

Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

mit Unterstützung der t2informatik GmbH

Model-driven Development of Front-Ends for CRUD Applications

Alexander Korzec

Matrikelnummer: 4915459

alexander.korzec@fu-berlin.de

Betreuer/in: Prof. Dr. Lutz Prechelt

Eingereicht bei: Prof. Dr. Lutz Prechelt

Berlin, August 8, 2022

Abstract

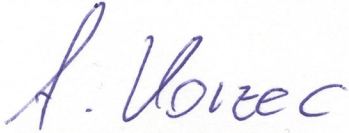
Create, Read, Update, Delete (CRUD) operations are a common part of UIs. Thus, the question arises whether there is a method that generates such predictable UIs without programming, but instead using other artifacts such as models. Model-driven Software Development (MDSD) and related approaches provide an answer to this question and establish the necessary conditions to use models as substitute for programming code.

This thesis gives an introduction to MDSD and summarizes requirements towards it based on an application example from a clinical management system. A MDSD method based on model interpretation is derived from these requirements and evaluated. As first step, the MDSD method uses a Low Code Platform to model the application flow and the appearance of CRUD UIs. Then, a standalone UI library that uses JSON Forms to display forms and JSON:API for data transmission interprets the artifacts created in the previous step.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

August 8, 2022



Alexander Korzec

Contents

1	Introduction	9
1.1	Motivation	9
1.2	Outline of Contribution	9
1.3	Challenges in Model-Driven Development	10
1.4	Structure of This Thesis	11
2	Model-driven Software Development	12
2.1	Overview of Model-driven Software Development	12
2.2	Model-Driven Architecture	15
2.3	Code Generation vs. Model Interpretation	16
3	Related Works	18
3.1	MockupToME	18
3.2	JSON Schema	19
3.3	JSON:API	19
3.4	JSON Forms	20
3.5	Eclipse Modeling Framework	21
3.6	Interaction Flow Modeling Language (IFML)	21
3.7	IFMLEdit.org	22
3.8	Webratio Platform	22
4	Analysis	25
4.1	Assumptions	25
4.2	User Group	25
4.3	Problems	26
4.4	Pseudo-Requirements	27
4.5	Application Example	27
5	Implementation	30
5.1	General Workflow	30
5.2	Low-Code Modelling	30
5.2.1	Description	30
5.2.2	Evaluation	35

5.3	UI Execution Environment	35
5.3.1	Input Specification	36
5.3.2	Implementation Details	36
5.3.3	Evaluation	38
6	Conclusion	39
A	Appendix	45

List Of Abbreviations

3GL	Third Generation Programming Language
API	Application Programming Interface
BNF	Backus–Naur form
CIM	Computation Independent Model
CORBA	Common Object Request Broker Architecture
CRUD	Create, Read, Update, Delete
DSL	Domain-specific Language
DSM	Domain-specific Modeling
EMF	Eclipse Modelling Framework
EMP	Eclipse Modelling Project
ERM	Entity Relationship Model
GPL	General Purpose Language
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IFML	Interaction Flow Modeling Language
J2EE	Java™ 2 Platform Enterprise Edition
J2SE	Java™ 2 Platform Standard Edition
JS	JavaScript
JSON	JavaScript Object Notation
LCP	Low Code Platform
M2M	Model-to-Model
M2T	Model-to-Text
MDA	Model-driven Architecture
MDD	Model-driven Development

MDE Model-driven Engineering
MDEE Model-driven Engineering Environment
MDSD Model-driven Software Development
MDWE Model-driven Web Engineering
MOF Meta Object Facility
MOFM2T MOF Model to Text
MVC Model View Controller
OCL Object Constraint Language
OMG Object Management Group
PIM Platform Independent Model
PoC Proof of Concept
PSM Platform Specific Model
QVT Query View Transformation
SPOs Software Producing Organisations
SQL Structured Query Language
TS TypeScript
UI User Interface
UML Unified Modeling Language
VHDL VHSIC Hardware Description Language
WYSIWYG What You See Is What You Get
XMI XML Metadata Interchange
XML Extensible Markup Language

1 Introduction

Chapter Introduction describes the goal and scope of this thesis. The first section gives an overview of how models and abstraction are used in software development. This is followed by the presentation of the goal of this thesis with the help of an example. After that, the generally valid challenges of this approach are highlighted. The last section outlines the structure of this thesis.

1.1 Motivation

Abstraction is a common theme in computer science and allows to handle complexity and produce elegant models and designs [34]. The identification and separation of critical and irrelevant requirements, the modelling of a system architecture to satisfy functional and non-functional requirements, and the implementation of applications with the help of platforms and high-level programming languages are typical tasks in software development that benefit from abstraction. As technologies and methods are further developed to provide more suitable and efficient abstractions for simpler use in practice and increased productivity, one may ask whether it is helpful to program with models instead of repeatedly writing technology-specific source code for similar products. Therefore, one may program a reusable tool-chain, which translates similar models (semi-)automatically into working parts of an application. It is reasonable to assume that building models is less time-consuming and tedious than implementing an application with a large amount of repetitive infrastructure code in a high-level programming language.

Niklaus Wirth¹ published a book [54] in 1976 with the title

$$\textit{Algorithms} + \textit{Data Structures} = \textit{Programs}.$$

Practitioners of MDSD who attempt to program with models postulate their own version based on the above equation, i.e., [9]

$$\textit{Models} + \textit{Model Transformations} = \textit{Programs}. \quad (1)$$

Equation 1 is also the guiding theme for this thesis and will be further elaborated in the next pages. MDSD enables MDSD practitioners to use models as primary artifacts in software development and this work explores how this is done in detail. The expectation towards MDSD is that it will enable faster and more productive building of software.

1.2 Outline of Contribution

This thesis is located within MDSD and deals with the model-driven development of CRUD UIs. CRUD stands for Create, Read, Update, Delete and means the creation, retrieval, update, and deletion of entries in a database. An example for CRUD UIs is given in Figure 9, where the CRUD functionality is represented in the upper part of the UIs. The objective of this thesis consists in following two parts:

¹Swiss computer scientist awarded with the ACM Turing Award in 1984. Chief designer of countless programming languages, including Pascal.

1. Introduction

- i) *Elaboration of the basic principles of MDSD and a compilation of tools and methods.*
- ii) *Design and development of a MDSD method to create a CRUD UI from a set of static and dynamic conceptual models.*²

In the context of this work, a method consists of a process and a tool-chain that supports its individual steps. A process consists of activities, i.e., a description of the individual steps of the process, artifacts, i.e., what the input and output of the activities are, and roles, i.e., the people who take part in activities and are responsible for the artifacts.

To better illustrate the latter goal consider the development of a application for hospital management as an example. Figure 9 shows screenshots of a final product and Figure 10 displays a static conceptional model, which may appear after an initial requirement elicitation phase. Note that static models do not describe the interaction in a UI and further models are required for a more complete view of a system. The previously stated second goal for this example would be to develop a MDSD method that takes a set of fairly general models (cf. Figure 10) and transforms them into UIs similar to the ones shown in Figure 9.

Clearly, the proposed transformation from Figure 10 to Figure 9 is not directly possible and has to be divided into multiple steps. The model in Figure 10 is underspecified and there are no clear rules on how to perform this transformation. The conceptual model does contain some human-readable domain-specific concepts, e.g., the UML classes *Medication*, *Nurse*, and *Ward*, but they cannot be processed by an algorithm because they are simple strings and there are no domain-specific UML stereotypes (cf. Figure 11) attached to these classes. The next step could be to label the individual classes with domain-specific machine-processable annotations, e.g., UML stereotypes. The new model can then be processed by an algorithm, which reads the annotations and outputs a model with a higher level of detail and a lower level of abstraction. The new model is still fairly general, since it neither describes a software architecture nor contains technology-specific details. At some point in time, the generation of an initial UI, which can be refined in further steps with the help of an UI editor, seems sensible. Not all steps in a MDSD method have to be fully automatized, but they should be at least partially assisted by appropriate tools.

