Schema-Driven Development of Semantic MediaWikis

MASTER THESIS

18th July 2015

Simon Heimler
Master of Applied Research (MAPR)
in Computer Science

Prof. Dr. Wolfgang Kowarschick
Faculty of Computer Science
University of Applied Sciences Augsburg

computerbauer
IT für Steuerberater

Hochschule
Augsburg University of
Applied Sciences

Abstract

As we developed a knowledge management system for Computer Bauer GmbH, we soon reached some limits of our chosen target platform, Semantic MediaWiki. Those limits were not missing features, but the increasing costs of developing and maintaining our rather complex model.

For that reason, I chose to build a tool that supports the development process by generating the final implementation code from a simpler, more abstract and object oriented model.

However, I didn't want to obtain the full complexity of conventional model-driven approaches. Instead, I've developed an approach that is as simple as possible while still meeting our modeling requirements. As the approach uses a schema language to describe the development model, I'll call the approach Schema-Driven Development (SDD).

In the theoretic part of this thesis, I'll introduce this approach on a generic and theoretical basis. Terms are defined; benefits and concepts from MDE/MDA are evaluated and put in context to the SDD approach. The main subject of the theory chapter is the use of a schema language as the primary model language and my proposal for a modular generator architecture.

The praxis part puts the discussed theoretical foundations and proposals into practice. It documents the development and technological decisions of my own SDD generator and its domain specific language based on the previously introduced principles and architecture. I decided to use JSON Schema as its schema language and YAML/JSON as the notation format.

I'll also evaluate how the SDD approach worked regarding the chosen target platform and our modeling requirements. It turned out to be well suited for our specific use case. It made an agile, rapid prototyping workflow possible that we as a team found very useful and led to a successful realization of our project.

Keywords: MDE, Model-Driven Engineering, MDD, Model-Driven Development, SDD, Schema-Driven Development, JSON Schema, Knowledge Management, Semantic MediaWiki, MediaWiki, Node.js

Table of Contents

List of Figures

Abbreviations

CMS	Content Management System
DRY	Don't Repeat Yourself
KMS	Knowledge Management System
MD*	The various Model Driven Approaches
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MDD	Model-Driven Development
MW	MediaWiki
SF	Semantic Forms
SDD	Schema-Driven Development
SMW	Semantic MediaWiki
UI	User Interface
UX	User Experience

1 Introduction

This chapter introduces the subject of the thesis. I'll also explain the motivation and some of the applied basic principles behind this thesis.

1.1 Preface

“I” refers to the author of this thesis, Simon Heimler.

“We” refers to my company Computer Bauer GmbH, especially Mathias Bauer, Moritz Abraham and Sebastian Schlegel, my advisor Prof. Dr. Wolfgang Kowarschick and, myself.

I’d like to thank them for their support and pleasant teamwork. Thanks to Moritz Abraham and Yaron Koren for proofreading.

I dedicate this work to my daughter, Sarah Heimler.

1.2 Background

This is my master thesis of the university course Master of Applied Research (MAPR) in Computer Science. Since it is currently a rather unusual degree, let me give you a short description first:

The MAPR is an applied research and industry oriented degree, split into a part-time master study, and part-time (paid) job for a cooperating research facility or company. The student has the freedom to choose fitting lectures and seminars and writes the master thesis on the subject of his work.

In my case, the cooperating company is Computer Bauer GmbH, an IT company providing hosting services for tax consultants, located in Munich, Germany.

The assignment from the company was to build a structured and integrated Knowledge Management System that handles IT Management information, CRM and general company knowledge.

The combination of research and application also influenced the structure of this thesis. Chapter 2 lays the theoretical groundwork of the project. The theory is grounded and evaluated through practice, which will be discussed in Chapter 3.

1.3 Motivation

To implement an internal knowledge management system for the company, we chose to use Semantic MediaWiki. Soon after we started developing the first drafts of our model, we reached a point where the model became increasingly hard to administer and maintain. It became difficult just to notate the model - visual representations were only helpful to a certain point. Since we were in a completely theoretical conceptualization phase, we could also not be confident about our decisions and their consequences.

To overcome those problems we needed a way to notate our development model in a scalable, abstract, and concise way. In order to ensure we made the right decisions, we would like to view, test and evaluate them in an actual working system.

With the chosen technologies, this approach would have been difficult, time consuming and error prone. Therefore, I started to develop a new toolset called mobo[1] that provided the workflow and agility we needed for our project.

The main technological decision that drives the workflow is the use of model-driven development techniques. I wanted the system to be as simple and easy as possible, especially since I had to develop it on my own besides the actual task of modeling/creating the KMS for Computer Bauer.

Therefore, I have started to write a MDE system from scratch, using much simpler technologies and concepts that are usually common. Because this approach goes a few unusual routes, I would like to introduce a new term for it: Schema-Driven Development (SDD).

This approach, in both theory and practice, will be the main topic of this thesis.

1.4 Principles

1.4.1 Introduction

There are a few principles (some are a matter of taste) that have guided my decisions, regarding the choice of technologies and architecture I propose in Chapter 2 and Chapter 3.

To understand those choices better, let me introduce the principles first.

1.4.2 Do One Thing and Do It Well

There is a commonly known UNIX maxim: “Do One Thing and Do It Well”, coined by Doug McIlroy[2]. The principle leads to many modular but highly specialized tools that develop their true potential when being used together.

This is the reason SDD focuses on data schemas as the model basis. Because the scope is limited, the system does not need to support every eventuality.

Of course, this restricts the SDD approach to some areas (or subareas). However, it is possible to combine it with traditional software development or MDE techniques, if the field of application can be sufficiently isolated.

[1] Simon Heimler, *mobo*

[2] M. D. McIlroy, E. N. Pinson, B. A. Tague, ‘Unix Time-Sharing System Forward’, 1902

1.4.3 Strive for Simplicity, stack necessary Complexity

When solving a problem, the simplest solution is preferable.

> *The only way to avoid these traps is to encourage a software culture that knows that small is beautiful, that actively resists bloat and complexity: an engineering tradition that puts a high value on simple solutions, that looks for ways to break program systems up into small cooperating pieces*[3]

There are occasions where more complex concepts and technologies may become necessary. This can easily hurt the simplicity principle through over-engineering.

To avoid this pitfall, the more complex technologies and concepts should always be built on top of the simpler ones. The more elementary parts should have no dependency to the more complex ones and should be sufficient for the simpler tasks.

This leads to a system, where the level of complexity can be adjusted/chosen to the level that is actually needed. It also enforces a more elegant architecture, forcing the system architect to abstract, simplify and modularize.

1.4.4 Reuse of existing standards and tools

Reusing and adjusting already established standards has a couple of benefits. For one, those standards have usually already been field-tested. Depending on the distribution of the software or standard, many developers are already familiar with it. This makes it more likely that they find and try your system.

Even if developers choose to learn those standards in order to use your system, they gain benefits by learning a standard that they can use and apply in different contexts.

[3] Eric Steven Raymond, *The Art of Unix Programming*, 40

2 Schema-Driven Development

Theory: This chapter introduces the concept and the term Schema-Driven Development (SDD). It will discuss how it relates to the concept of Model Driven Engineering (MDE).

I will also propose a modular SDD software architecture that lays the foundations for Chapter 3.

2.1 Introduction

There are already various terms for the type of approach I'm proposing in circulation. So why add another one? The first reason is that most of those terms already occupied with certain technologies and approaches. Second, some of these terms do apply, but are very broad and therefore rather nondescriptive.

To give this approach some distinction and theoretical backbone, I'd like to introduce a new term and concept: Schema-Driven Development.

I am not the inventor of this concept. There are already projects that use a schema-driven approach, like Swagger[4]. I couldn't find a distinctive theory or awareness of this approach, so here we go.

2.2 Schema-Driven Development

A definition of this approach could be:

> *Schema-Driven Development uses annotated data schemas, which specify the expected data structures, as models to generate system artifacts (code, documentation, tests, etc.) automatically.*

It is a greatly simplified and data-centric subset of Model-Driven Engineering.

SDD is not general purpose and thus not meant as a substitution for MDE in general or even ordinary programming languages. Instead, by adhering to the niche of schema-oriented use cases, it can stay simple and specialized. It should be therefore quick to learn and apply, especially compared to traditional MDE approaches. This turns it into an interesting candidate for use cases where those would be too difficult or expensive to implement.

Appropriate fields of application may be the (partial or complete) generation of APIs and various CRUD applications like CMS and KMS.

2.3 MD*

Before heading into the details on the Schema-Driven Development approach, I'd like to introduce the basic principles of Model-Driven Engineering (MDE) and the like (I'll subsume the various acronyms to MD*). I will also put them in relation to the proposed SDD approach. MDE is a broad topic on its own, so I will not go into much detail however. While familiarity with the subject is helpful, it is not required to understand the SDD approach.

[4] http://swagger.io/

There are numerous terms in use. Commonly used are Model-Driven Engineering (MDE) and Model-Driven Development (MDD). Moreover, there is the Model Driven Architecture[5] (MDA), which is a specific approach advocated by the Object Management Group (OMG) since 2001[6].

SDD is a subset of MDE and MDD. While SDD can be classified as a specific flavor of Model Driven approaches, most Model-Driven approaches are not Schema-Driven.

SDD does not meet the criteria to count as a MDA approach. MDA assumes the use of specific standardized technologies like MOF[7] and UML[8] and prescribes a certain architecture – which SDD does not. Similar to MDA, SDD is opinionated regarding its choice to use schema languages as modeling basis. Contrary to MDA, it doesn't prescribe the concrete technologies to be used.

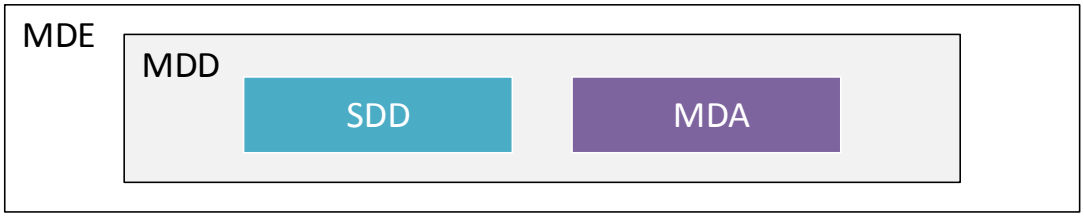

Figure 1: SDD in context to the various MD* acronyms

All the previously mentioned terms include Model-Driven. Let's review what this means:

The term Model (and therefore modeling) is very generic:

> *Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality.*[9]

[5] OMG, 'MDA Guide Version 1.0'

[6] Truyen, 'The Fast Guide to Model Driven Architecture', 2

[7] OMG, 'Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification'

[8] Cook, 'OMG Unified Modeling Language'

[9] Rothenberg, *The Nature of Modeling*, 1

The main purpose of using models is therefore to tame complexity by raising the level of abstraction. They can also be used to reduce the problem to certain, specialized viewpoints.

In this broad sense, even high-level programming languages are models of the resulting machine code. Programming in C is therefore a kind of model-driven development. When people are talking about MDE, they of course assume a higher abstraction level of the model than the one of current programming languages.

The approach is Model-**Driven**, because the model is not only used for conceptualization or illustration purposes[10].

> *It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification.*[11]

To sum it up:

> *Model-Driven Development (MDD) is a development paradigm that uses models as the primary artifact of the development process. Usually, in MDD the implementation is (semi)automatically generated from the models.*[12]

MDE technologies and standards have been researched, developed and standardized for a long time. With their standardization process and rising flexibility, they also got more complex, up to a point where it is difficult to weight the cost of learning those technologies to the benefits. The more intricate technologies often require additional (and often proprietary) tooling support in order to be used in a productive way.

Schema-Driven Development is a type of MDD that explicitly defines data schemas as the modeling basis. Because of that constraint, it can afford to be simpler.

> ***The goal of SDD is to reach many (or most) benefits of the MDE approach, while reducing the complexity of the approach itself to a reasonable small amount.***

Since schema languages are usually text based and relatively simple in nature, tooling support can become optional and a matter of taste.

---

[10] Stahl, *Modellgetriebene Softwareentwicklung*, 11

[11] OMG, 'MDA Guide Version 1.0'

[12] Brambilla, Cabot and Wimmer, 'Model-Driven Software Engineering in Practice', 9

2.4 Benefits of SDD

Why should developers use a schema-driven approach instead of the common and well-understood traditional software development? There are many benefits[13], which also apply to model-driven approaches in general.

Let me briefly introduce a few of them.

