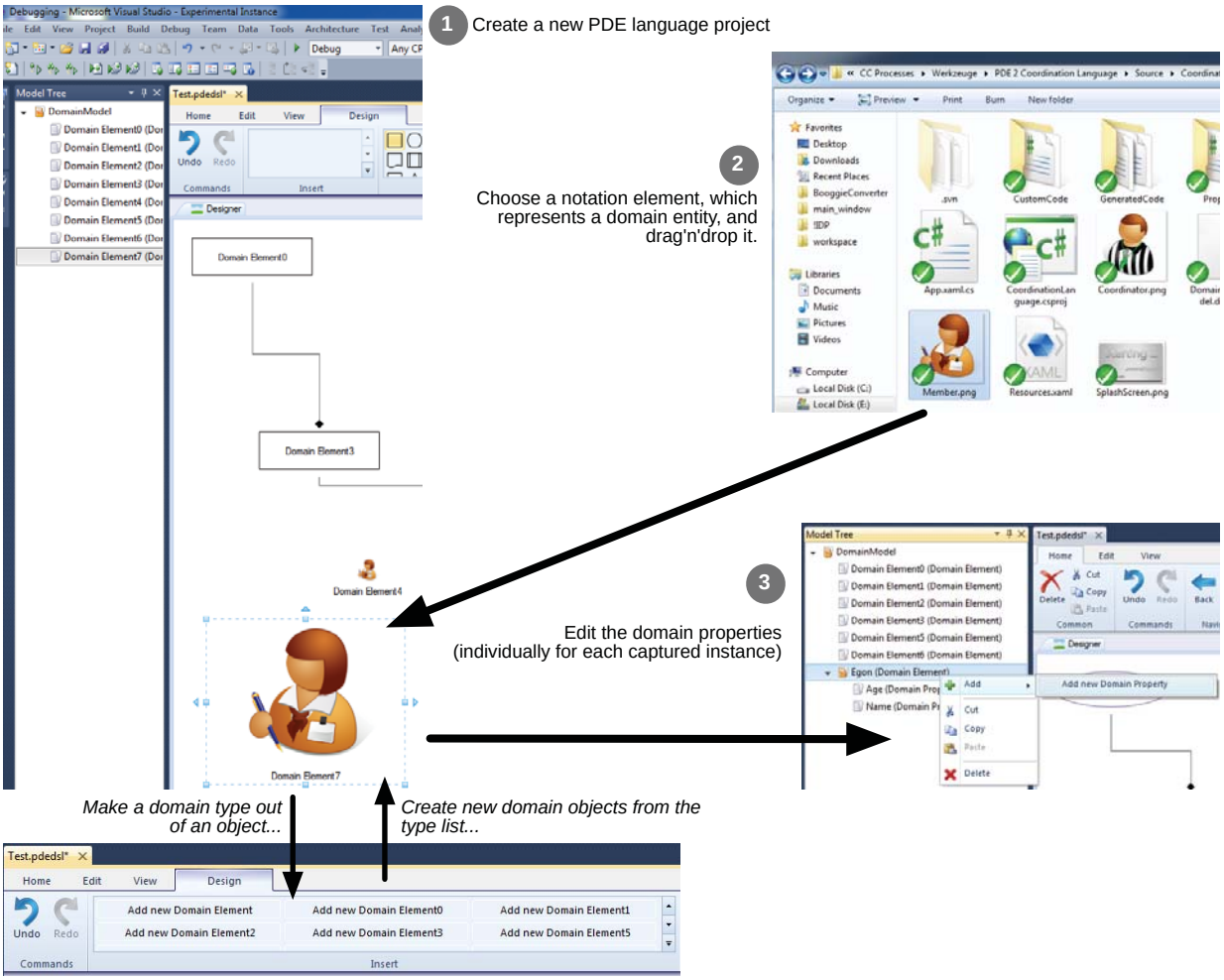

Figure 4: Concept of instance design: A PDE-based Visual Studio-hosted editor pane is used to capture model instances. A model explorer allows for the precise definition of types, domain properties, and, finally, for the derivation of a concrete modelling language.