1.3 Challenges in Model-Driven Development

There are several issues with using models for more than just communication in development teams or between a development team and stakeholders.

On one hand, there are some conceptual aspects to be considered. Firstly, models need precise syntax and semantics to be machine-processable. Secondly, the modeling languages used should provide appropriate domain-specific abstractions so that models

²Static models emphasize the structure and architecture of a system, e.g., UML structure diagrams, while dynamic models capture the execution of a sequence of actions in a system, the interplay between components of a system, or the management and change of the internal state of a system, e.g., UML behavior diagrams. [9]

not only provide a higher level of abstraction than programming languages, but also an appropriate level of expressiveness. One may even work with a set of models that focus on different aspects of a system with different levels of abstraction. Thirdly, models may need to be refined and transformed iteratively, which requires the development of a method suited to the given problem domain.

On the other hand, there are some practical problems as well. Firstly, one has to ensure that models never become outdated and describe the system accurately on their abstraction level. Secondly, services and clients are often developed by separate teams, and different software versions may be in operation at the same time, which has been taken into account by the MDSD tool-chain. Thirdly, non-functional requirements may require to interpret models at runtime or a more flexible handling of API endpoints and resources, if one uses techniques such as load balancing. The discussion of challenges in adoption of MDSD in companies and the usability of MDSD tools are certainly interesting, but not the focus of this work.

1.4 Structure of This Thesis

This thesis is structured as follows:

Chapter Model-driven Software Development provides the theoretical foundation for this thesis. It defines and explains different approaches to MDSD. In addition, this chapter addresses the conceptual aspects mentioned in the introduction.

Next, works and software related to the MDSD are presented and discussed in Chapter Related Works. These are then used to develop this thesis MDSD solution in Chapter Implementation.

Chapter Analysis covers all assumptions and considerations needed to develop an MDSD approach for generating CRUD UIs. At the start, all user groups of this MDSD method and their requirements are examined. Next, the practical aspects mentioned in the introduction are discussed in detail. In addition, the use case outlined in Section Outline of Contribution is further specified.

Chapter Implementation gives an overview of the developed MDSD method. First, the general workflow is presented. Following this, the low-code modelling process is explained in detail. At the end of this chapter, an overview of the UI execution environment is given.

The last Chapter Conclusion summarizes the result of this thesis and provides an outlook on possible future work.

2 Model-driven Software Development

This chapter provides an introduction to Model-driven Software Development (MDSD) and related approaches. Furthermore, the benefits and drawbacks of code generation and model interpretation are discussed.

2.1 Overview of Model-driven Software Development

First, a generic description of MDSD is given, which serves as a template for its concrete realizations and as a basis for the MDSD method shown in Chapter Implementation. Several different realizations of MDSD are known, such as MDA, MDE and MDWE.³ Here, MDA will be described in more detail later as an example.

Models There is no consensus in the literature on the definition of a model. As a working definition for this thesis, the three main characteristics of a model stated by Stachowiak in his book on model theory [47] are listed.

- *Mapping feature:* A model is a representation of a natural or artificial original, which itself can be a model.
- *Reduction feature:* A model captures only those aspects of an original that the creator considers relevant and important.
- *Pragmatic feature:* Models always have a particular purpose.

Equation 1 from Chapter Introduction indicates that models and model transformations play important roles in MDSD. Compared to traditional software development approaches, models not only are used for documentation and communication purposes, but may take a role similar to source code in programming. They may serve as a basis for code generation or be directly executed by an interpreter and hence require a precise definition of their syntax and semantics, which is achieved by appropriate formalism such as metamodeling.

Metamodeling While models are an abstraction of concrete instances, a metamodel formally defines a set of possible models, thus a model is an instance of a metamodel. For example, one may capture a concrete business process by using UML activity diagrams, which are described by a UML metamodel. UML metamodels in turn are instances of a metamodel defined in Meta Object Facility (MOF). One can see that the term "meta" is relative with respect to two models (cf. Figure 1). This cascade of models could be in principle extended ad infinitum, but in practice a metamodel is reflexive, i.e., it is able to describe itself [48].

³The umbrella term MD* can be used to summarize acronyms such as MDA, MDE, MDSD, and MDWE. This thesis instead uses the acronym MDSD, when talking about related approaches.

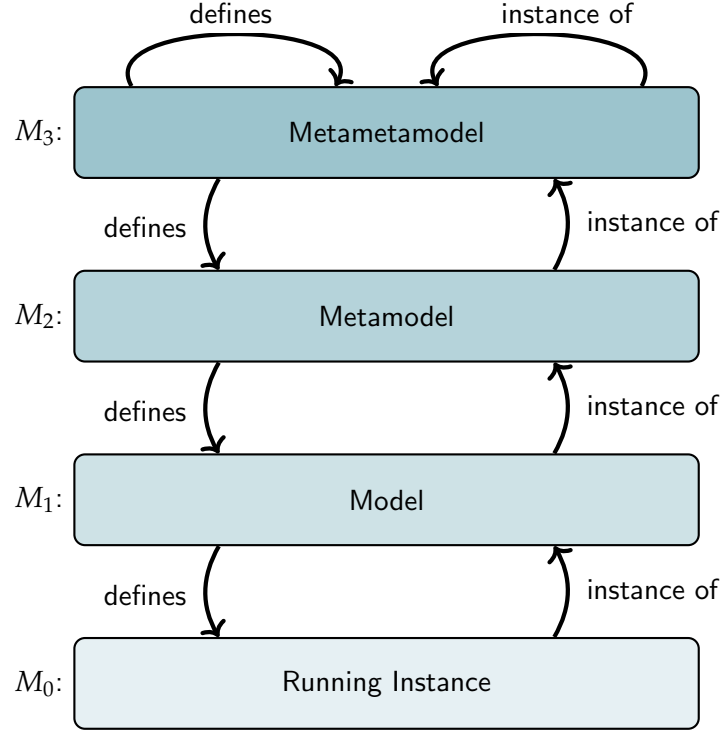


Figure 1: Illustration of a model hierarchy. The different levels in the hierarchy are called M_0 , M_1 , M_2 etc. In principle, this hierarchy can even be extended beyond the M_3 level, but typically the M_3 level has the special role that it defines itself [48]. As an example, consider the programming language \mathcal{C} . Written source code corresponds to M_1 and the running program to M_0 . M_2 level would contain a specification of \mathcal{C} with the help of a context-free grammar [33], which conforms to the definition of a context-free grammar at the M_3 level.

Domain-specific Language A Domain-specific Language (DSL) is used to capture and express relevant concepts of a domain [13]. The complexity of DSLs can range from simple files with configuration parameters for a software family to a full-fledged Third Generation Programming Language (3GL), possibly extended with domain-specific abstractions [48]. Examples of DSLs include SQL for relational databases, HTML for web technology, UML class diagrams for modelling object-oriented systems, and VHDL for describing digital circuit designs [9], [49]. The distinction between a DSL and a GPL is not clear-cut and depends on the given situation. For instance, UML may also be regarded as a GPL, since it can be used to model any domain [9]. Typically, DSLs should be as flexible as needed and as simple as possible [48]. In this regard, a 3GL like Java or C++ without domain-specific concepts would typically be a sub-par choice as modeling language.

Syntax and Semantics of a DSL Typically at least four ingredients are required to define a (semi-)formal DSL:

- *Abstract Syntax*: Defines the structure and operations of a DSL independently of

2. Model-driven Software Development

its realization.

- *Concrete Syntax*: Realization of some abstract syntax. Can take a textual or graphical form (or a combination of the two).
- *Static Semantics*: Specifies which elements conforming to an abstract syntax have a well-defined meaning. For instance, the restriction that uninitialized variables should not appear in a valid Java source code is realized via static semantics.
- *(Dynamic) Semantics*: Complements static semantics by giving further meaning to elements that are well-defined with respect to some static semantics.