2.4.1 Simplification through Abstraction

The main reason for using models instead of implementation code is that they have a much higher level of abstraction. The models can therefore be simpler, more concise and focused. In case of schema languages, the models are solution oriented – they describe the expected or required data structure.

Because the building blocks are simpler and more generic, it is possible to reuse them easier. The model can therefore be very DRY (Don't repeat yourself)[14].

Complex systems can also become very overwhelming to the people who have to develop, maintain and therefore understand them. Taming that complexity reduces the mental demand of those tasks.

2.4.2 Less Error-prone

Because of the higher abstraction level, developing the model is simpler and requires fewer lines of code. Therefore, there is less to do wrong.

Likewise, it is easier for the computer to detect errors when the model can be analyzed for consistency according to some defined rules and introspection. By using a generator, there is an additional compilation step (from model to implementation code) where validation can happen.

2.4.3 Faster and more Agile Development

Because of the simpler model and better validation, the development speed can increase drastically. This makes the use of agile development methodologies much more feasible.

Of course, there is not only the effort of developing the model. If no generator software exists, it needs to be developed first or parallel to the model. The cost of this and the benefits of model-driven development has to be weighted. There are cases where the effort might not be worth it, especially when the use case is too complicated or specialized.

---

[13] Johan den Haan, '15 reasons why you should start using Model Driven Development'

[14] Hunt and Thomas, *The pragmatic programmer*

In the case that the project is very complex and hard to maintain with traditional development or a lot of similar software has to be developed repeatedly, MDE has the potential to save a lot of effort in the long term.

2.4.4 Reuse and Migration

Models and Data in general also often have a much longer life span than the software that created / uses it itself. If they are generic enough, they can be used and reused in many ways – even originally unintended ones.

There might be the case that the end system changes or might even be replaced. If the model is generic enough, the programmers could adjust the generator to the new requirements and the old model will work with the new system as it is.

This also means that systems developed through models can get improved features or performance just by updating the generator software and compiling it again.

2.4.5 Separation of Concerns

In MDE, the domain knowledge and the implementation logic/details can be separated. Most of the complicated implementation will go into the generator software itself. People with much less (or none at all) software development skills can develop the model, because much of the implementation complexity is outsourced to the generator.

The model itself can be separated to different viewpoints, each focusing on a different aspect of the system.

2.4.6 Code Quality

The generator works by definite rules on how to transform the model into the final code. Because of that, the resulting code will always be consistent. Improving the generator transformation logic does automatically improve the code quality of the complete result without having to refactor the model itself.

Updates of the generator could then introduce new features and improvements to the quality and performance of the code. In many cases, it is sufficient to recompile the old model with the updated generator and the improvements are applied.

2.5 Using a Schema Language as Model

2.5.1 The Schema Language

Obviously, a choice has to be made as to how the model should be written. This splits usually into two decisions: The notation format (syntax) and a given standard how to structure the content, the schema language.

In case of Schema-Driven Development, a few standardized formats specialize in describing expected data structures. The most commonly known are probably XML Schema[15] (XSD) that is notated in XML[16] and JSON Schema[17] that is notated in JSON[18].

Because data schemas are usually written in the same data format the schema describes, this leads to an interesting feature: They are capable of self-description. This implies that a data-schema can describe itself by using its own capabilities. The specification of the schema language can be written in the schema language[19].

With SDD, we choose to use data schemas to describe expected data structures within the implementation system. The Generator also needs to define the expected data structure of the model through a schema. It is referred to as the meta-schema, as it describes the schema itself.

If the schema language has a specification, it is the meta-meta-model. It describes how both models and meta-models need to be structured.

To sum it up: the model schema validates the resulting data in the end system; the meta-schema validates the model schema, and everything can be validated against the meta-meta-schema, since all of them are written in the same schema language.

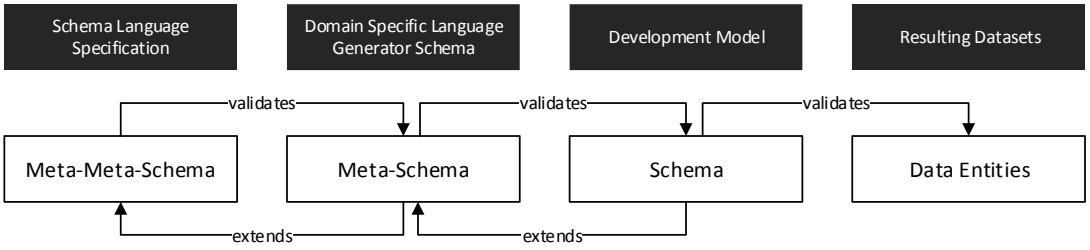

Figure 2: The schema hierarchy

Initially, this might sound complicated and abstract. In practice, it is a very elegant and simple way to use only one schema language for modeling purposes – on the model side as well as on the generator side, from bottom to top.

While the meta-meta-schema is already specified, the meta-schema most likely needs to be custom developed according to the given requirements and chosen target platforms.

[15] David C. Fallside, Priscilla Walmsley, 'XML Schema Part 0: Primer Second Edition'

[16] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, 'Extensible Markup Language (XML) 1.0 (Fifth Edition)'

[17] Francis Galiegue, Kris Zyp and Gary Court, 'JSON Schema'

[18] Douglas Crockford, 'The application/json Media Type for JavaScript Object Notation (JSON)'

[19] Example: http://json-schema.org/draft-04/schema

2.5.2 Domain Specific Language Development

The Meta-Schema is a Domain Specific Language (DSL). DSL are not general-purpose languages, but designed for specific use cases, describing certain domains.

> *A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.*[20]

In many cases, it is not sufficient to describe the only the data structure itself. The generator might need some more hints. It is also useful if some standard behavior could be adjusted or overwritten.

To do this, the data schema can be annotated with that information. I'd like to propose three main categories for classifying these according to their specificity.

There are also two internal categories which are specific only to the generator itself and therefore do not fall within the three other categories.

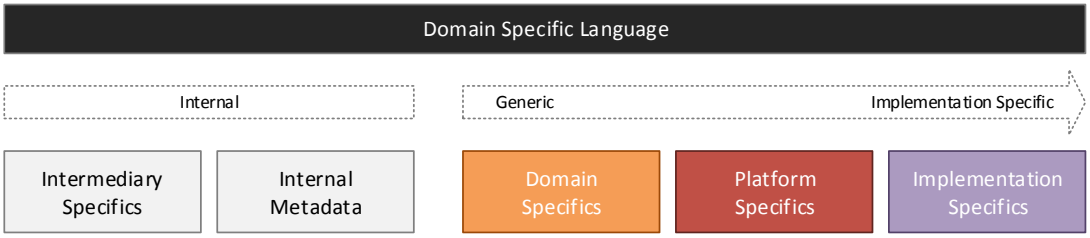

Figure 3: From domain specifics to implementation specifics

Domain specifics will therefore be transformed to platform specifics, increasing the proximity to the implementation system. Properties that are more specific will always override less specific properties, however.

I hope the term DSL is not confusing in this context, as it also includes platform- and implementation specifics. The DSL is the sum of all the proposed categories, because it serves as the container language to declare all those specifics.

2.5.3 Domain Specifics

Domain Specifics are independent from the technical implementation and describe the subject in terms domain experts use. Since SDD has data centric use cases, the chosen Schema Language already brings a vocabulary to describe data structures. They can be counted as domain specific and provide a free, domain specific base vocabulary to build upon.

---

[20] Arie van Deursen, Paul Klint, Joost Visser, 'Domain-Specific Languages - An Annotated Bibliography'

In MDA terms, the domain specifics (or domain model) is a mix between the Computational Independent Model (CIM)[21] and the Platform Independent Model (PIM)[22].

2.5.4 Platform Specifics

Platform specifics contain information that are specific to a certain target (software) platform. They directly translate into options, functions or concepts of the platform. Using platform specifics therefore delegate the duty of defining an API from the DSL to the target platform.

> *A platform model provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform.*[23]

2.5.5 Implementation Specifics

The Implementation Specifics already use the target language and need not to be transformed by the generator anymore. They can be utilized to overwrite or add code directly. This is a two edged sword, however:

On the one hand, this enables the model to be completely flexible, since everything that cannot be modeled can be added or overwritten in the implementation language.

On the other hand, everything that is implementation specific is outside the code generator and therefore cannot be inspected, validated or optimized on the same level as the rest of the model.

> ***I recommend always using the least specific way necessary to achieve a modeling goal in order to maximize the benefits of the model-driven approach.***

2.5.6 Intermediary Specifics

In the case, that any dynamic functionality needs to be added, intermediary-specific attributes must be introduced. The intermediary layer applies the logic, which the intermediary specific attributes define, to the model. After they are processed, the helper attributes can (or should) be removed.

[21] OMG, 'MDA Guide Version 1.0', 15

[22] OMG, 'MDA Guide Version 1.0', 16

[23] OMG, 'MDA Guide Version 1.0', 16

Intermediary specifics make it possible to support dynamic and advanced features, which the chosen schema language does not support by itself. Because the intermediary layer removes those features after they are applied, the resulting model is 100% compatible with the schema language specification again.

If an intermediary layer is given, it is therefore possible to introduce features to the development model that are not compliant with the schema language specifications but still helpful for the development process. This is of course a design decision whether to break that compliance or not.

Paragraph 2.6.3.3 will further explain the intermediary layer and gives some examples of possible features.

2.5.7 Internal Metadata

Internal metadata have no impact on the generated implementation code. They are strictly behind the scenes and hidden to the user. Possible applications are internal helper variables, debugging or statistics.

2.5.8 Separation of the Specifics

In MDA, the model itself is divided into four categories. This might be helpful with big teams with distributed responsibilities. However, this also implies that more models need to be maintained and kept consistent to each other.

A simpler approach is to prefix the attributes according to their specific role and declare in the Meta-Schema to which category they belong. One model can then contain all those information at the same time.

If a custom editor tool is utilized, it could then still limit the viewpoints according to roles and permissions.

2.5.9 Side Note: Semantics

In some cases, annotated data schemas may not be expressive enough. In those cases, the SDD approach might not be the best way to go.

In my project, I did not have to go beyond schemas. In the case, that more advanced expressiveness is necessary, I've thought about adding an additional "semantic" layer on top of the schema layer.

This way, the more basic requirements can be achieved through the much simpler schema languages and the more complex problem can be solved with semantic languages, like description logics. This would conform to the principle of stacked complexity. In case of JSON Schema, this would mean to embed JSON-LD to express additional semantics, while the hard validation still happens on the schema level.

2.6 Architecture of the Generator

2.6.1 Overview

The generator is the software that loads the model, transforms it and outputs the final implementation code.

Based on my experiences with building a SDD generator, I'd like to propose the following architecture shown in Figure 4. It is a flexible proposal, containing several optional components (drawn with dotted lines).

It is not necessary to implement the full architecture to get started. In fact, mobo started with a few proposed layers missing, which were added later as the project grew.

In the simplest implementation, the development model is identical to the expanded model. No model-to-model transformation layers are used. The development model is directly transformed into the implementation code. In this case, the generator is not much more than a template engine or XSLT Transformations[24].