common attributes that can be extracted into a base class. This approach is based on refactoring (Fowler et al. 1999) and is applied on tentative languages. Currently, the platform allows for optimisations triggered by properties and relationships. Figure 5 shows two examples: In the left part of the figure a first optimisation strategy is shown. Starting with a number of designed domain entities, PDE analyses those entities for potentially 'sharable' attributes. The analysis criteria are the name of an attribute as well as its domain type. If there were any attributes meeting those criteria, PDE asks, whether those should be refactored using a shared base class for the shared attribute. In the second optimisation opportunity, PDE analyses the captured model for similarities of relationships (Fig. 5, depicted on the right). Reference relationships are deemed similar if their roles are named equally and are of the same type. If the platform finds candidates, it asks, whether a shared base class should be created that realises the extracted relationship.

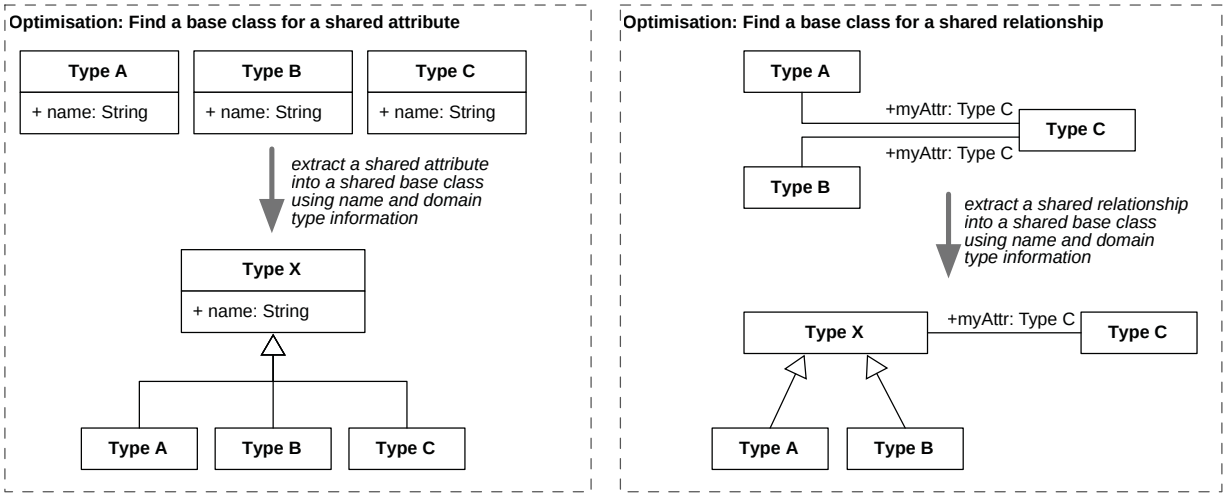

Figure 5: Language optimisation: A captured instance is analysed and optimised in order to create a 'real' domain-specific language. The platform proposes possible optimisations.