A metamodel can be used to specify the abstract syntax and static semantics of a DSL. The user of a DSL uses its concrete syntax, which is a realization of an abstract syntax described by a metamodel. As a concrete example, one can specify the DSL SQL with the help of a BNF grammar [31]. The BNF grammar not only provides a metamodel, i.e., abstract syntax and static semantics, but in this case also a concrete syntax.⁴ [9]

The separation into abstract and concrete syntax allows for multiple DSLs acting as different view points on the same domain. For example, there can be two DSLs conforming to the same metamodel, where one DSL is textual and designed for IT experts, while the other DSL is graphical and more geared towards domain experts with less technical expertise. In case of the previous SQL example, one can define a graphical query builder for less proficient SQL users.

Further semantics of DSLs can be specified in different fashions. In case of the SQL example, the BNF grammar does not specify the meaning of keywords such as **SELECT**, **FROM**, **WHERE**, so the example requires the definition of additional semantics. Some example techniques for defining these additional semantics are given below. [9], [32], [48]

- *Denotational semantics*: Creates a mapping between a programming language and mathematical formalism.
- *Operational semantics*: Defines the semantics of a programming language by building an interpreter that directly describes the behavior of given programming language.
- *Translational semantics*: Specifies a programming language by a mapping to another programming language with already defined semantics.

Model Transformations Another key aspect of MDSD are Model-to-Model (M2M) and Model-to-Text (M2T) transformations. A desirable goal in MDSD is to create a tool-chain that takes a set of models as input and yields a (nearly) fully functional application by applying a sequence of M2M-transformations and a final M2T-transformation to the input. Architecturally, the described tool-chain follows a Pipes and Filters style, where

⁴The abstract and concrete syntax may not always be clearly separable, but they are not identical either! The specification of the metamodel via a BNF grammar is a particular case.

each M2M-transformation enhances its input with some details. This approach offers some flexibility by construction. For instance, instead of applying a M2T-transformation to construct an Android application, one could apply a M2T-transformation for creating an iOS application, i.e., the M2T-transformation can be adapted to the target platform. It should be noted that M2T-transformations may also be used to create test cases, further documentation, or deployment scripts. The described transformations need to be implemented, but it is possible that the return on investment can be reached faster compared to non-MDSD approaches, since the transformations are executed (semi-)automatically and the application no longer needs to be implemented by hand. In addition, good modularisation of a MDSD tool-chain can support reusability of transformations in further software projects. [9], [48].

It should be noted that metamodeling can be utilized to formally define the input and output of each transformation of a MDSD tool-chain. Therefore, not only the input and output of the tool-chain is required to conform to their respective metamodels, but intermediate artifacts of the M2M-transformations as well. Even model transformations themselves are treated as models in MDSD and defined by further metamodels.

Platforms The outlined tool-chain becomes especially useful, if the created application has access to a platform, e.g., a middleware, library, or framework. Examples for platforms include Jakarta EE, .NET, and CORBA, but the concrete meaning of a platform depends on the application and scope [37], [48]. In the case of code generation, the M2T-transformation can use components defined by a platform and does not need to implement a most likely inferior solution for already solved problems, making M2T-transformations simpler and less error-prone.

2.2 Model-Driven Architecture

Model-driven Architecture (MDA) is a flavor of MDSD specified by the Object Management Group (OMG) that focuses on portability and interoperability of MDSD software. A Platform Independent Model (PIM) and a Platform Specific Model (PSM) is defined relative to a prescribed platform. The difference between these models is that PIMs do not contain platform-specific details in contrast to PSMs. For example, consider the development of a MVC-based CRUD application for the J2SE platform. On one hand, the PIM could represent a GUI layout of a CRUD interface, which can be described with UML class diagrams and a UML profile for GUIs (e.g. [45]). On the other hand, the PSM structures the GUI into MVC-pattern and explicitly references resources from the J2SE platform. Moreover, the term Computation Independent Model (CIM) can be introduced to distinct pure business models from PIMs that define high-level system architectures. CIMs are often defined informally, e.g., textual use cases and paper prototypes are examples for CIMs. [5], [37], [48]

The OMG defines a set of standards for DSLs and transformations that can be used to implement a MDA tool-chain. Any language used in MDA must work with MOF-compatible tools and therefore be MOF-based. Usually, models are expressed using UML and UML profiles. However, this is not mandatory as long as the modeling

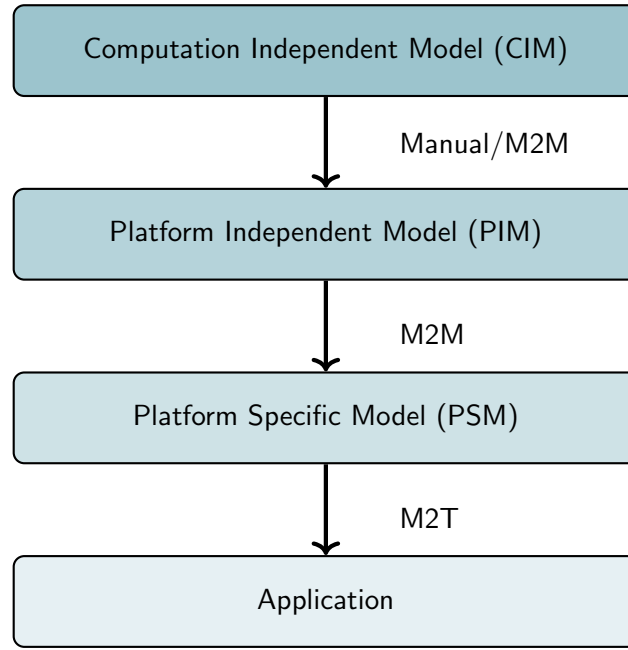


Figure 2: Overview of MDA hierarchy. The deeper the layer in the hierarchy, the more detailed and technical are the contained models. The arrows indicate model transformations between models with different levels of detail. Horizontal transformations between models at the same level are not shown.

language used is MOF-based. Moreover, semantics in form of constraints, e.g. invariants and pre- and postconditions of functions, can be modeled using the Object Constraint Language (OCL). M2M-Transformations can be described using the MOF-based Query View Transformation (QVT) specification. For specifying M2T-Transformations the OMG standard called MOF Model to Text (MOFM2T) language can be used. All MOF-based models are supposed to be serializable in a standardized XML Metadata Interchange (XMI) format. There are also JSON-based formats such as JSOI that can be used as interchange format for models [29]. Note how these standardization efforts support the main goals of MDA mentioned in the previous paragraph. [9]

2.3 Code Generation vs. Model Interpretation

Source code can be run directly by an interpreter or compiled into a program and later executed by the OS. There are also hybrid approaches as in Java, where Java byte code is first generated from Java source code and later interpreted by the Java virtual machine. MDSD is similar in the sense that models substitute for the role of source code. One has to decide whether the MDSD tool-chain is a code generator or directly interprets the input model. Overeem et al. call the former approach generative MDSD and the latter approach interpretive MDSD. Of course, hybrid approaches are still possible in this setting. For example, the input model may be first simplified by a transformation into a second model, which is then interpreted at run-time, or a MDSD tool-chain may even match-and-mix both code generation and model interpretation for different part of its application. [40]

At first glance, the outlined approaches may seem equivalent in terms of functional requirements, but depending on the non-functional requirements, one of these solutions may seem superior to the other. Overeem et al. [40] conducted a literature study and compiled the results of thirty-five articles on the performance of the above approaches with respect to different non-functional requirements. In addition, they interviewed twenty-two product experts of sixteen different Software Producing Organisations (SPOs) that develop Model-driven Engineering Environment (MDEE) to learn who the target users of MDEE are and how SPOs make design decisions when developing MDEEs.

While generative MDSD appears superior to interpretive MDSD in terms of run-time time behavior and resource allocation, interpretive MDSD beats its counterpart in the build-time time behavior category. These results seem intuitive, since interpreters create additional overhead during the execution of an application, but the application does not need to be compiled before execution. Overeem et al. [40] found out that most authors agree that interpretive MDSD leads to more modular, analysable, and modifiable MDSD tool-chains. In contrast, the evidence on testability was not conclusive enough to decide whether a generative or interpretive approach is better suited for this aspect. Moreover, Overeem et al. [40] claim that SPOs have not an explicit rational for the design of their MDEE and that there is a lack of guidance and knowledge for SPOs.