[24] Henry Zongaro, Andrew Coleman, C. M. Sperberg-McQueen, 'XSLT and XQuery Serialization 3.0'

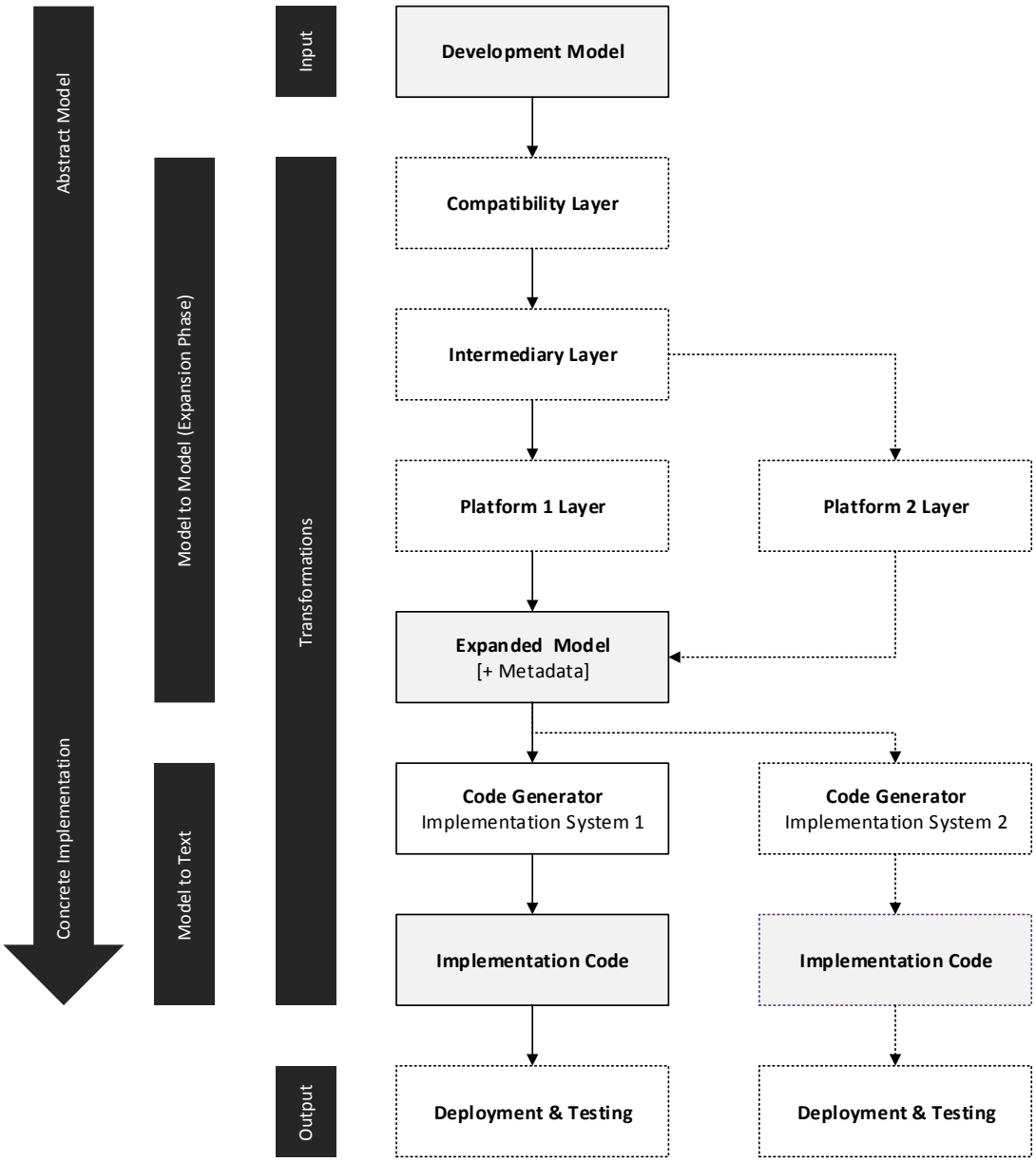

Figure 4: Proposed modular architecture of the generator

2.6.2 Input

The generator reads the development model from the file system. Afterwards, it uses a parser (depending on the notation format) to convert the schema documents into an adequate internal data structure (depending on the programming language of the generator).

Note that it is possible to support different notation formats by providing multiple parsers, as long as the structure of the schema document stays consistent.

2.6.3 Model to Model Transformations

2.6.3.1 Model Expansion

The development model should be as abstract, concise and non-redundant as possible to be convenient to use and manage. The generator itself has different requirements, though. First, verbosity and duplicated information are not an issue when they are programmatically introduced. However, they can be very convenient from a programming perspective, especially when accessing data.

For example, implicit properties from the model can be explicitly created through some clearly defined rules. That way, the user doesn't have to write them (and keep them maintained) and the generator can still depend on their existence and has not to infer them repeatedly.

Therefore, in this stage the input model is repeatedly transformed to yet another, expanded (usually less abstract and more verbose) model. This stage can be called the model expansion phase and the resulting model the expanded model. The JSON LD standard[25] uses a similar concept, which they call the expanded document form.

There are languages that are specifically designed for this job like QVT[26] from the OMG. It is also possible to use common programming languages for those transformations – in the simplest case the same language the generator is written in.

2.6.3.2 Compatibility Layer

The first proposed step is to add an optional compatibility layer that transforms outdated and deprecated features of older models back to the latest standard. This becomes useful or even necessary, when new features need to be added to the generator without breaking older models immediately.

The compatibility layer can also notice the user about the changes they have to make in order to be compliant with the latest standard.

2.6.3.3 Intermediary Layer

The optional intermediary layer introduces helper functions that make writing the development model easier and more dynamic. After the intermediary specific annotations are applied, they can be removed. The expanded model (after the transformation) is then completely compatible with the original schema specification.

Features that rely on dynamic logic would make an intermediary layer necessary:

[25] Manu Sporny et al, 'JSON-LD 1.0'

[26] OMG, 'Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification'

The intermediary layer could for example implement custom inheritance and return the expanded model (where the inheritance has already been applied) as the transformation result. This provides a good example where the development model can be non-redundant while the expanded model contains a lot of duplicated and verbose information.

Other interesting features would be internationalization, dynamic code injections, templating capabilities in the development model itself, etc.

2.6.3.4 Platform Layer

The generator may include any number of platform layers, depending on how many platforms are supported. This layer will transform the generic domain specifics to platform specific information.

This is an optional preparation step to pre-optimize the model for the code generator. Those transformations could also be done in the code generator itself, but if they can be implemented as model-to-model instead of model-to-text transformations, it makes more sense to put them in this layer.

2.6.3.5 Expanded Model

The result of the model-to-model transformation phase is the expanded model. It should be 100% compliant to the schema language chosen and contain additional metadata. It is machine-optimized whereas the development model is human-optimized.

2.6.4 Model to Text Transformation

2.6.4.1 Code Generator

The expanded model is the foundation that the following model-to-text transformation uses to generate implementation code.

The generator can support one or more code generators – one for each platform that needs to be supported.

There are many ways code can be generated from models. It is of course possible to write the logic in an ordinary programming language, just concatenate the resulting code and write it to a big text file.

It is also possible to use template engines that specialize in this task. Template engines take a text file, written in the desired target language and inject template tags for custom logic and inserting variables. They are usually (and intentionally) limited in how much logic and complexity they allow within a template, however.

Personally, I recommend a hybrid approach. More advanced template engines can be extended by writing new custom functions. Those are programmed in the same language the template engine itself is written in. Application-specific custom functions therefore improve the modularity and flexibility of the template engine.

This way, templates can use the standard functionality of the template engine where it is sufficient and custom functions where not. Custom function are also a good choice when the default capabilities become too complicated and inconvenient.

2.6.4.2 Implementation Code

The resulting code is the implementation code.

As a last step, the generator could now apply coding guidelines and styles by using a code beautifier and similar libraries.

2.6.5 Output

Depending on the requirements, the generator can output the implementation code as text files on the file system and/or handle the deployment by itself.

If the generator handles the deployment, it can generate/update the end-system in real-time, allowing for an agile, prototyping workflow. It would also allow running automated tests against the deployed end-system in case that this feature is a part of the generator.

2.7 Tooling

2.7.1 Validation

Validation of the model is a very useful feature that helps improving the quality of the end system and the development speed, as errors can be found much faster. The generator can provide Validation on many different levels: Syntax (1), Structure (2) and Semantics (3).

The input stage of the generator will use parsers to read the development model. Nearly all parser libraries already come with error feedback in case of syntax errors.

Since the meta-schema is written in the same schema language as the model, the development model can be validated against the meta-schema. This will detect structural errors.

It is also possible to implement custom logic that also looks for semantic errors, where the model may be syntactically and structurally correct but makes no sense from a domain perspective.

The resulting implementation code can also be validated using linters, compilers or (auto generated or already existing) unit-tests.

2.7.2 Model Inspection Tools

The generator might come with some tools that help users to understand and inspect the state of the model. This could include viewing the model in its various stages (development model, expanded model and implementation code), a visual representation, general documentation, etc.

2.7.3 Model Development Tools

The fact that data-schemas are usually text based, allows the straightforward use of version control systems like Git. It also means that the development model can be written with common text editors and IDE's. Specialized tooling can therefore become unnecessary or at least optional.

It is a matter of preference whether to use visual tools or text based tools. Some tools help with creating data-schemas in a visual way, like Altova XMLSpy[27] or JSON buddy[28].

In some cases, it might also be a good idea to write a custom editor that is explicitly optimized for use with the developed generator.

2.7.4 Helper Utilities

The generator can use the information from the development model for more than just generating the implementation system.

There could be utilities like programmatic data imports and synchronization. Random test data[29], based on the model schema, can be automatically generated. The model can also be used to validate or assess the quality of the existing data in the end system. There are a lot more possibilities.

2.8 Knowledge Requirements

2.8.1 From Generator Developer Perspective

The developer of the SDD system needs to choose and learn at least one data notation format. Those are usually very simple and most developers know a few of them

[27] http://www.altova.com/xmlspy.html

[28] http://www.json-buddy.com/

[29] Example: Random data from annotated JSON Schema: http://schematic-ipsum.herokuapp.com/

anyway. A decision has also to be made which schema language to use. Those vary in complexity - JSON Schema is arguably simpler than XML Schema.

The generator can be written in a programming of choice.

The Domain Specific Language (DSL), in our case the meta-schema, has to be developed. Since the DSL is written in the schema language itself, it makes considerable sense to develop a documentation generator that generates the API Docs automatically.

Some optional technologies can be used if they seem appropriate: Template Engines, Transformation Languages etc.

2.8.2 From User Perspective

The user of the SDD software (a domain expert, or a technical platform expert) does not need to understand the Generator and the inner workings of it. He has to understand the following technologies and concepts:

If no specialized editor (tooling) is available, it is mandatory to be write syntactically correct models. Fluency with the chosen data notation format is therefore the most basic requirement. It is also important to understand the structure and concepts of the chosen data schema language.

The Domain Specific Language and its features must be learned, too. It is vital to have a decent documentation, since it is specific only to the generator itself. Documentation about those features cannot be found elsewhere.

Depending on how deep the users goes into the technical implementation, understanding of the target platform it is helpful to mandatory.

3 Generating Semantic MediaWiki Structure with SDD in practice

Praxis: Using the Schema-Driven Development approach to develop and generate the structure of Semantic MediaWikis. An extended, object oriented JSON Schema is used as the schema language.

3.1 Introduction

This chapter puts the proposed theory and architecture of Schema-Driven Development into practice.

My SDD generator implementation has the name mobo[30]. It is written in JavaScript, runs as a CLI application and can be easily downloaded and updated as a NPM package[31].

Mobo is cross-platform and open source[32].

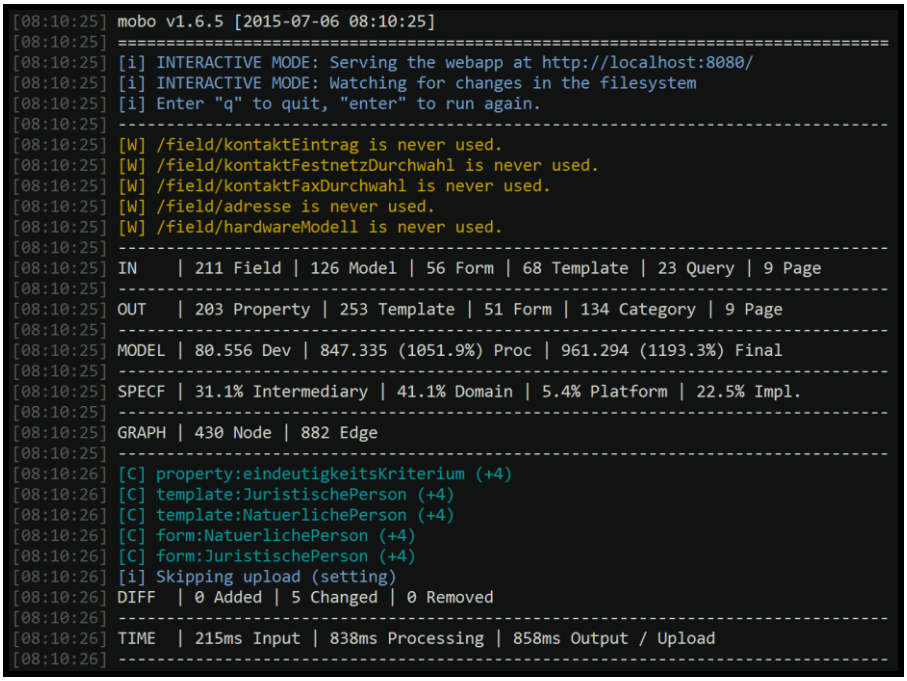

Figure 5: Screenshot of the mobo CLI application

The user documentation, including tutorial and auto-generated DSL documentation is available at GitBook[33].