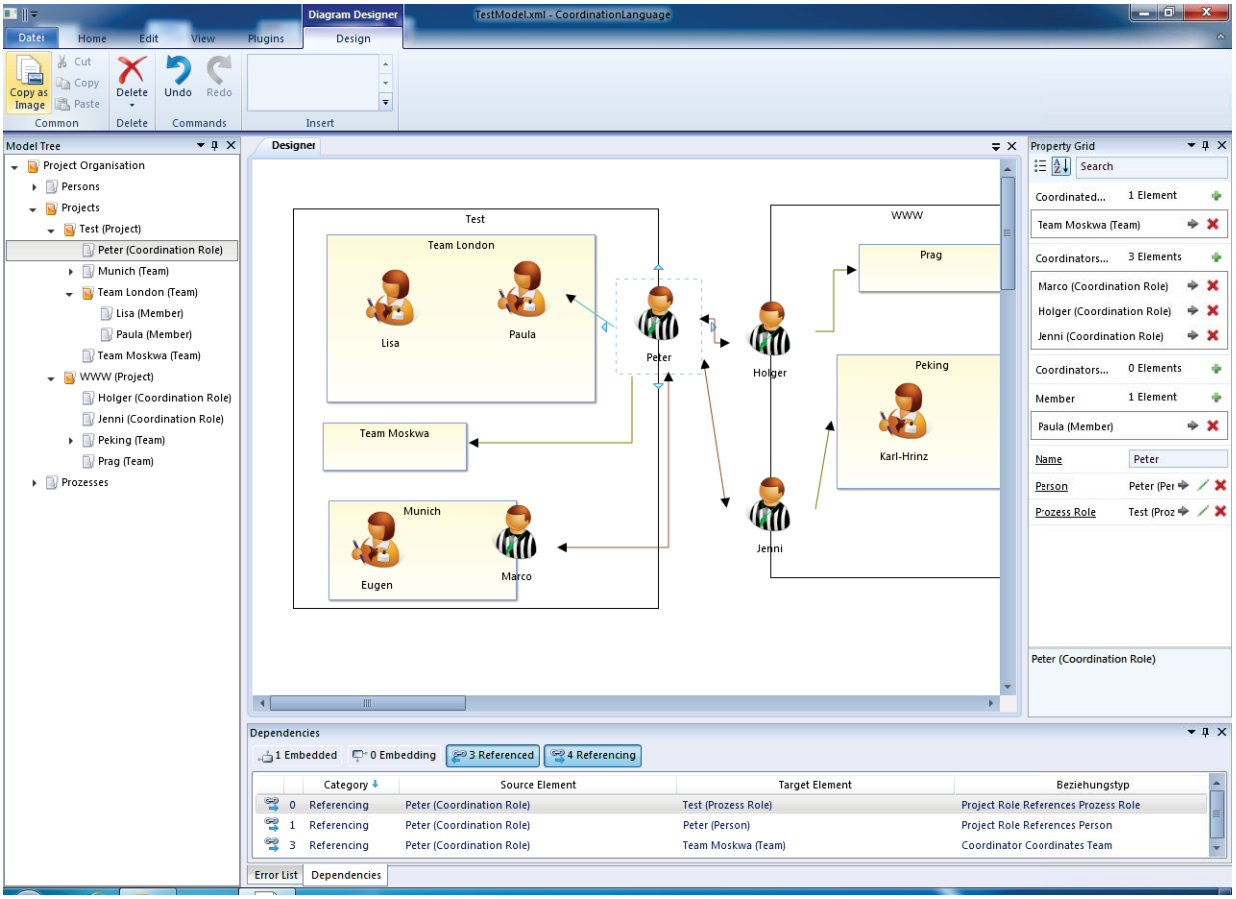

Figure 6: The classically designed DSL and the generated tool according to (Kuhrmann et al. 2010b).