Good build-time time behavior is especially helpful when software developers and domain experts need to collaborate in order to create an appropriate domain model and MDSD tool-chain, because changes in the input model are reflected instantaneously. Furthermore, a tool-chain based on model interpretation only needs to be developed once and can then be deployed with different input models to meet different customer requirements. With generative MDSD, one would have to compile a new version each time instead. However, the flexibility of interpretive MDSD has a serious drawback. Here, the model is exposed to the application, which can lead to confidentiality and security issues.

3 Related Works

This chapter lists and describes related systems and DSLs which are important for this thesis.

3.1 MockupToME

Between 2008 and 2010 Basso et al. developed the MockupToMe methodology for Model-driven Web Engineering (MDWE) of CRUD, Filter, List, and Report use cases based on MDA. MDWE means MDSD applied to web development. Web front-ends are usually represented by GUI components and behavioral diagrams, but these models need to be decorated with semantics for the actions of users, screen flows and business logic. The manual enrichment of the intermediate products in MDWE is time-consuming and discourages the use of MDWE, especially in software projects, where short iterations are needed. Basso et al. give a solution to the described problem by developing automated design techniques and a custom DSL called MockupToMe DSL that overcomes limitations of the GUI profile.⁵ For instance, they added the concept for Master/Detail views, which is needed for more complex CRUD interfaces with nested input masks. [5]

The authors of [5] give a summary of their methodology in four abstract levels:

- An initial *paper prototyping* phase, where a requirements engineer elicits textual use cases and paper prototypes. A conceptional UML model and a use case diagram, both annotated with stereotypes from the CRUD UML profile, are created. The end-products of this level are created fully manually and correspond to CIMs described in Chapter Overview of Model-driven Software Development.
- The *evolutionary prototyping* phase, where a mockup designer creates preliminary mockup models with the MockupToMe DSL and proposes different UI variants to the client. Based on client feedback, the initial mockups are further refined and annotated. Finally, a runnable prototype is created that is accepted by the client. The prototype is part of the PIM and the transition from CIM to PIM is performed by the mockup designer.
- In the *architectural prototyping* phase, a software architect structures the refined prototype from the previous phase with the help of predefined M2M-transformations into MVC-based application models. UML structural and behavioral diagrams and concrete GUI components are obtained with the help of M2M-transformations. After this process, the GUI components are further refined with annotations to prepare for the M2T-transformations and optionally a set of different UML profiles (GUI DSLs, Action Profile, ORM Profile, and Service Profile) is applied. This final step resembles the transition from a PIM to a PSM.
- *Functional prototyping* is the last phase in the MockupToMe methodology, where the M2T-transformations are executed. The input of the M2T-transformations are the previously generated UML structural and behavioral diagrams and the

⁵UML is used as modelling language by the authors.

refined platform-specific GUI components. The client can interact with a working piece of the application and perform an acceptance test.

3.2 JSON Schema

JSON Schema is an extensive JSON format specification standardized by the IETF [30]. JSON schema is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data [23]. Both JSON:API and JSON Forms are based on this standard.

3.3 JSON:API

JSON:API is a format specification for lightweight JSON responses. It supports sparse fieldsets, i.e., an client may request that an API endpoint returns only specific fields in the response, and pagination links, i.e., the server may provide links to traverse a paginated data set. An API endpoint may return an array of resource objects that are related to the primary data via the attribute `included` (cf. Listing 1 and Listing 2). JSON:API can be used to structure inputs and outputs in a MDSD method or for the data transmission between client and service. [55]

```
{
  "jsonapi": {
    "version": "1.0"
  },
  "data": [
    {
      "type": "Doctor",
      "id": "37251",
      "attributes": {
        "name": "Dietrich",
        "surname": "Peters",
        "title": "Dr. med."
      },
      "relationships": {
        "address": {
          "links": {
            "self": "http://api.clinic.com/doc/37251/relationships/address",
            "related": "http://api.clinic.com/doc/37251/address"
          }
        },
        "data": {
          "type": "Location",
          "id": "273123"
        }
      }
    }
  ]
}
```

3. Related Works

```
],  
  "included": [  
    {  
      "type": "Location",  
      "id": "273123",  
      "attributes": {  
        "country": "DE",  
        "city": "Jena",  
        "street": "Im Sommerfeld 164"  
      }  
    }  
  ]  
}
```

Listing 1: Example of a JSON object conforming to JSON:API specification. A `Location` object is related to the primary object `Doctor` and is contained in `included`.

```
GET /doctors?id=37251&include=address&fields[address]=country,  
    city,street  
HTTP/1.1
```

Listing 2: This HTTP request may lead to the response in Listing 1. A doctor with `id 37251` is retrieved together with its addresses. Sparse fieldsets are used to filter the corresponding address objects for the attributes `country`, `city`, and `street`, omitting other attributes such as `zipcode`. The `included` parameter ensures that not only references to related objects are supplied, but also the referenced objects themselves. In this example, `included=address` means that the API endpoint returns the address objects related to the requested doctor. This feature can be used to reduce the number of requests in a client.

3.4 JSON Forms

JSON Forms is a JSON Schema based approach for creating forms and comes with off the shelf support for React, Angular and Vue. This framework requires as input a data schema that defines the underlying data to be shown in the UI and a UI schema that defines how the data is rendered as a form.⁶ This framework can be extended with custom renderers to offer custom components and to support further web frameworks and libraries. Moreover, more complex forms that require conditional rendering or validation of inputs can be realized with JSON Forms. JSON Forms uses the JSON schema validator AJV to validate constraints in the form of regular expressions, data type checks of inputs, and user-defined JS functions. [20]

```
{  
  "properties": {  
    "staff_data": {  
      "$ref": "api.clinic.com/schemas/staff/staff_data.json"  
    }  
  }  
}
```

⁶The UI schema is auto-generated if not provided.

```

    },
    "address": {
      "type": "object",
      "properties": {
        "street": {
          "title": "Street",
          "type": "string"
        },
        "city_postcode": {
          "title": "Postcode/City",
          "type": "string"
        },
        "State_Country": {
          "title": "Form of address",
          "type": "string"
        }
      }
    }
  }
}

```

Listing 3: Example of a data schema to be rendered in a UI. Further JSON files with data schema definitions can be referenced via `$ref`.

3.5 Eclipse Modeling Framework

Eclipse is a well-known and extendable open-source IDE [17]. The Eclipse Modelling Project (EMP) focuses on the evolution of model-based development technologies within the Eclipse community [18] and includes the Eclipse Modelling Framework (EMF) used to develop tools for code generation based on the ECore metamodel. The ECore metamodel is a metamodel similar to UML and specifies the data structure and concepts of models developed with the EMF [19]. Moreover, it contains the EMF.Edit framework including generic classes for building editors for EMF models and the code generation facility EMF.Codegen, which is capable of building a complete editor for an EMF model [19].

3.6 Interaction Flow Modeling Language (IFML)

IFML is standardized and published in 2015 by the OMG and can be used to specify front-ends, independently of their concrete implementation details. It is a comprehensive DSL, which covers the following aspects of front-ends of applications: [9], [38]

- The composition of the view in terms of view containers and view components.
- The content of the view, i.e., the data to be displayed on the UI.
- Commands and interaction events supported by the UI.
- Control logic that determines the actions to be executed after triggering an event.

3. Related Works

- Parameter binding, defining the data items shared between elements of the UI.

Presentation aspects of a UI cannot be captured with IFML [9]. Acerbis et al. claim that IFML is inspired by WebML and that it includes several innovations such as the separation of business logic and UI specifications and that the core concepts of IFML can be applied to any type of UIs [1].

3.7 IFMLEdit.org

Bernaschina created the web-based IFML editor IFMLEdit.org [7] as a case study for his M2M and M2T transformation framework ALMOsT.js [6]. IFMLEdit.org does not require the installation of any software and is thus a convenient way to create an IFML model (cf. Figure 5). The created model can be exported as a JSON file.