3.2 The Target Platform

3.2.1 Introduction

The target system is a Knowledge Management System (KMS). Mobo targets the features of some extensions in addition to the base system itself.

[30] Simon Heimler, *mobo*

[31] https://www.npmjs.com/package/mobo

[32] https://github.com/Fannon/mobo

[33] Simon Heimler, 'mobo documentation'

Allow me to give a brief introduction on the most important parts and aspects of our target system. For a more comprehensive introduction, I recommend reading the book "Working with MediaWiki"[34].

3.2.2 MediaWiki

The chosen base system is MediaWiki[35] (MW). It is a free, open-source KMS that follows the wiki approach. MW is widespread, well established and has a broad community of users and developers. The system has already proven its stability and scalability by powering Wikipedia.org.

3.2.3 Semantic MediaWiki

MediaWiki itself has only rudimentary features to store and retrieve structured information; its core strength lies in the management of unstructured text. Some structure can be achieved through the definition and use of templates, but they will only be stored in plain text, nevertheless.

This is where Semantic MediaWiki[36] (SMW) comes in. SMW adds the capability to store and query structured data within MW. It provides a notation that allows declaring facts, either within templates or within free wikitext. That information is stored in a flexible, graph-oriented structure. SMW comes with a simple (but limited) query language, called ASK which allows to retrieve and reuse those facts.

SMW makes use of Semantic Web Technologies. For example, it can additionally use a Triplestore together with the W3C standardized query language SPARQL[37]. This provides powerful querying and even reasoning capabilities, accessible through a RESTful API.

3.2.4 Semantic Forms

While SMW introduces structured storage of information, it can only be entered by writing wikitext markup. To ensure a better user experience and enforce a desired structure of the information, the Semantic Forms[38] extension (SF) allows defining custom web forms.

Those forms support basic validation, auto completion depending on existing data and various input widgets.

---

[34] Koren, *Working with MediaWiki*

[35] Wikipedia, 'MediaWiki'

[36] http://semantic-mediawiki.org/

[37] Andy Seaborne and Steven Harris, 'SPARQL 1.1 Query Language'

[38] https://www.mediawiki.org/wiki/Extension:Semantic_Forms

3.2.5 Challenges and Peculiarities

3.2.5.1 Wikitext as target language

MediaWiki uses its own markup language, wikitext. Wikitext markup can be extended through custom functions, templates, magic words, etc. A good example is the ParserFunctions[39] extension, which adds a lot of programmatic functionality, like if-statements. Wikitext is Turing complete.[40]

The implementation language of the generator is the MW specific wikitext markup language, which is therefore also a Domain Specific Language (DSL). The generator is therefore transforming one DSL into another one.

Wikitext has a few notable characteristics that influenced the implementation of the generator:

Wikitext is always valid. Therefore, it cannot be validated. This can cause a few problems, as "broken" markup results in broken layout or functionality, but is technically still valid. This characteristic makes errors hard to spot and avoid.

Since wikitext misses validation capabilities, the ability of the generator to validate the development model is a big benefit over using wikitext directly.

Wikitext is whitespace sensitive. The resulting wikitext pages and templates can therefore become very difficult to write and read, because inserting whitespaces for readability might break the markup. The development model doesn't have this problem; it even allows for inline comments. The generated wikitext code may be ugly, but even the handwritten wikitext often *needs* to be.

Wikitext supports no inheritance. The true origin of this problem lies in the fact that wikitext does not support data as a native concept. Templates can be used to declare information once and reuse it somewhere else. However, since those templates are text with mainly text transforming (and nesting) capabilities, they do not support more advanced data specific features like inheritance.

When designing and implementing the knowledge management structure, this becomes a limitation, leading to duplicated wikitext. This also makes the model much harder to maintain and keeping it consistent.

[39] https://www.mediawiki.org/wiki/Extension:ParserFunctions

[40] Jared, 'Wikimedia Proves Greenspun's Tenth Law'

3.2.5.2 Targeting an existing Platform

Targeting an existing platform has advantages; the wheel does not have to be reinvented. Existing platforms have already been used in the field, tested and found useful in real use cases. They also bring extensions, their community and documentation with them, which should not be underestimated.

However, targeting an existing system limits the generator, as it can only support features that the target platform supports. The target platform will also influence the design of the Meta-Model, the DSL. In some cases, this saves a lot of work because someone has already put a lot of thought into it. In some cases, it means that inconsistencies or design problems of the implementation platform may leak back into the DSL.

For instance, the Semantic Forms extension does not support real-time HTML5 form validation. The mobo Schema contains enough information to take advantage of it, but since the SF does not, it is currently unsupported. To implement this feature, it would need to be implemented in Semantic Forms first.

3.3 Schema Language Development

3.3.1 Schema Language

3.3.1.1 JSON Schema

Mobo uses JSON Schema[41] as schema language. It was the simplest standard (the specification is only a few pages long) that I've found that still could do the job.

JSON Schema is currently an IETF draft[42] in version 4. There is work going on, creating a version 5 draft. This means that JSON Schema is still in development. The version 4 standard has a broad collection of libraries and programming language support[43].

3.3.1.2 Benefits of JSON Schema

JSON Schemas biggest strength is that it is simple and extendable. The simplicity of JSON Schema makes the model easy to read and develop with nothing but a text editor. It also makes the development of the generator easier, because the standard is easy to use and extend from a developer perspective, too.

[41] Francis Galiegue, Kris Zyp and Gary Court, 'JSON Schema'

[42] http://tools.ietf.org/html/draft-zyp-json-schema-04

[43] http://json-schema.org/implementations.html

Many libraries use and support JSON Schema. Mobo uses a few of them, which definitely saved some development effort. E.g. for internal schema validation, the tv4 library[44] is used.

3.3.1.3 Limitations of JSON Schema

There were two major issues with using JSON Schema for SDD purposes, which had to be resolved.

No defined Inheritance Behavior.

The official `$ref` keyword can be used to reuse or import internal and external parts of JSON Schema. In the (unsupported) case that the $ref keyword is on the same level with other properties, it is unclear how to resolve the resulting conflict. Some implementations, like the tv4 validator library[45], apply inheritance in this case, while some do not.

Therefore, I've decided to introduce a custom inheritance capability through three new keywords: `$extend`, `$remove` and `$abstract`.

The order of properties

From modeling and data structure perspective, it makes the most sense to use the property notation to define sub-elements within the model. They are stored as maps (JavaScript objects) and therefore easy to access.

There is one serious problem with that choice from the code generator perspective. While for validation and data storage purposes the order of those properties is irrelevant, the generator depends on the given order. The order of the input fields within a form must be definite. While JavaScript engines are usually generous and won't mix up the order of object properties (until a certain size of the object is reached), the official standard does not require them to:

> *An object is a member of the type Object. It is an unordered collection of properties each of which contains a primitive value, object, or function.*[46]

Since mobo stores each part of the model in its own file, the ID is already declared through the filename. Using properties would duplicate this information by having to type them as key names again. This is not only inconvenient, but potentially introduces inconsistencies when the key name is not identical to the filename.

---

[44] https://github.com/geraintluff/tv4

[45] https://github.com/geraintluff/tv4

[46] Ecma, 'Standard ECMA-262 3rd Edition'

I went with the compromise that the development model uses the items notation, using an (ordered) array structure. Mobo will then internally convert it back to the more convenient property notation and store the order of the items as metadata in a separate array.

Code Example:

```
# Item Notation
type: array
items:
  - $extend: /field/fieldOne
  - $extend: /field/fieldTwo
```

```
# Is internally converted to:
type: object
$itemsOrder: ['fieldOne', 'fieldTwo']
properties:
    fieldOne:
        $extend: /field/fieldOne
    fieldTwo:
        $extend: /field/fieldTwo
```

3.3.2 Notation Format

Mobo supports both YAML[47] and JSON[48] as notation format.

Internally it only uses a YAML Parser[49] that is used for both YAML and JSON files. This is possible because YAML is a superset of JSON. Valid JSON is therefore valid YAML, too. The YAML parser has the additional benefit, that it returns more detailed syntax errors than the default `JSON.parse()` implementation.

I highly recommend using the YAML notation to write the development model. It is more concise and more convenient to write for humans. The JSON notation does not even support comments, which are very convenient to have in the development model. That's because JSON is primarily a machine optimized serialization format.

[47] Oren Ben-Kiki, Clark Evans and Ingy döt Net, 'YAML Ain't Markup Language (YAML™) Version 1.2'

[48] Douglas Crockford, 'The application/json Media Type for JavaScript Object Notation (JSON)'

[49] https://github.com/nodeca/js-yaml

3.3.3 Developing the Meta-Schema (DSL)

3.3.3.1 Overall structure

While it is possible to use only a few extensive schema files to describe a system, using many smaller files has a few advantages. First, it makes reusing them easier and more straightforward. Smaller files are also easier to comprehend and manage.

In the case of modeling SMW/SF structure, it makes sense to come up with a foundational structure that roughly resemble the nature of the end system. The target system uses SMW Properties, MW Templates, MediaWiki Categories and Semantic Forms to define and implement the Knowledge Structure.

I chose to split the mobo development model into three main categories: Mobo forms, mobo models and mobo fields. Mobo forms can also directly refer to implementation specific MW Templates.

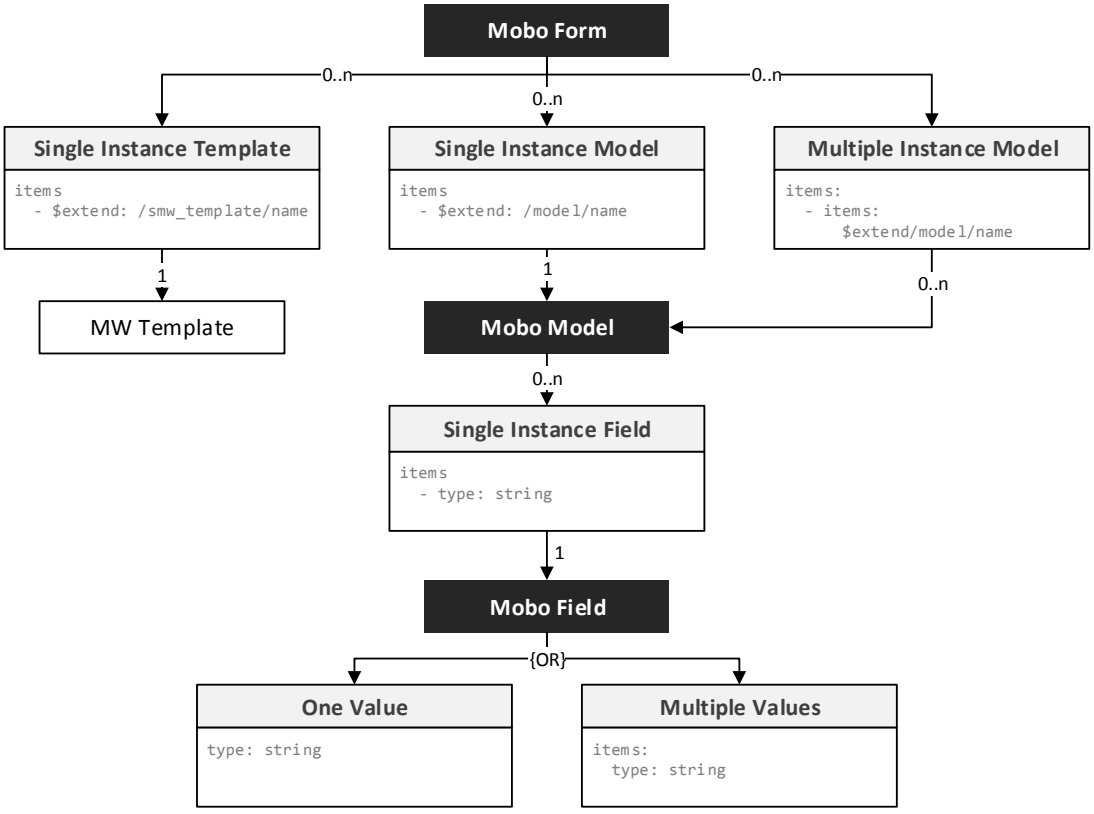

Figure 6: The mobos development model structure

Because mobo supports object-oriented inheritance in the development model, the mobo model structure can be more elegant as the platform structure, which is lacking this capability.

Figure 6 summarizes this structure. Mobo forms can define any number of single- and multiple instance models. (This reflects SF capability of using single- and multiple instance templates). A model defines any number of fields it uses. Fields can be either single- or multiple value: Single value fields represent input fields that hold exactly one value; multiple values fields can hold any number of values, e.g. a field with comma-separated values.