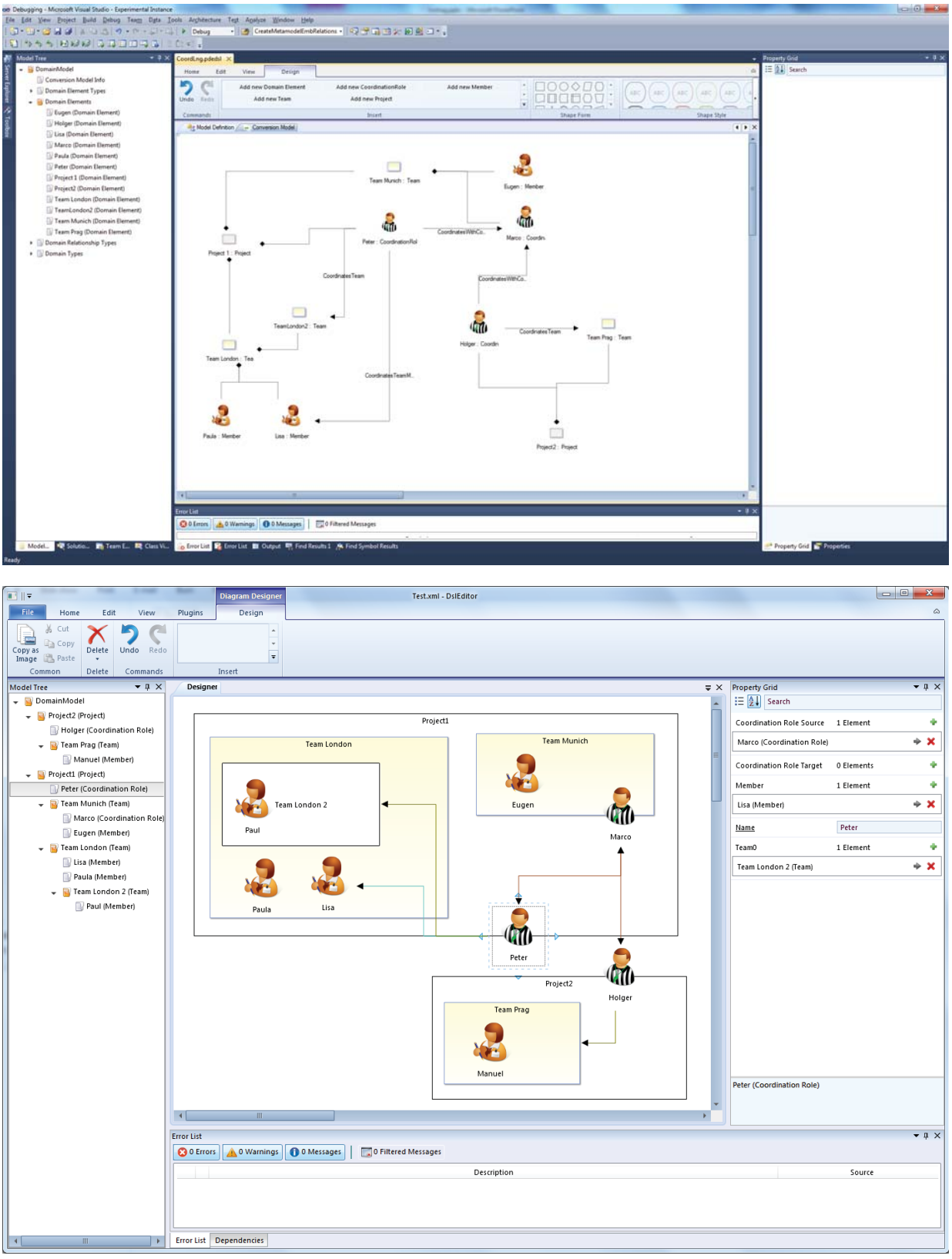

Figure 7: The modeled instance that derives the team modelling DSL (upper part) and the generated editor for the team modelling DSL (lower part).

5 Case Study

Since our research was exploratory, we opted for a case that gave us a 'reference point': In (Kuhrmann et al. 2010b) we discussed PDE using a small domain-specific language for modelling teams at different sites in a globally distributed development project. For the realisation of the instance modeler the primary requirement was that the inferred domain-specific language (including the resulting modelling tools) was at least as powerful as the classically designed one.

Figure 6 shows the editor and an exemplary model of the so-called *team coordination language*[2], which was developed using the 'classic' PDE DSL development process (cf. Fig. 3). According to our key requirement, a domain-specific language which is created based on the instance modelling approach has to contain all the domain types, the domain attributes, and so on. Furthermore, the resulting editor should have the same appearance as the 'classic editor'. Consequently the new editor has to open and read concrete models that were designed using the old editor.

Figure 7 shows in the upper part the Visual Studio-hosted instance modeler and a captured instance of a team model, including different sites, relationships, and different kinds of team members. According to the aforementioned development process (instance capturing, optimisation), the instance modeler add-on to PDE infers a domain-specific language from the instance. The inferred language is the input (see Fig. 3) for the generation of an editor (lower part of Fig. 7). The selected simple case shows that the key requirement was completely achieved. The domain-specific language that was created using the instance modelling approach was, finally, a 'clone' of the originally designed one. Even the models, which were created with the old editor, could be opened with the new editor.

[2]Based on the keynote 'Speculations on Coordination Models' by Len Bass at the International Conference on Global Software Engineering in Princeton, 2010.