Since this software is a PoC for ALMOsT.js implemented by a single person, the functionality of this application is reduced. The focus of IFMLEdit.org is on UIs with «List» and «Detail» elements and less on «Form» elements. There is no pre-defined example with a «Form» element on the website and the form does only support simple textfields. Additionally, there is no export to XMI, which would be more practical for a MDA toolchain (cf. Section 2.2).

3.8 Webratio Platform

Webratio platform is a tool for model-driven development of web and mobile applications based on IFML. This platform can be used in conjunction with Eclipse and offers various features such as model checking, full-code generation, and lifecycle management. Additionally, Webratio platform supports the implementation of custom components and contains a layout template and style design environment for UI layout and styling. The modelled application can then be generated and tested with the included webserver. [1]

Staff Data

Last name

First name(s)

Form of address

Title

Birth date

Nationality

User name

Address

Street

Postcode/City

Form of address

Figure 3: Form generated from Listing 3. The layout can be further specified via a UI schema (not shown in Listing 3). Otherwise a standard vertical layout is generated. The information for staff data referenced via `$ref` in Listing 3 is resolved and displayed.

3. Related Works

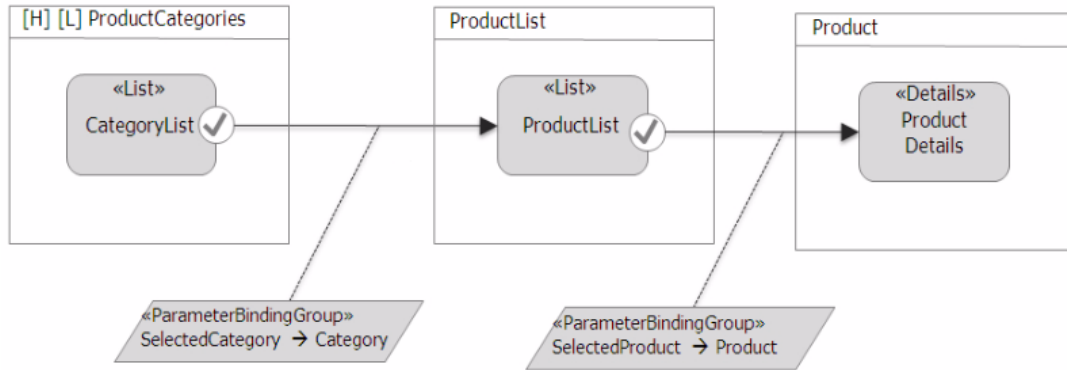


Figure 4: Modelling example of an online bookstore with IFML from the specification [38].

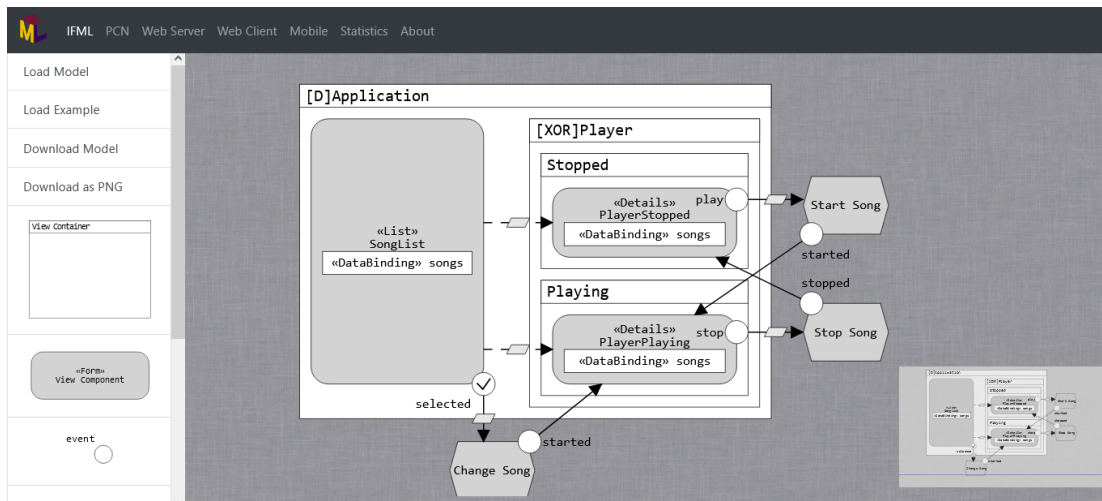


Figure 5: Screenshot of IFMLEdit.org. The IFML model in this figure shows a basic music player.

4 Analysis

Chapter Analysis describes the requirements towards the MDSD method developed in this thesis and the target user group. The practical problems mentioned in Section Challenges in Model-Driven Development are further elaborated. Additionally, the application example for the MDSD method sketched in Section Outline of Contribution is specified.

4.1 Assumptions

When designing a system, its boundaries and assumptions should also be stated. The design of back-end services is not considered by this thesis. The structure of each response from an API endpoint is assumed to conform to some version of the JSON:API standard (cf. Section 3.3). However, later semantic changes to back-end services, e.g., the renaming or addition of attributes in response data objects, will likely have an impact on the interaction between front-end and back-end and must be considered.

Tools for UI design are not developed, but the proposed MDSD method has still to define activities where the development of UI forms would take place. Recall that the focus of this thesis is on MDSD tool-chains based on model interpretation (cf. Section 2.3), and hence the MDSD tool-chain is based on this paradigm.

4.2 User Group

The user group of the MDSD method consists entirely of software developers, which is by no means an uniform group. Here is a list of roles, which may appear in the course of a project:

- *Domain expert*: Models the application domain and is, at least indirectly, responsible for the content of an UI and thus sets the general conditions for developers. This group is not directly targeted in this thesis.
- *UI designer*: Set the layout, content, and navigation of UIs. The focus of this role is on the appearance and design of an UI. The data binding aspect between UIs and services is not handled by this role.
- *Front-end developer*: Uses the information provided by the UI designers and domain experts to implement the UI. Additionally, this role implements the communication between front-end and back-end and thus binds elements from the UI to services.
- *Back-end developer*: Develops the API endpoints. This group is not primarily targeted by the MDSD method, but back-end developers can introduce changes to API endpoints that may affect the work of front-end developers.

In summary, the target of the MDSD method are (mainly) technical experts.

4.3 Problems

As mentioned in Chapter Introduction, the main goal of this thesis is to define a MDSD method, which takes an appropriate set of inputs (cf. Figure 10) and produces a CRUD UI (cf. Figure 9). This goal comes with some technical and process related issues. The ones found through interviews with members from t2informatik GmbH [50] are listed below.

1. *Process related problems:* This category includes problems arising from a high-level point of view on the MDSD methodology. Of course, the required activities, their artifacts, and the involved roles have to be first identified and defined, depending on the desired end goal. Furthermore, the following specific process related problems are identified as important in context of this thesis:
 - (a) *Adjustable UI appearance:* Regardless of the concrete MDSD process, there should be some activity in which an initial (generated) UI layout can be further (semi-)manually refined. The refined UI should still be compatible as an input for further activities of the MDSD process.
 - (b) *Collaboration of multiple independent teams:* The MDSD process should allow for activities that can be performed by independent teams. On one hand, the inputs to these activities should ideally be sufficiently modularized to allow for this. On the other hand, the artifacts produced by these teams should be combinable without high overhead.
2. *Technical problems:* This part is divided into general problems that appear regardless of the MDSD tool-chain implementation and specific problems that are more relevant when working with model interpretation.
 - (a) *General technical problems:* These listed problems must be considered whether a generative or interpretive approach is used.
 - i. *UI description:* A concise and holistic description of an UI has to be specified. In case of a multi-page UI the following aspects are worth considering:
 - the static structure, i.e., which elements are shown on which screens,
 - the user flow, and
 - the visual appearance of the UI.
 - ii. *Data binding:* UIs communicate with services to fetch and transmit data. This requires data binding between the elements of an UI and the data and methods provided by a service.
 - iii. *Parallel development:* The MDSD method should allow for different consumer and service versions to run in parallel.
 - (b) *Specific technical problems:* Recall that the end product consists of an interpreter and a model to be interpreted in case of model interpretation (cf. Section 2.3). The question arises whether the model should be delivered together with the interpreter or whether the model should be provided by a service and retrieved by a client at run-time. A hybrid approach is also

possible here, where some static assets are provided along with the interpreter and other dynamic resources are loaded at run-time. The following two consequences are important for this thesis:

- i. *Lightweight data transmission:* The transmission of models and resources should be lightweight and omit unnecessary information. For instance, when using JSON Forms (cf. Section 3.4), it is not advisable to transmit schema information for forms in which the client is not interested, especially in case of mobile devices as clients.
- ii. *Validation rules:* Form data needs to be validated at least on server-side. One may ask, whether a client-side validation of form data is desirable to reduce server load and give faster user feedback. Static distribution of validation rules is inflexible and may lead to outdated validation rules, when further versions of the end product are developed. In the worst case, the server may reject form data if the validation rules are not synchronized between the service and the client.