In contrast, Semantic Forms define within itself which widget to use for each SMW Property. In the case, that a field is used by more than one Semantic Form, this information has to be duplicated. In the mobo structure, this information is declared in the mobo field and inherited all the way up to the mobo form, with the possibility to overwrite it at any point.

3.3.3.2 Structure of the Mobo Schema

I chose to use JSON Schema as the schema language of mobo. The specification of JSON Schema is available as a self-describing JSON Schema file[50], so it can be directly used as meta-meta-schema.

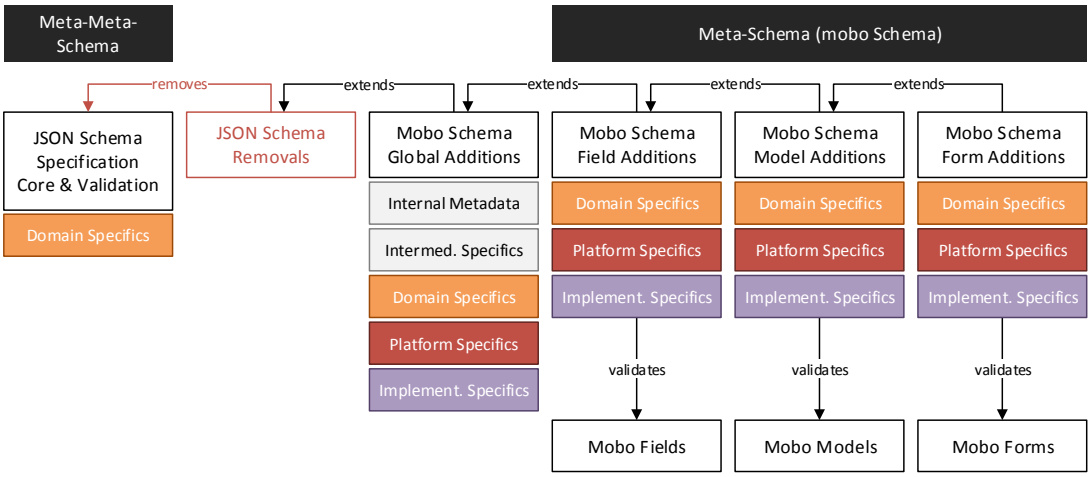

Figure 7: Structure of the mobo Schema

I'll call the meta-schema of mobo the mobo Schema. It starts by being a duplicate of the meta-meta-schema, the JSON Schema specification. All unsupported or unnecessary features of JSON Schema are removed afterwards (JSON Schema Removals).

The JSON Schema specification itself is divided into a core and a validation specification. The latter includes many properties that can be utilized to describe and validate data structures. Mobo Schema includes the validation schema, which is used as a base domain specific vocabulary.

[50] http://json-schema.org/draft-04/schema

Since Mobo does not support all of JSON Schemas validation features (because Semantic Forms does not), some of them are removed. The use of `$ref` and `definition` is substituted by the custom inheritance logic that is implemented through the intermediary system.

As Figure 7 implies, the mobo Schema is split into a mobo field schema, mobo model schema and a mobo form schema. The model schema includes all of the field schema and the form schema the entire model schema. This is necessary, because a model contains fields and may overwrite field specific properties on the model level.

Specifics

The mobo Schema defines the domain specific, platform specific and implementation specific properties. They are all notated together in the same file. The properties are prefixed depending on their specificity and the mobo Schema internally defines their role.

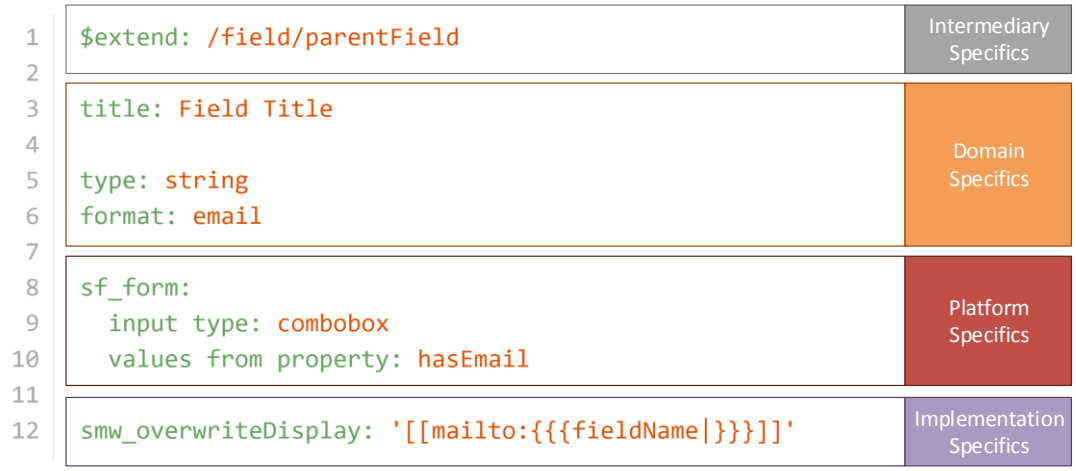

Figure 8: Mobo Schema with indicated specifics

Intermediary Specifics

The intermediary property `$extend` triggers the custom inheritance logic in the intermediary system. Intermediary specifics are prefixed with a dollar sign.

Domain Specifics

The `title`, `type` and `format` properties already come with the JSON Schema validation specification and describe information in a domain specific way. The mobo Schema additionally introduces a few own domain specific properties. They have no prefix.

Platform Specifics

Platform specific information directly resembles and translates into the options of the target platform. E.g., the `sf_form` property is platform specific, as it hints though

its prefix. It directly targets the SF (Semantic Forms) platform. In this case, the options are described in the official Semantic Forms documentation[51]. Because the options are only forwarded, mobo does not have to support them on its own.

Using those platform specific properties has the disadvantage of them being dependent on the target platform. They cannot easily be translated to a different platform. Since mobo only supports one target platform, this is not much of a problem.

The benefit of this approach is that new or changed features of the target platform are immediately available to use. Mobo does not need to be updated in order to support them. This also saves mobo a lot of DSL development effort.

Implementation Specifics

Implementation specific attributes like `smw_overwriteDisplay` share their prefix with the platform specifics, since both target a specific platform. They can be distinguished by the fact that implementation specific properties directly use wikitext as values.

Paragraph 3.4.6 will further explain the example of Figure 8 by illustrating how it is transformed through the individual layers.

Documentation Generation

Mobo can automatically generate the technical documentation of the mobo schema. It is rendered in markdown[52] format, which then is included to the official GitBook[53] documentation though imports.

To keep the documentation clear, only those properties are rendered that are natively specific to the model part.

[51] https://www.mediawiki.org/wiki/Extension:Semantic_Forms/Defining_forms#.27field.27_tag

[52] The Daring Fireball Company LLC, 'Markdown'

[53] https://www.gitbook.com

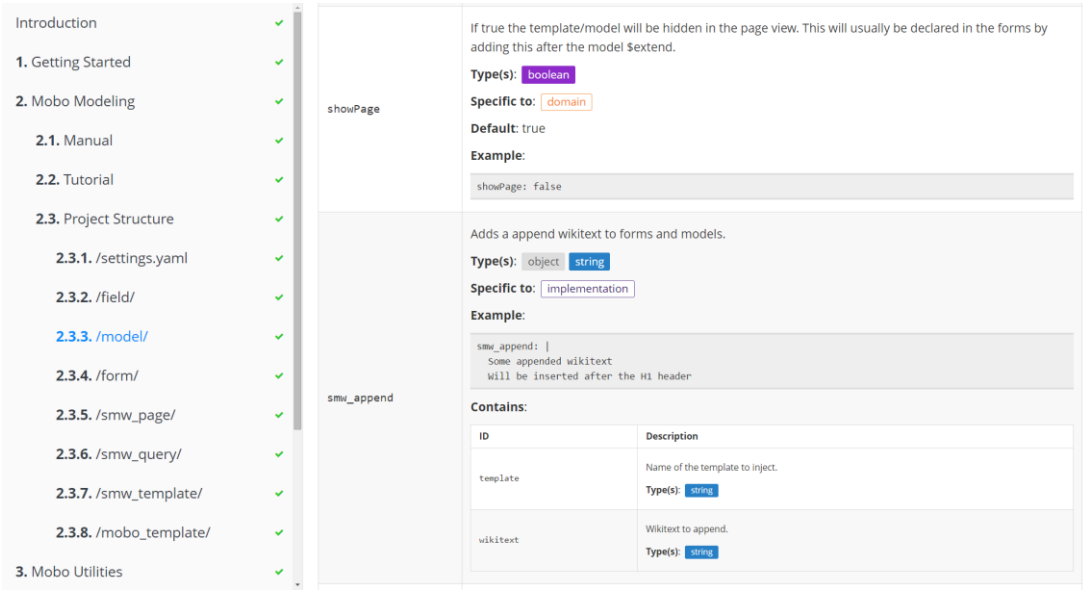

Figure 9: Auto-generated mobo Schema documentation

3.3.4 Custom inheritance

An important feature of mobo is the custom inheritance logic. It does not only help reusing the code, but also helps to establish the actual structure of the model.

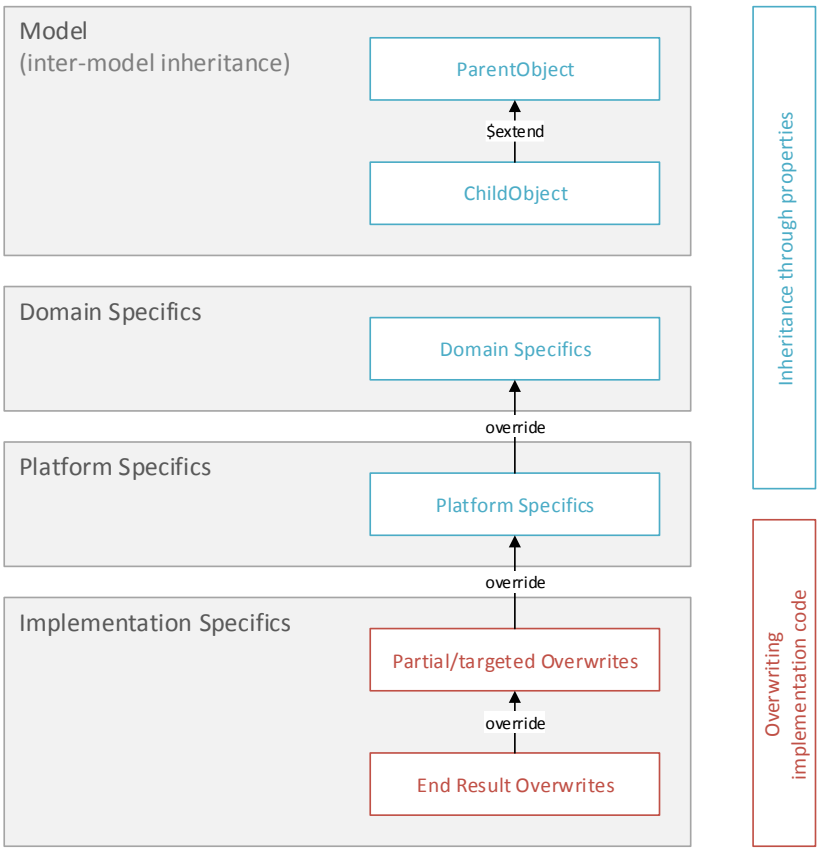

Figure 10: Model inheritance behavior

A child object uses `$extend` to declare its parent and will inherit all properties of the parent. Properties of the child object will always overwrite properties of the parent object.

In the case that both share the same property, the merging behavior is the following:

- Primitive properties (String, Number, Boolean) will be overwritten.
- Objects will be merged. If the object has identical properties, the merging logic will be recursively re-executed at this point.
- Arrays are handled accordingly to the annotations they contain. For example, an array containing `@prepend` and `@unique`, will first be merged so that the child properties come first. Then all duplicate items will be removed. For all options, please refer to the mobo user manual[54].

Figure 10 also illustrates how properties within the model are resolved. More implementation specific properties always override the less specific properties.

3.4 Architecture of the Generator

3.4.1 Overview

The current implementation of mobo uses all of those proposed architectural parts (see Figure 4), but does not implement all possible aspects of it.

In fact, the architecture was developed parallel to mobo. In some cases, I have inferred the architecture from the actual practice and in other cases, designing the architecture led to refactoring mobo. For those historical reasons, the actual architecture of mobo may not always be as ideal as the proposed architecture.