6 Extended Language Development

Based on instance modelling we re-define the language development process. Figure 4 shows a series of screenshots that illustrate the extension. Beside the 'classic' approach as described in Sect. 3.2 a language engineer can open a new PDE-DSL-Project (1). The editor pane is hosted in a Visual Studio environment.

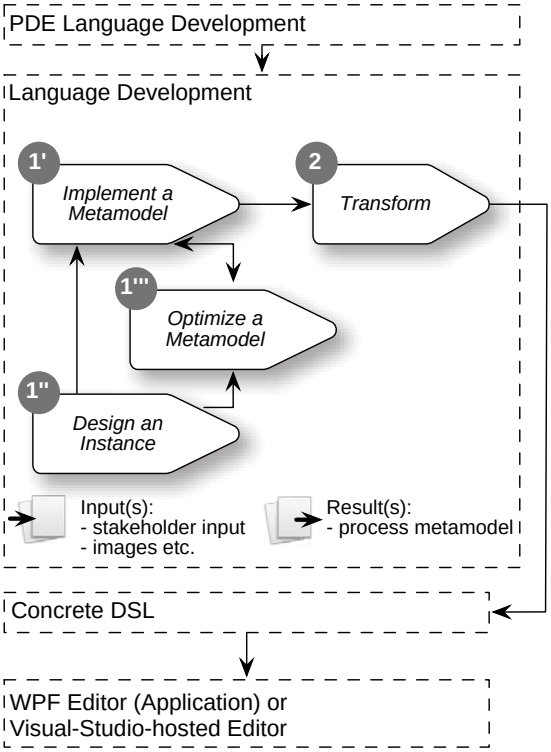

Figure 8: Extended language development process w.r.t. instance modelling.

To create a new graphical language element (domain type), the language engineer only needs to drag and drop, e.g., an image onto the pane (2). The picture immediately becomes a domain type to which corresponding attributes can be added (3). Furthermore, the domain type is also placed in the design ribbon and can be used to create new domain objects of the type. Having drawn the instance of interest, the mechanism described in Sect. 4 comes into play to (1) generate a PDE-based metamodel, and to (2) optimise the inferred metamodel.

Figure 8 shows the modification of the language development process, which was described in Fig. 3. Instead of simply modelling a domain-specific language, another way to create a domain-specific language is added. The first step is now to directly *Implement a Metamodel* (step 1′) **or** to *Design an Instance* (step 1″), which is transferred into a domain-specific language. Additionally, step 1‴ *Optimise a Metamodel* can be executed to optimise a designed or a directly implemented metamodel. The outcome of those steps is, itself, input for the transformation step, which leads to the classical language development process.

7 Conclusion & Future Work

We presented an extension to our PDE platform for instance modelling to capture domain models and to transform them into a domain-specific language. The user-centered part of domain modelling is similar to approaches known from drawing tools, such as Microsoft Visio and therefore easier to learn and understand for non-technophile stakeholders. The PDE platform creates domain-specific languages from such drawn figures in the background and provides language engineers with some refactoring-like optimisation capabilities.

Summarised, the instance modelling extension allows users to draw a figure of the currently considered domain, and creates a domain-specific language from the drawing.

In (Kuhrmann 2011) we already discussed first ideas, concepts, and prototypes. We also discussed some challenges — similar to Cho et al. (2011) — e.g., semantics of pictures, language derivation and respective mappings, or structuring of complex languages. We furthermore discussed, if 'the modelling pane is just another domain-specific language' and 'and to what extent a user-defined domain-specific language can be derived automatically?'

Currently, we have a first prototype that allows to derive a domain-specific language from one particular instance (Sect. 5). Also, we decided to realise the PDE extension as a domain-specific language, too. The mechanism behind the prototype is, therefore, a model transformation at the PIM to PIM level (Kleppe et al. 2003).

Future Work

This paper outlined some (promising) steps to support rapid language design. In ongoing research we have first to improve the capabilities of domain capturing and the language derivation techniques. Here, we need to extend our prototype to be able to extract a domain-specific language from different instances instead of only one. Furthermore we have to improve the usability. Although the user interface is already quite simple and easy to understand, even for non-technophile stakeholders, the working process is, still, complex and requires expertise. Without guidance of a PDE-trained language engineer 'ordinary' users are certainly not able to perform domain capturing.