4.4 Pseudo-Requirements

There are two pseudo-requirements for this thesis:

1. Front-end produced by proposed MDSD method must be able to handle JSON:API responses.
2. JSON Forms must be the core library used by the MDSD tool-chain.

4.5 Application Example

The general setting is a clinic administration software for the management of multiple clinical centers. Two UI screens are described in more detail as application example for the proposed MDSD method (cf. Figure 7).

The UI in the first screenshot manages notifications for a set of teams. Each team consists of multiple staff members for whom notifications are created. The teams are chosen via the 'Colour' dropdown menu. A notification can be either a message or a simple comment, which can be selected with the dropdown menu 'Type'. Depending on the value in 'Type' different options are selectable or not in the checkbox.

In the second screenshot, shifts can be created to which teams can be assigned later. Depending on the clinic selected in the dropdown menu 'Center', a different set of wards can be chosen, i.e., there are two dependent dropdown menus in this UI.



Figure 6: ERM describing a possible relationship between entities *Shift*, *Center*, and *Ward*.

4. Analysis

When modelling shifts, centers, and wards as separate entities like in Figure 6, the problem arises that the UI not only needs to query the *work* relationship between *Work* and *Center*, but also the *has* relationship between *Center* and *Ward*.

As one can see, the UI screenshots in Figure 7 and Figure 9 follow a Master/Detail pattern. The table gives an overview over the elements stored in the back-end, i.e., the table takes the role of the master element. Elements can be created or modified in the form, i.e., the form takes the role of the detail element. This seems to be a sensible choice for CRUD UIs in general, since one can get an overview of the stored elements and edit individual elements at the same time.

Components, whose selectable options depend on each other, are not supported by JSON Forms by default. A possible solution is to implement and register a custom renderer that manages two or more concrete dependent components. Furthermore, the client needs information about the dependency in case of two or more UI elements, e.g., which ward options are displayed if the center 'B. Brown' is chosen in the shifts screen (cf. Figure 7). Validation rules for UI elements can be, for instance, distributed together with one of the JSON inputs for JSON Forms, i.e., in the data schema or the UI schema, if possible, or these rules have to be distributed via an additional file and checked by another library or custom code.⁷ A way to transmit separate validation rules, or a referrer to these rules, is as property of the `meta` field of a JSON:API response. Using sparse fieldsets and embedding related resources with `include` is also important to address Problem 2, (b), i. Considering these implementation details seems to be a rather straightforward task at first glance, but combining these aspects together with the MDSD methodology requires careful consideration, since the specific problem is not known in advance, apart from the creation of CRUD UIs in the context of this work.

⁷AJV would be a possible candidate, since it is integrated in JSON Forms.

4.5 Application Example

New
Delete
Notification Categories
Apply
Cancel

Details
Category name: Appointments for patients
Colour: Dark blue
Type: Message
☒ Needs confirmation
☐ Allow repetition
☒ Pass to monitoring
☐ Used for monitoring comments
Comments:

Name	Colour	Type	Needs confi...	Allow repetition	Pass to Mon...	Default for ...	Comment
Appointments for patients	Dark Blue	Message	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Appointments for patients...	Dark Blue	Message	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Care handover	Dark Green	Comment	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Doctors prescription once	White	Message	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Doctors prescription with r...	White	Message	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Individual Patient Directive	Dark Blue	Message	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Missed Dialysis	Red	Comment	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Report from CQI Meeting	Magenta	Comment	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Report from Dialysis treat...	Yellow	Comment	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	(from Monitoring)
Report on current problems	Cyan	Comment	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Report on current wound	Green	Comment	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Report on nursing	Blue	Comment	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Update From Hospitalisation	Red	Comment	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

New
Delete
Shifts
Apply
Cancel

Details
Shift name: M-W-F AM Ward1
Center: B. Braun Avitum ...
Ward: Ward 1
Shift start time: 07:00
Shift end time: 12:00
Comments:
Weekdays:
☒ Monday
☐ Tuesday
☒ Wednesday
☐ Thursday
☒ Friday
☐ Saturday
☐ Sunday

Name	Center	Ward	StartTime	EndTime	Comment
M-W-F AM Ward1	B. Braun Avitum Re...	Ward 1	07:00	12:00	
M-W-F AM Ward2	B. Braun Avitum Re...	Ward 2	07:00	12:00	
M-W-F PM Ward1	B. Braun Avitum Re...	Ward 1	13:00	18:00	
M-W-F PM Ward2	B. Braun Avitum Re...	Ward 2	13:00	18:00	
T-Th-S AM Ward1	B. Braun Avitum Re...	Ward 1	07:00	12:00	
T-Th-S AM Ward2	B. Braun Avitum Re...	Ward 2	07:00	12:00	
T-Th-S PM Ward1	B. Braun Avitum Re...	Ward 1	13:00	18:00	
T-Th-S PM Ward2	B. Braun Avitum Re...	Ward 2	13:00	18:00	

Figure 7: UI screenshots concerning the application example .

5 Implementation

This chapter proposes an MDSD method that addresses the situation described in Chapter Analysis. First, an overview of the designed method is given. Then, the Low Code Platform (LCP) used to design CRUD UIs is explained in detail. Afterwards, an interpreter used to execute the created artifacts is discussed. The problems listed in Chapter Analysis are used at the end of each section to evaluate the MDSD method.

5.1 General Workflow

In the proposed MDSD approach, UI designers focus on creating UIs without considering issues such as validation and data binding to API endpoints, and front-end developers instead focus on integrating these forms into complex interfaces and processes.

In recent years, the term Low-Code Platforms (LCPs) has become popular [8], [11], [46]. LCPs are based on MDSD principles, but its intended target group are non-technical specialists in particular. According to Sahay et al. [46], the development process in a LCP⁸ consists of the following five steps:

1. Data modelling
2. User interface definition
3. Specification of business logic rules and workflows
4. Integration of external services via third-party APIs
5. Application Deployment

The general structure of the MDSD tool-chain can be summarized by the following steps:

1. A LCP for the creation of JSON Forms inputs and client-side validation rules.
2. A standalone library, which executes the artifacts from the previous point.

The first four points listed by Sahay et al. are covered in the next chapter. The deployment of the artifacts generated by the LCP, i.e., the library needed for application deployment, is considered in the subsequent chapter.

5.2 Low-Code Modelling

5.2.1 Description

Assumptions Note that developers working on the back-end of an application may have a different perspective on the domain model than developers working on the front-end. Resources appearing in the domain model can be scattered across different services

⁸Sahay et al. use the similar term Low-Code Development Platform.

and additional structures from the solution space may appear. Furthermore, attributes and relationships can be renamed in the course of the development of the back-end. Thus, the following assumption is important for the remaining steps:

The domain model from the view of front-end and back-end developers contains essentially the same information, but can be structured differently.

Step 0: Domain Modelling

Input: \emptyset

Output: Domain model as UML or ERM.

Roles involved: Domain experts.