3.4.2 Input

Mobo stores each element of the model in its own file and organizes the structure through folders. The first level of the folder structure separates the different model areas, like mobo forms, mobo models, mobo fields, MW templates, etc. This is a mandatory structure that mobo will generate when initializing a new project.

The directory structure within those folders will be flattened and ignored. The files can therefore be organized and moved freely without having to adjust the exact paths. The names of the files must be unique anyway.

Mobo will read those files, parse them and store the resulting data structure in the internal registry object.

[54] Simon Heimler, 'mobo documentation'

3.4.3 Model to Model Transformations

3.4.3.1 Introduction

Mobo uses JavaScript for model-to-model transformations and no specialized transformation languages. The lodash utility library[55] proved to be very helpful for data analysis and transformation tasks.

3.4.3.2 Compatibility Layer

The compatibility layer became necessary when I decided to refactor some parts of the mobo Schema. I had already developed a rather big model that was already in productive use.

In order not to break my current model, I had to introduce some transformations that upgrade legacy properties or schema structures to the latest standard.

As a side effect, the compatibility layer made a sort of test-driven development workflow possible. When new features are introduced or existing ones changed, I could implement them in the compatibility layer first. The generator will then fail on its task to generate the model until the introduced features are developed and integrated.

The generator also keeps a diff to the last uploaded state of the target system. This can be used to detect involuntary changes to the implementation code. In the case, that a refactoring must not change the result, I could develop and fix the newly introduced feature until the diff was identical again.

The compatibility layer also tells the user, which parts of the model have to be upgraded, and give a short hint how to do it.

3.4.3.3 Intermediary Layer

The intermediary layer subsequently adds object-oriented inheritance capability to JSON Schema. Currently, it has no other functionality.

Please note, that I have already introduced the inheritance feature in paragraph 3.3.4 and explained the intermediary layer on a generic level.

Internally, mobo makes heavy use of the lodash library, especially `_.merge`[56] for merging objects (providing custom logic, which handles arrays) and `_.cloneDeep`[57] to ensure that objects aren't accidentally mutated in the process.

---

[55] https://lodash.com/ and the book Boduch, *Lo-Dash essentials*

[56] https://lodash.com/docs#merge

[57] https://lodash.com/docs#cloneDeep

The inheritance layer checks for circular dependencies and gives a warning if in case one is detected. Additional metadata is added that helps analyzing the model, e.g. for tree shaking to detect unused parts of the model.

3.4.3.4 Platform Layer

At the time of writing, some platform and implementation logic is still implemented in the code generator itself; mostly for historical reasons. Refactoring them out would definitely improve the architecture of mobo and probably save some lines of code, too.

It would also make sense to create a small independent library with helper methods to turn data structures into wikitext, e.g. template calls, functions, tables. This library could be then be used in the platform layer, the template engine (through a small wrapper) and other, unrelated projects.

3.4.4 Model to Text Transformation

3.4.4.1 Code Generator

The code generator currently makes use of the handlebars.js template engine[58] with some custom functions added. Handlebars is a rather simple template. It does not support some more advanced features like template inheritance.

The fact that the template engine uses some of the same control characters (for opening and closing tags/functions) as wikitext makes the use of escaping very necessary.

A useful feature of handlebar.js is the use of the circumflex character to strip white spaces before and after handlebars expressions. This allows introducing some white space to make the templates more readable while the resulting wikitext can be the whitespace-less clutter it sometimes needs to be.

Example

```
{{#each template~}}


{{~#if this.prepend}}{{{this.prepend}}}{{/if~}}
```

3.4.5 Output

Mobo can handle the deployment to the target wiki itself or simply write the resulting wikitext files on the file system.

By default, mobo runs in an interactive mode. It detects changes on the local file system and automatically (re)triggers the process. Depending on the settings, which

[58] http://handlebarsjs.com/

can specify which steps mobo should automatically execute, this leads to the following workflow steps:

1. Validation of the development model
2. Compilation of the development model to the final wikitext
3. Display which of the resulting wiki pages have changed since the last successful upload to the end system
4. Automatic and optional upload of all wikipages or (by default) only the pages that have been affected.
5. Writing optional statistics and log files. Uploads an optional report to the wiki.

Mobo currently does not support unit-tests or end-to end tests, so the deployment does not include automated testing.

3.4.6 Transformation Example

To get a broader picture of the overall transformation process, I'd like to get back to the example code in Figure 8.

For sake of better readability and formatting, I have added white spaces and line breaks. The example is simplified and reduced, to illustrate the relevant aspects of the transformation layers.

Development Model:

The example is written in the YAML notation. It contains intermediary, domain, platform and implementation specific properties.

```
$extend: /field/parentField
title: Field Title
type: string
format: email
smw_form:
  input type: combobox
  values from property: hasEmail
smw_overwriteDisplay: '[[mailto:{{{fieldName|}}}]]'
```

Input Stage

In the input stage, the development model is parsed to the internal data structure of the target language. In case of JavaScript, it can be expressed through the JSON notation.

```
{
  "$extend": "/field/parentField",
  "$path": "C:/the/path/to/someFieldId.yaml",
```

```
  "title": "Field Title",
  "type": "string",
  "format": "email",
  "smw_form": {
    "input type": "combobox",
    "values from property": "hasEmail"
  },
  "smw_overwriteDisplay": "[[mailto:{{{fieldName|}}}]]"
}
```

Compatibility Layer

The compatibility layer detects deprecated parts of the development model. In this example, the `smw_form` property name will be renamed to `sf_form`.

```
{
  "$extend": "/field/parentField",
  "$path": "C:/the/path/to/someFieldId.yaml",
  "title": "Field Title",
  "type": "string",
  "format": "email",
  "sf_form": {
    "input type": "combobox",
    "values from property": "hasEmail"
  },
  "smw_overwriteDisplay": "[[mailto:{{{fieldName|}}}]]"
}
```

The compatibility layer will also notice the user that it has done so.

```
[i] Renaming deprecated properties "smw_form" to "sf_form" (1)
```

After the Intermediary Layer

A new `sf_form` option, `existing values only`, is inherited. All other properties are overwritten by the children itself.

After the inheritance is applied, the `$extend` property is removed.

```
{
  "$path": "C:/the/path/to/someFieldId.yaml",
  "title": "Field Title",
  "type": "string",
  "format": "email",
  "sf_form": {
```

```
    "input type": "combobox",
    "values from property": "hasEmail",
    "existing values only": true
  },
  "smw_overwriteDisplay": "[[mailto:{{{fieldName|}}}]]"
}
```

After the Platform Layer

The platform layer infers the platform specific `smw_type` property from the domain specific type and format properties.

```
{
  "$path": "C:/the/path/to/someFieldId.yaml",
  "title": "Field Title",
  "type": "string",
  "format": "email",
  "smw_type": "Email",
  "sf_form": {
    "input type": "combobox",
    "values from property": "hasEmail",
    "existing values only": true
  },
  "smw_overwriteDisplay": "[[mailto:{{{fieldName|}}}]]"
}
```

After the Code Generator

The code generator produces the final wikitext pages. Please note that the mobo structure is not directly translated into the wikitext structure. In this example, a mobo field will result in one dedicated wikipage and parts of at least two (depending on the inheritance) more wikipages.

Property:SomeFieldId

For each field, a SMW Property page is created, declaring the internal SMW datatype.

```
<noinclude>
<div class="mobo-generated">This page is autogenerated, do not edit it
manually!</div>
[[Category:mobo-generated]]
</noinclude>
* This is an attribute of the datatype [[Has type::Email]].
```

Template:SomeModelId

The mobo field will also transform into a part of a MW Template. The template defines how the field is rendered in the page view mode.

```
[...]
{{#if: {{{someFieldId|}}} |
<div class="row">
    <div class="col-sm-4 col-md-3 row-label">Field Title</div>
    <div class="col-sm-8 col-md-9 row-value" data-property="someFieldId">
      {{{ someFieldId |}}}
    </div>
</div> |}}
[...]
```

Form:SomeFormId

The SF platform specific field information end up as part of a SF Semantic Form and define how the field is rendered in the form view mode.

```
[...]
<div class="sfFieldContent col-sm-8 col-md-9">
{{{field
|someFieldId|class=attr_someFieldId
|input type=combobox|max values=1|existing values only
|values from property=hasEmail
}}}
</div>
[...]
```

3.5 Tooling

3.5.1 Validation

Since wikitext can't be validated, mobos internal validation becomes an even more important aspect of the generator. The validation is realized on a syntax, structure and semantic level.

If mobo is run in interactive mode, it will provide validation feedback nearly immediately after a file of the development model is saved.

3.5.2 Model Inspection Tools

When run in interactive mode, mobo serves a web application, the mobo inspector. It allows viewing the development model, expanded model and the implementation

system code (the resulting wikitext). In the case that the final wikitext code differs from the last upload state, a visual DIFF is displayed.

The mobo inspector also comes with automatic documentation generation, using the docson[59] library. It displays exemplary forms through the jsoneditor[60] library, though this is mostly to demonstrate the ability to generate forms from JSON Schema on a more abstract level. It is not a replacement for viewing the forms on the end system.

Figure 11: The mobo inspector

Mobo can generate an interactive, graph-based visualization of the development model. To use this feature, the graph mobo generates needs to be prepared by applying a graph layout algorithm first, however. This can be done with Gephi[61], a graph analysis and visualization software.

[59] https://github.com/lbovet/docson

[60] https://github.com/jdorn/json-editor

[61] http://gephi.github.io/

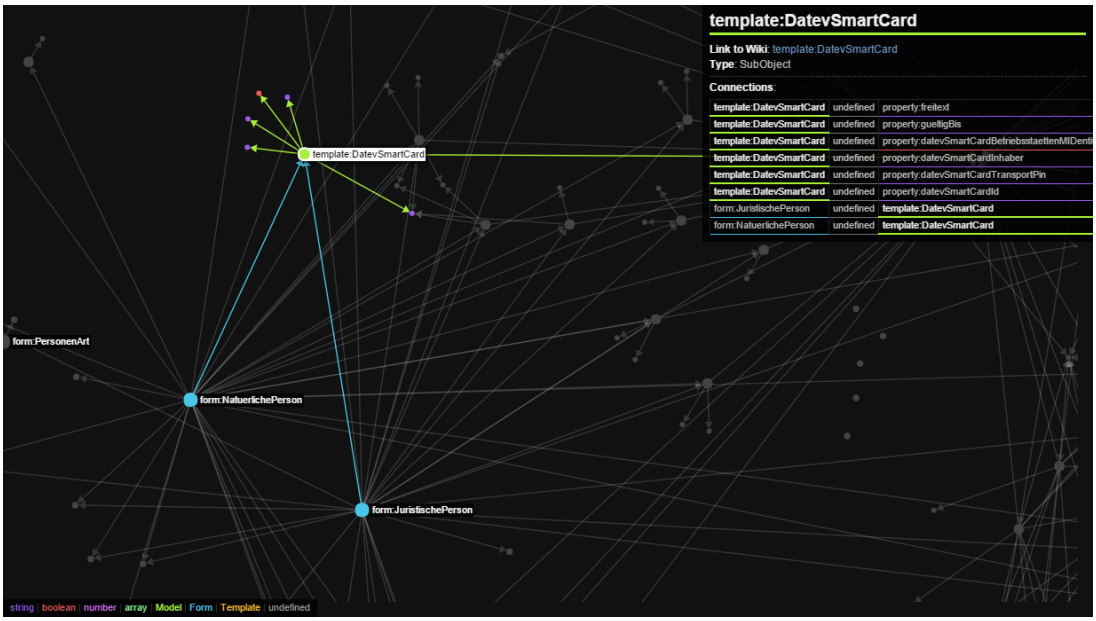

Figure 12: The interactive graph explorer, visualizing the development model

3.5.3 Model Development Tools

Mobo does not come with custom model development tools. JSON Schema, especially when using the YAML notation, is easy to write with standard text editors.

Personally, I prefer a text editor to visual tools when the markup is reasonable simple. I've made good experiences using SublimeText[62] and the Atom[63] editor. I recommend using a YAML linter for real-time syntax error feedback and the fuzzy file finder for quickly navigating through the development model project.

Using only a text editor resulted in a very fast and convenient model development workflow for me personally.

3.5.4 Helper Utilities

Mobo features a few helper utilities. The actual documentation of those utilities can be found in the mobo user manual[64].

Nuker

The nuker module allows to batch-delete all wikipages of certain namespaces. This comes in handy when prototyping on a development wiki system.