Finally, we validated our idea against only a few key requirements and provided a proof of concept. We need, however, to intensively evaluate the feasibility and the economic impacts of our ideas to answer the questions: Is the instance modelling approach faster? Is the quality of the generated (and optimised) domain-specific languages comparable to those that are developed the classic way? How much money can be saved using this approach? To this end, the most recent release of the PDE platform, including the instance modelling add-on, is available at `http://pde.codeplex.com` to everybody for testing and evaluation purposes.

Acknowledgments

We want to thank Eugen Wachtel and Manuel Then for their essential support during the development of PDE. We also thank Sebastian Eder for reviewing this paper.

References

Cho H., Gray J., Sun Y., White J. (2011) Key Challenges for Modeling Language Creation By Demonstration. In: ICSE Wsh. Flexible Modeling Tools

Cho H., Gray J., Syriani E. (2012) Creating Visual Domain-Specific Modeling Languages from End-User Demonstration. In: Int. Wsh. Modelling in Software Engineering (MiSE)

Cook S., Jones G., Kent S., Wills A. C. (2007) Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley

Davis, R. (2010) ARIS Design Platform: Advanced Process Modeling and Administration. Springer

Folz B., Jablonski S. (2010) OMME — A Flexible Modeling Environment. In: SPLASH Wsh. Flexible Modeling Tools

Fowler M., Beck K., Brant J., Opdyke W., Roberts D. (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley

Fowler M., Parsons R. (2010) Domain-Specific Languages. Addison Wesley

Greenfield J., Short K. (2004) Software Factories. Wiley & Sons

Kimelman D., Herschman K. (2011) A Spectrum of Flexibility – Lowering Barriers to Modeling Tool Adoption. In: ICSE Wsh. Flexible Modeling Tools

Kleppe, A. (2008) Software Language Engineering. Addison-Wesley

Kleppe A., Bast W., Warmer J. B. (2003) MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley

Kuhrmann M., Kalus G., Then M., Wachtel E. (2010a) From Design to Tools: Process Modeling and Enactment with PDE and PET. In: ASE Wsh. Academic Software Development Tools and Techniques (WASDeTT-3)

Kuhrmann M., Kalus G., Wachtel E., Broy M. (2010b) Visual Process Model Design using Domain-specific Languages. In: SPLASH Wsh. Flexible Modeling Tools

Kuhrmann M. (2011) User Assistance during Domain-specific Language Design. In: ICSE Wsh. Flexible Modeling Tools

Managed Extensibility Framework. Online: http://mef.codeplex.com/

MetaCase: Company's homepage and samples. http://www.metacase.com

OMG (2008) Software & Systems Process Engineering Metamodel Specification (SPEM) Version 2.0. Specification. Object Management Group

OMG (2010) Business Process Model and Notation (BPMN) Version 2.0. Specification. Object Management Group

OMG (2011a) Meta Object Facility (MOF) Core Specification Version 2.4.1. Specification. Object Management Group

OMG (2011b) Unified Modeling Language (UML): Superstructure Version 2.4.1. Specification. Object Management Group

Schätz, B. (2001) The ODL Operation Definition Language and the Autofocus/Quest Application Framework AQUA. Tech. Rep. TUM-I0111. Technische Universität München

Steinberg D., Budinsky F., Paternostro M., Merks E. (2008) EMF: Eclipse Modeling Framework, 2nd ed. Addison-Wesley

Sych O. (2007) T4: Text Template Transformation Toolkit. http://www.olegsych.com/2007/12/text-template-transformation-toolkit/

Volz B., Zeising M., Jablonski S. (2011) The Open Meta Modeling Environment. In: ICSE Wsh. Flexible Modeling Tools

Marco Kuhrmann, Georg Kalus
Technische Universität München
Faculty of Informatics
Boltzmanntstr. 3
85748 Garching
{kuhrmann | kalus}@in.tum.de

Alexander Knapp
Universität Augsburg
Institute of Informatics
Universitätsstr. 6a
86159 Augsburg
knapp@informatik.uni-augsburg.de