The creation of a CIM is usually the first step in building an application. This step falls outside the scope of this process and is therefore not further specified, but is nevertheless of great importance for further activities.

Step 1: Selection of Primary Resources

Input: Domain model from Step 0.

Output: Basic CRUD UI skeleton for each primary resource.

Roles involved: UI designers.

As mentioned earlier, the goal is to construct a CRUD UI that fits an already defined domain model. The UI designer first chooses a set of entities from the domain model and declares them as primary resources. This creates a Master/Detail skeleton as first UI definition for each primary resource according to the IFML diagram in Figure 8. A basic workflow is already given by the Master/Detail skeleton. Each of the primary resource is shown in a app bar similarly to Figure 7.

Step 2: UI Refinement

Input: Set of basic Master/Detail skeletons from Step 1 and domain model from Step 0.

Output: Refined UI designs for each Master/Detail skeleton.

Roles involved: UI designers.

One can observe that Figure 8 does not give any information about the concrete UI appearance. Thus, the next step is to design the «List» and «Form» view components in Figure 8 with a WYSIWYG-alike UI editor. The UI designer also binds UI elements to elements from the domain model from Step 0.

5. Implementation

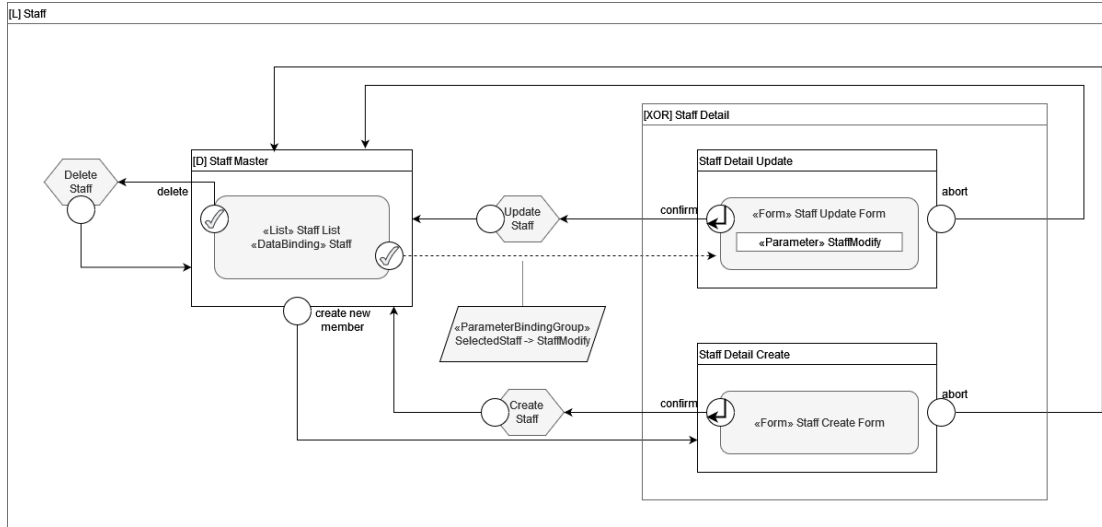


Figure 8: Basic Master/Detail skeleton for a *Staff* resource as example. A landmark view container (marked with [L]), which is reachable from any UI screen, is encapsulating the whole Master/Detail structure. A list of *Staff* data is shown first to the user, since its view component is encapsulated in a default view container (marked with [D]). Depending on the user actions, *Staff* data can be deleted, updated or created. A quick reference card for IFML notation can be found in [53].

Step 2a: Modelling Basic Attributes The UI designer sets the columns of «List» view components and the design of the «Form» view components.⁹ A set of common UI elements such as textfields, dropdown menus, and radio buttons can be selected by the UI designer and placed on the UI screen.

Step 2b: Modelling of Secondary CRUD UIs A resource may have relationships to further resources. A UI designer can create an additional Master/Detail skeleton similar to Figure 8 to define CRUD operations for these relationships. For instance, in a UI screen for the management of teams, one may want to edit the staff members of a selected team. Step 2a is repeated for this newly created Master/Detail skeleton, but not Step 2b.

A requirement for Step 5 is that the UI components are organized in a component tree by the WYSIWYG-alike UI editor. At this point the UI can be shown to non-technical stakeholders for feedback and Step 2 can be repeated until a satisfactory UI design is achieved.

Consider the *Shifts* UI screen in Figure 7. Assume that the underlying domain model contains the ERM in Figure 6. When modelling the dropdown menus 'Center' and 'Ward', the WYSIWYG-alike UI editor should offer an option to bind the *Center* entity, i.e., the other end point of the relationship between *Shift* and *Center*, to the 'Center' dropdown menu. The WYSIWYG-alike UI editor should now include the relationships

⁹See Figure 11 for an explanation of the «...» notation.

of *Center* as binding options in similar way.^{10,11,12} Now, the *Ward* entity should be assignable to the 'Ward' dropdown menu, since it has a relationship with *Center* and can be uniquely inferred when traversing Figure 6 from *Shift* to *Ward*. Finally, for both dropdown menus the attribute to be displayed must be specified, since both *Shift* and *Ward* can contain multiple attributes. Note that this problem cannot be resolved by Step 2b, because it introduces another UI (sub-)screen.

Step 3: M2M-Transformation for Data Bindings

Input: Refined UI design from Step 2, domain model from Step 0, information about the representation of the domain model in the back-end.

Output: Object of type `ResourceFile` (cf. Listing 4) and a file containing the mapping between the domain model from Step 0 and the back-end representation of the domain model.

Roles involved: Mainly front-end developers. Additionally back-end developers, when questions arise.

The UI designers performed a data binding between UI elements and the domain model from Step 0 in the previous step. These bindings do not necessarily have to be compatible with the back-end representation of the domain model. Thus, front-end developers may have to adapt these bindings to the back-end representation of the domain model.

Firstly, the location of the resources from Step 1 has to be defined. The output of this step is a file of type `ResourceFile` (cf. Listing 4). Secondly, the bindings defined on the domain model from Step 0 have to be mapped to the back-end domain model representation. The advantage of defining an additional mapping file is that UI designer can still work on the domain model from Step 0, if changes of UI content or appearance is required. This step can be interpreted as a manual M2M-transformation between the initial domain model and the back-end representation of it. An additional tool besides the mentioned WYSIWYG-alike UI editor used previously by the UI designers could help with this task, but it cannot be easily automated, since it requires semantic knowledge of both models and the transformation depends on the problem domain and back-end implementation.

Step 4: Annotation of UI Screens With Client-Side Validation Rules

Input: Output of Step 2.

Output: Refined UI designs annotated with client-side validation rules.

Roles involved: Front-end developers.

¹⁰With the exception of the previously binded relationship.

¹¹It can be helpful to track the 'context' of the relationship. One can use the context to resolve collisions, which would appear if *Shift* would have an additional relationship with *Ward*.

¹²The case of multiple relationships between two entities is not discussed.

¹³Step 3 can be omitted, if the domain model from Step 0 and the back-end domain model representation do match, contrary to the assumption from the beginning of this subchapter.

5. Implementation

The front-end developers annotate the «Form» view components in the artifact from Step 2 with client-side validation rules such as regular expressions for e-mail addresses, the minimum and maximum bounds of number inputs, or even custom code. The WYSIWYG-alike UI editor should offer an advanced view, where UI elements can be annotated by front-end developers. The validation rules can be specified for the UI bindings from the initial domain model, since they can be transformed with the mapping defined in Step 3.

More complex client-side validation rules that cannot be processed alone by JSON Forms are handled by AJV. Thus, complex client-side validation rules are formulated by front-end developers as JSON Schema according to the API of AJV. Depending on the scope of a client-side validation rule, a front-end developer may annotate a single UI element, a group of UI elements, or the form itself.

Regarding the application example from Chapter Analysis, the front-end developer could specify client-side validation rules for the dropdown menu and checkbox of the *Notification* UI screen (cf. Figure 7) using the UI bindings referencing the initial domain model. The front-end developer may also specify the center and ward dependency of the *Shifts* UI screen, but this dependency can be deduced automatically from the JSON:API responses of the API endpoint under mild assumptions, which is explained in Chapter UI Execution Environment.