[62] http://www.sublimetext.com/

[63] https://atom.io/

[64] Simon Heimler, 'mobo documentation'

Importer

Mobo also comes with an importer module. It can batch-import static wikipages or use a programmatic import. The programmatic importer comes with several helper functions, e.g. validating imported data against the development model, automatically enhancing incomplete information. It also comes with a few helper functions that allow creating wikipages easier.

The importer module can also run in an interval, continually importing new data from external sources.

Statistics and Reports

When generating the model, mobo can calculate some statistics and append them to a CSV file. This helps keeping track of the development model history.

Mobo can also upload reports of its activity and an annotated tree outline of the generated model.

3.6 Use case: Modeling IT Management Knowledge

3.6.1 Introduction

I have developed mobo along a real modeling use case of Computer Bauer GmbH. Their requirements were rather special in some cases, so we had to realize a custom KMS structure.

The main subject of the model was IT hardware- and software management, with a tightly integrated CRM.

3.6.2 Using SDD in Practice

Once mobo was developed, we switched from a conceptual and rather theoretic model development to an agile, rapid prototyping workflow.

We met once or twice a week as a group and discussed new features. Minor features or changes could be developed and deployed in a matter of seconds/minutes. We could therefore implement, test and evaluate those features in very short sprints. Reviewing the result in an actual working end-system with the same functionality as the final system was very helpful in this process.

It proved possible to implement even medium sized features that resulted in minor structural changes in less than an hour. This is less enough effort, to encourage experimenting with ideas and evaluate them not just on a theoretical basis. Since the model is versioned, it is simple to roll back those changes again - there is no risk involved.

The low implementation costs and low risk allowed (and encouraged!) us to try ideas and features we otherwise might have omitted.

Bigger changes and refactoring phases I've implemented between the meetings.

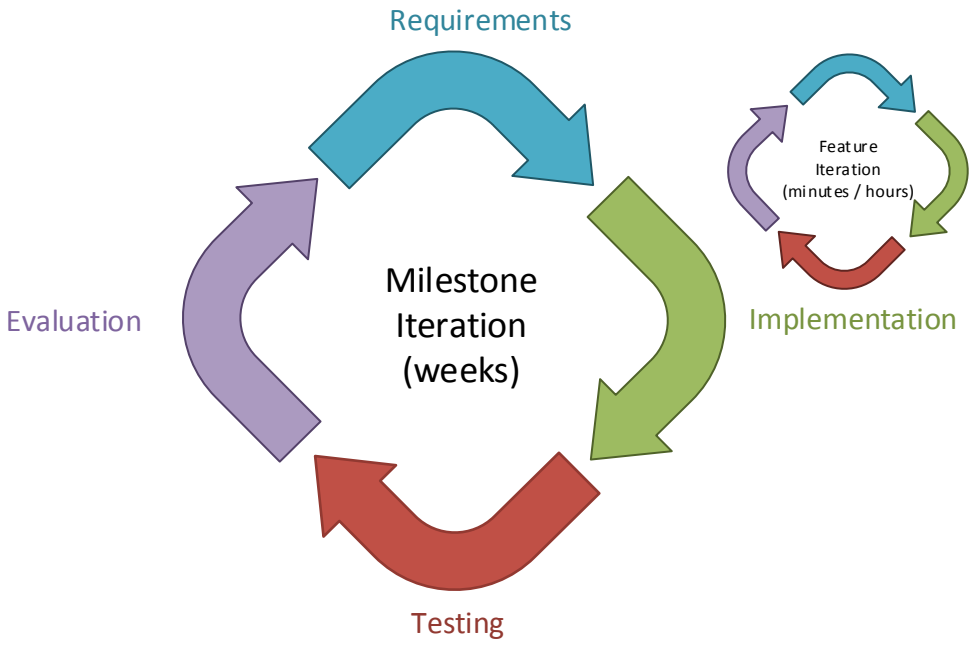

Figure 13: Agile/iterative development workflow

When bigger milestones were reached, we conducted user testing with end users, testing actual use cases from the company. The feedback went back into the next iteration. Minor changes we usually implemented at once.

Overall, we were very pleased with the development workflow that mobo made possible. Given the complexity of our project, using a model/schema driven workflow may have even become a necessity for the success of this project.

Once the KMS was officially introduced, we had only a few problems to solve in the live system, since it was already well tested.

Of course, most of the total development time went into mobo itself. It is difficult to judge whether overall development time has been saved. However, the parallel development of the model and the generator allowed shifting much of the hard work and biggest time investments to the generator. The development of the generator is therefore a preparation, as it can be done in advance or between the meetings.

In the long run, the development effort of the generator will increasingly pay off as it can be reused for other projects.

The actual model development usually requires the knowledge and involvement of many peoples. The SDD approach saves them a lot of time or makes this sort of teamwork feasible in the first place.

3.6.3 Evaluation of the Development Model

The development model consists of 211 mobo fields, 126 mobo models, 56 mobo forms, 68 templates, 23 queries and 9 pages. The development model is 80.556 characters (~78kB) big.

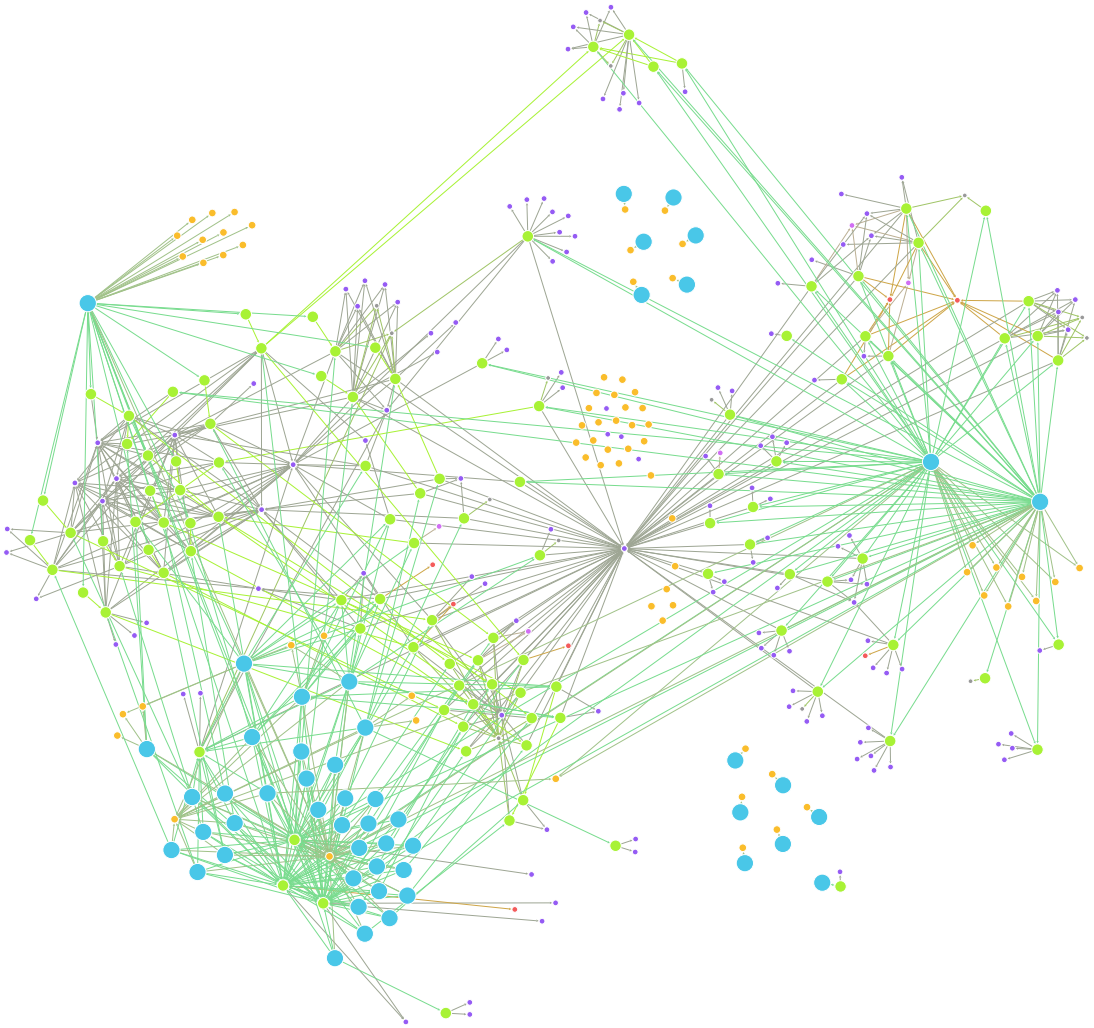

Figure 14: A force-layouted graph visualization of the development model

Those numbers does not include imported pages. We imported a few dozen static pages and a few thousands wikipages that were programmatically generated, based on data from external databases.

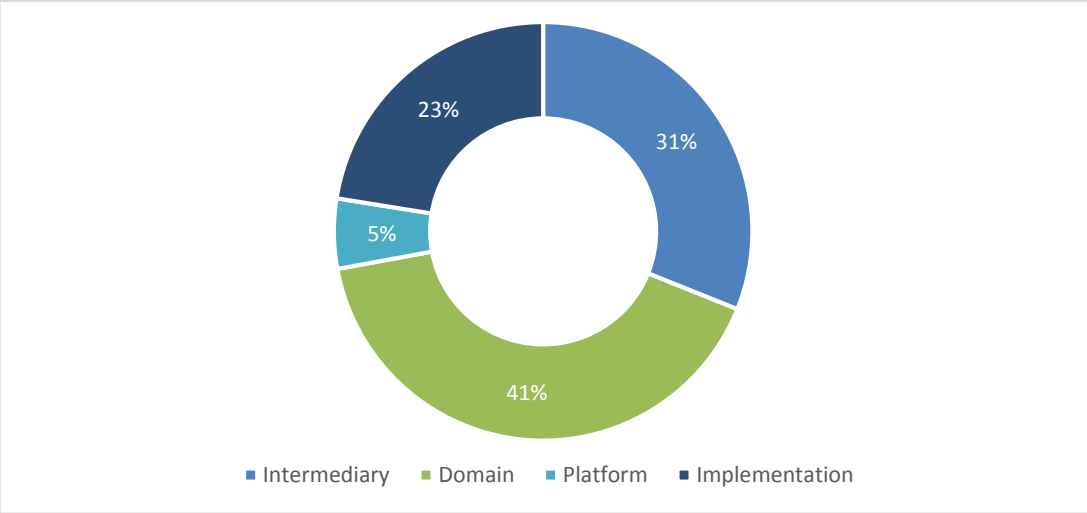

Figure 15: The development model, split into the specifics

The development model consists of 31% intermediary specific properties, 41% domain specific, 5% platform specific and 23% implementation specific.

The intermediary specific properties should be considered as domain specific, since they define the actual structure of the model itself.

The high number of implementation specifics stems from the fact some parts of the model necessarily have to be written in wikitext, e.g. queries. In the case of our model, some implementation specific properties result in rather selective and small overwrites. In most cases, the implementation code is prepended or appended and therefore an addition to the generated code. There was no need to overwrite generated wikipages as a whole in any case.

Since mobo keeps historical track of the statistics, it is possible to view and analyze the development process. Let me give you a short, interpreted example:

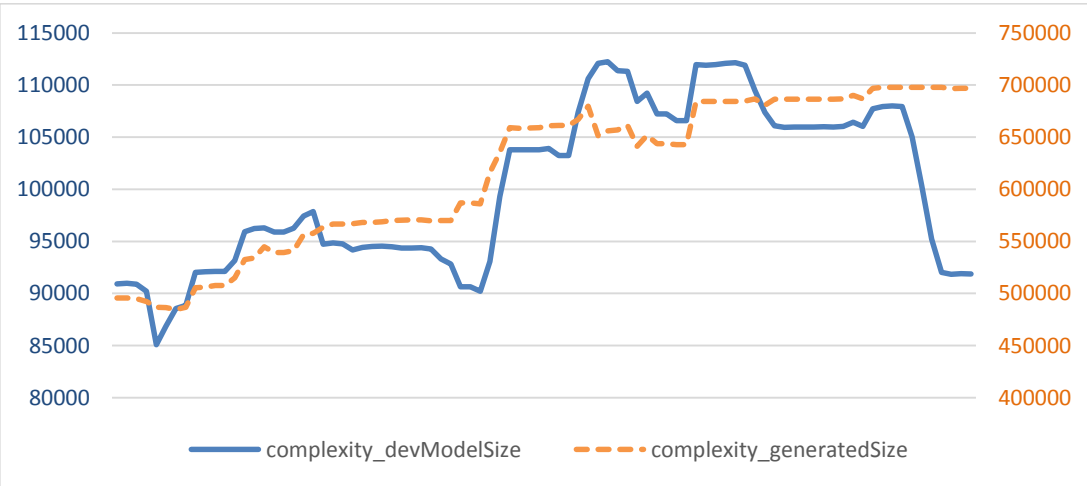

Figure 16: The size of the development model and the implementation code over time.

The diagram indicates the progression of the development model size and the implementation code size (dashed line). Please note that the lines have different axis scales for visual reasons; the important aspect is their correlation.

The implementation code size usually grows, except when actual features are removed. The development model size is a lot more dynamic. This is the result of our rapid-prototyping approach and indicating the cycle between fast model development and the subsequent refactoring/cleanup phase. At our meetings, we developed new features very fast. When those features were accepted, I put some refactoring effort into them like improving code reuse and reducing the technical debt that comes with rapid-prototyping development.

This refactoring phase is especially obvious at the end of the chart, where the development model size decreases notably, while the size of the implementation code stays identical.

3.6.4 Evaluation of the Implementation Code

The resulting implementation consists of 203 SMW properties, 254 MW templates, 51 SF forms, 134 MW categories and 9 MW pages. It is 961.294 characters (~ 939kb) big.

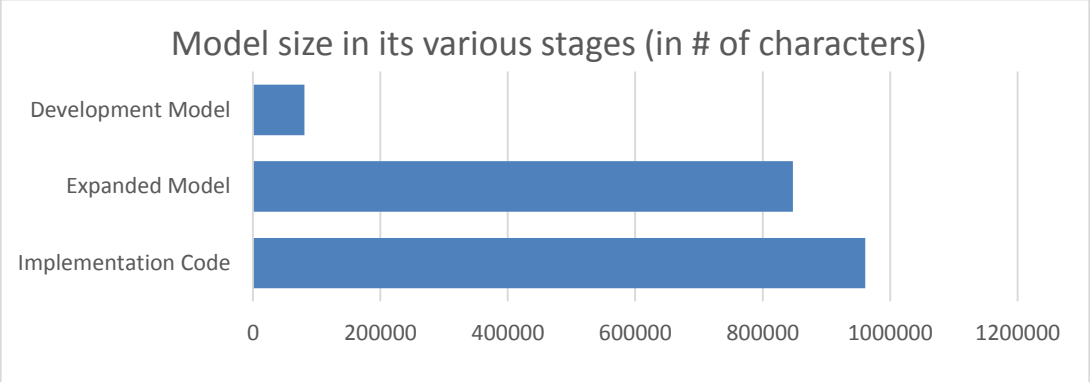

Figure 17: The model size in its various stages; measured in number of characters (x-axis)

For every character in the development model, there are nearly 12 characters of generated implementation code.

I do not argue that this results in 12 times more productivity. It depends on what part of the model is worked on. In cases where a lot of inheritance is involved, changes that take only a few seconds in a mobo could result in up to an hour of refactoring and a lot of risk to introduce inconsistencies while doing so. Other changes (on queries for example) take practically identical time.

3.7 Outlook

3.7.1 Mobo

While realizing this project, I have collected a few possible features and improvements for future releases of mobo.

Internationalization

The intermediary layer could add internationalization to the model. The user could then write custom language definitions and define in the settings.yaml which language to use.

The internationalization could also happen on the wiki level, in case of multi lingual MediaWiki installations.

A more flexible template engine

As hinted at in the paragraph 3.4.4.1 about the code generator, the current template engine is rather simple. The fact that no template inheritance/extension is supported results in loss of flexibility.

When a new mobo version introduces breaking-changes to the mobo templates, the current development model has to update its templates. In the case, that they have been adapted, those adaptions must be made anew. This could problem could be partially solved, by modularizing the template render modes to individual files and integrate/overwrite the default mobo templates.

I am currently considering two templates engines that might be better suited for the job: Swig[65] and nunjucks[66].

Multiple Models in one Project

An improvement on the modularity of model development could be made by the introduction of modular models. This would make mobo more comfortable to use when deploying existing, modular model-parts.

In this case, a mobo project would consist of only one model, but a collection of models and one central settings file that declares the order and importance of those model-modules. In the case, that adjustments have to be made, a module containing all customizations can be added.

Splitting the Project into multiple modules makes the model development more manageable (modules are smaller and contained) and reusable.

[65] http://paularmstrong.github.io/swig/

[66] https://github.com/mozilla/nunjucks

Automated Testing

End-to-end testing would be a useful addition. Mobo already knows how the final forms should look like and behave, so it could generate those tests at least semi-automatically.

Of course, this would mostly test the correctness of the generator (mobo) and the correct functionality of the end system. However, in cases where wikitext is injected directly (and this is hardly to avoid), it would be very useful to test that the custom wikitext does not break the final generated wikitext.

Quality Bot

This idea came up early in our project. Mobo could be used as a bot that reads and rates the quality and validity of the stored information. This can't be achived with the current platform, for several reasons. First, Semantic Forms has only elementary validation capabilities. Second, there are validation issues with the entered information that depends on the context of already existing data. Third, some information may be technically valid but unreasonable. There is also the problem that while some information is not required to provide, their existence does contribute greatly to the quality of the knowledge base.

To assess those softer quality criteria and more difficult validation rules, it would be useful to have a bot that scans the wiki for those issues. This makes the bot somewhat similar to a reasoner[67]. After extracting and reasoning about the quality of the information available, it could rate the content together with a report explaining the quality assessment and store it on the affected wikipage. The criteria how to rate the quality could be part of the development model.

A quality bot would make it faster and easier to spot information quality issues in the knowledge base.

MediaWiki already has a strong tradition of using bots to aggregate and rate content.[68] Building a quality bot on top of a SDD project would be especially convenient. The bot can build upon the already existing schema structure.

3.7.2 SDD

For our particular use case, the SDD approach worked well. It proved to be easy enough to develop the DSL and the generator, so that the benefits that come with SDD did outweigh the time investments to develop the system.

[67] Wikipedia, 'Semantic reasoner - Wikipedia, the free encyclopedia'

[68] https://en.wikipedia.org/wiki/Wikipedia:Bots/Status#Active_bots

It is important to address the issue of accidental complexity[69] that comes with creating an intermediary system and DSL. I would argue that in our case, it kept relatively low, as the SDD approach requires very few technologies to learn and much of the domain specifics of the schema language can be reused.

The use of platform and implementation specific properties reduces not only the complexity of designing the DSL itself, it also allows to put existing knowledge of the end system directly into use. The direct use of platform specific properties that translate directly to platform options is probably the biggest contributor to simplifying the DSL.

It is hard to measure accidental complexity, but in the case of mobo, my personal assessment is that it is reasonably low. After all, reducing the complexity of the approach itself is one of the primary goals of SDD.

[69] France and Rumpe, 'Does model driven engineering tame complexity?'

References

ANDY SEABORNE, and STEVEN HARRIS, 'SPARQL 1.1 Query Language', 2013 <http://www.w3.org/TR/2013/REC-sparql11-query-20130321/>, updated 2013, accessed 14 Oct 2013.

ARIE VAN DEURSEN, PAUL KLINT, JOOST VISSER, 'Domain-Specific Languages - An Annotated Bibliography'.

BODUCH, ADAM, *Lo-Dash essentials: Implement fast, lean, and readable code effectively with Lo-Dash* (Community Experience Distilled, Birmingham, England: Packt Publishing, 2015).

BRAMBILLA, MARCO, CABOT, JORDI, and WIMMER, MANUEL, 'Model-Driven Software Engineering in Practice', *Synthesis Lectures on Software Engineering*, 1/1 (2012), 1–182.

COOK, STEVE, 'OMG Unified Modeling Language. Version 2.5', 2013 <http://www.omg.org/spec/UML/2.5/Beta2/>, updated 5 Sep 2013, accessed 20 Mar 2015.

DAVID C. FALLSIDE, PRISCILLA WALMSLEY, 'XML Schema Part 0: Primer Second Edition', 2004 <http://www.w3.org/TR/xmlschema-0/>, updated 28 Oct 2004, accessed 25 Oct 2014.

DOUGLAS CROCKFORD, 'The application/json Media Type for JavaScript Object Notation (JSON)', 2006 <https://www.ietf.org/rfc/rfc4627.txt>, updated 27 Jul 2006, accessed 28 Mar 2015.

ECMA, 'Standard ECMA-262 3rd Edition. ECMAScript Language Specification', 1999.

ERIC STEVEN RAYMOND, *The Art of Unix Programming*.

FRANCE, ROBERT, and RUMPE, BERNHARD, 'Does model driven engineering tame complexity?', *Softw Syst Model*, 6/1 (2007), 1–2.

FRANCIS GALIEGUE, KRIS ZYP, and GARY COURT, 'JSON Schema. core definitions and terminology', 2015 <http://json-schema.org/latest/json-schema-core.html>, updated 19 Feb 2015, accessed 13 Mar 2015.

HENRY ZONGARO, ANDREW COLEMAN, C. M. SPERBERG-MCQUEEN, 'XSLT and XQuery Serialization 3.0', 2014 <http://www.w3.org/TR/2014/REC-xslt-xquery-serialization-30-20140408/>, updated 7 Apr 2014, accessed 3 Jul 2015.

HUNT, ANDREW, and THOMAS, DAVID, *The pragmatic programmer: From journeyman to master* (Reading, Mass.: Addison-Wesley, 2000).

JARED, 'Wikimedia Proves Greenspun's Tenth Law', 2006 <https://web.archive.org/web/20130101055126/http://mentalpolyphonics.com/posts/wikimedia-proves-greenspuns-tenth-law>, updated 11 Aug 2006, accessed 30 Jun 2015.

JOHAN DEN HAAN, '15 reasons why you should start using Model Driven Development', 2009 <http://www.theenterprisearchitect.eu/blog/2009/11/25/15-reasons-why-you-should-start-using-model-driven-development/>, updated 25 Sep 2009, accessed 1 Jul 2015.

KOREN, YARON, *Working with MediaWiki* ([San Francisco]: WikiWorks Press, 2012).

M. D. MCILROY, E. N. PINSON, B. A. TAGUE, 'Unix Time-Sharing System Forward', *The Bell System Technical Journal*, 1978, accessed 15 Jul 2015.

MANU SPORNY ET AL, 'JSON-LD 1.0', 2014 <http://www.w3.org/TR/json-ld/>, updated 16 Jan 2014, accessed 24 Oct 2014.

OMG, 'MDA Guide Version 1.0', 2003.

——, 'Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification', 2011 <http://www.omg.org/spec/QVT/1.1/PDF/>, updated January 2011, accessed 30 Jun 2015.

OREN BEN-KIKI, CLARK EVANS, and INGY DÖT NET, 'YAML Ain't Markup Language (YAML™) Version 1.2. 3rd Edition, Patched at 2009-10-01', 2009 <http://www.yaml.org/spec/1.2/spec.html>, updated 1 Oct 2009, accessed 8 Apr 2015.

ROTHENBERG, J., *The Nature of Modeling* (Rand, 1989) <https://books.google.de/books?id=wc7qAAAAMAAJ>.

SIMON HEIMLER, *mobo* (2014) <https://github.com/Fannon/mobo>, accessed 3 Jul 2015.

——, 'mobo documentation', 2015 <https://www.gitbook.com/book/fannon/mobo-documentation/details>, updated 8 Jul 2015, accessed 8 Jul 2015.

STAHL, THOMAS, *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management* (2., aktualisierte und erw. Aufl., Heidelberg: Dpunkt-Verl., 2007).

THE DARING FIREBALL COMPANY LLC, 'Markdown', 2004 <http://daringfireball.net/projects/markdown/>, updated 1 Dec 2004, accessed 16 Jul 2015.

TIM BRAY, JEAN PAOLI, C. M. SPERBERG-MCQUEEN, 'Extensible Markup Language (XML) 1.0 (Fifth Edition)', 2008 <http://www.w3.org/TR/REC-xml/>, updated 26 Nov 2008, accessed 29 Jun 2015.

TRUYEN, FRANK, 'The Fast Guide to Model Driven Architecture'.

WIKIPEDIA, 'MediaWiki', 2013 <https://de.wikipedia.org/w/index.php?oldid=123483893>, updated 18 Oct 2013, accessed 18 Oct 2013.

——, 'Semantic reasoner - Wikipedia, the free encyclopedia', 2015 <https://en.wikipedia.org/w/index.php?oldid=658967672>, updated 26 Jun 2015, accessed 6 Jul 2015.