Step 5: M2T-Transformation for UI Structure and Appearance

Input: Output of Step 4.

Output: Set of JSON Forms schemas and UI schemas, possibly a file containing validation rules, and possibly a configuration file for tables to be displayed.

Roles involved: None (fully automated step).

An algorithm based on depth-first search or breath-first search, which constructs a JSON Forms UI schema (cf. Listing 5) from the component tree defined in the WYSIWYG-alike UI editor can be applied to each «Form» view component. The JSON Forms schema (cf. Listing 3) can be directly extracted from configurations in the WYSIWYG-alike UI editor. The UI bindings are taken from the initial domain model and can later be transformed by the UI execution environment to match the back-end domain model representation.

The client-side validation rules are handled differently depending on their scope and complexity. Constraints such as data types and the range of number fields can be written directly to the JSON Forms schema, as it can be directly processed by JSON Forms. More complex constraints requiring AJV are exported to a separate file.

Last but not least, the configuration of the «Table» view components must also be exported.

Extension With IFML Interpreter The proposed LCP can be supplemented by an IFML interpreter as the Master/Detail skeleton in Figure 8 is formulated in IFML. The end product with this extension would look like IFMLEdit.org with additional steps for

validation rules and data binding to the back-end. The UI components obtained after Step 2 can then be in principle extended by additional IFML constructs. In this way its possible to have an approach that works fully on model interpretation as the UI execution environment with JSON Forms as core is based on model interpretation.

5.2.2 Evaluation

It is difficult to extract which attributes and relationships are important to show in a table or form, thus the forms and tables need to be designed by the UI designer. However, Step 1 provides an initial Master/Detail skeleton to the UI designer and Step 2 makes the tables and forms adjustable. Therefore, the proposed MDSD method covers Problem 1, (a).

Only one role is involved in each step, with the exception of Step 3. Hence, Problem 1, (b) is also addressed by the proposed MDSD method. Step 3 and Step 4 can even be performed in parallel.

The non-extended version allows the UI designer to define the content and visual appearance of the UI, but the user flow is restricted to a set of Master/Detail skeletons. Problem 2, (a), i. can be tackled by combining the proposed MDSD method with an IFML editor and interpreter. IFML is an appropriate modelling language to describe the UI flow, but may require additional training.

Problem 2, (a), ii. is explicitly addressed by Step 3 and Step 4 partially concerns Problem 2, (b), ii. since it allows the definition of client-side validation rules.

5.3 UI Execution Environment

This part describes the implementation of an interpreter that can be used to execute the artifacts generated by Step 3 and Step 5 from Chapter Low-Code Modelling. The interpreter can also be used as standalone library for any CRUD UI, regardless of the problem domain of the developed application.

Recall that JSON Forms is able to show a form based on a data schema and a UI schema (cf. Listing 3 and Figure 3). The input to the form is tracked with a JSON object, but any communication between client and service is not implemented by JSON Forms and must be defined outside of it. For instance, a submit button for a form is not provided by JSON Forms and needs to be implemented outside of it. Components such as tables and app bars are not available in JSON Forms per default. However, one can write and register custom renderers to add custom components to JSON Forms. Overall, JSON Forms is useful to display and manage the core of a form based on a model interpretation, but not enough to implement a whole CRUD UI including business logic. Thus, the proposed MDSD tool-chain contains an UI execution environment that uses JSON Forms and supplements the missing parts for functioning CRUD UIs.

5. Implementation

5.3.1 Input Specification

Mandatory inputs:

- File of type `ResourceFile` (cf. Listing 4) containing links to API endpoints with resources and corresponding JSON Forms data schema and possibly corresponding JSON Forms UI schema.

Optional inputs:

- Mapping file from Step 3 from Chapter Low-Code Modelling.
- Configuration file for tables to be displayed.
- File with client-side validation rules.

All these inputs can be created by the approach in Chapter Low-Code Modelling.

5.3.2 Implementation Details

Custom Components and State Management JSON Forms has indeed an extendable architecture and provides a method to register new custom components. Unfortunately, every UI element in JSON Forms can only see its own state. JSON Forms uses Redux to manage its state and it does not offer a convenient method to expose the whole form state of JSON Forms, i.e., a part of the Redux store of JSON Forms, to a custom component.¹⁴ As a workaround, one may work with an own Redux store, which mimics the form state of JSON Forms. To implement the UI execution environment, almost every UI element must be customized to allow for data sharing between UI elements. The data sharing can then be used to cover the use cases described in Chapter Application Example. Fortunately, the JSON Forms UI elements can be reused to define the needed custom components.

A database with custom components can be created to support the LCP from the previous chapter.

Relationship Processing and Traversal Recall that resources in JSON:API responses have a `data` field containing the fields `attributes` and `relationships` (cf. Listing 1). A connection between the naming conventions of the forms and tables and the API response can be made by using the mapping file from Step 3 from Chapter Low-Code Modelling. An identity assignment is assumed when it is not present.

The UI execution environment may query both the `attributes` and `relationships` field, if it does not know whether the UI element in question references a relationship. After the first query, it does know this information and can memorize in which field the information for the UI elements was found. Another way would be to set a flag in

¹⁴The recommended way via the React hook `useStore()` does only expose the Redux store of the UI execution environment.

one of the configuration files, whenever an UI element references another resource via a relationship. To reduce the number of requests, the query parameter **include** is set accordingly (cf. Listing 2). Otherwise, the UI execution environment would have to explicitly query the resources referenced in **relationships** every single time.

The claim at the end of Step 4 in Chapter Low-Code Modelling is now justified in more detail. This approach is easily implementable, if the name of each attribute in the back-end representation of the domain model is globally unique. The attribute being searched for is first translated using the input mapping file. Then, the UI execution environment encounters the 'Center' and 'Ward' dropdown fields, which display attributes not belonging to the entity *Shift*. Thus, the UI execution environment requests the resources related to *Shift* and encounters the *Center* resource during this process, which contains the matching attribute. The resources related to *Center* are then queried, since neither *Shift* nor its related resources contain the translated attribute for the 'Ward' dropdown menu. Then, the missing attribute is found in *Ward* and displayed. The information gathered during this traversal is stored for future API requests and the **include** query parameter is used to reduce the number of API requests.

Validation Rules The possible ways to create validation rules have already been explained in Step 4 of Chapter Low-Code Modelling. Still, the transmission of more complex client-side validation rules has to be specified.

One possible way is to transmit the link to validation rules as meta data together with the API responses. The **meta** field can be used to transmit this information. The development team has to agree on the field name of the link to the validation rules, e.g., one can specify a **validation** field in **meta**.¹⁵

The **meta** keyword may appear at depth 1 of the API response along with the **validation** keyword to affect the entire resource or at least multiple UI components at once, or it may appear at depth 2 to affect a single resource instance. For the former situation, one can specify that the root path of the API endpoint contains the '(semi-)global' validation rules for its stored resources. This solution can be relevant, if the resource data is not fetched before the creation of the form.¹⁶ The latter situation can be used when the attributes of a particular resource instance are supposed to be read-only.

Tables and Sparse Fieldsets Sparse fieldsets for requesting table data are set according to the table configuration file. This step is implemented by chaining

```
fields[<resource>]=<attr>
```

parameters for the API request. This feature is disabled, if no table configuration file is provided and the whole set of attributes is then requested and displayed by the tables.

¹⁵**meta** contains an object with non-standard meta-information. Thus, the addition of the field **validation** is conform to JSON:API specification.

¹⁶This is not currently relevant for the Master/Detail skeleton from Chapter Low-Code Modelling, since resource instances are fetched before the form is shown. However, it can become relevant when developing the LCP further by adding further Master/Detail skeletons.

5. Implementation

Versioning Problem 2., (a), iii. can be addressed by choosing a versioning scheme. Changes in the back-end may invalidate the mapping file that is used to translate between front-end and back-end identifiers or the back-end may reject form inputs because of inconsistent validation rules. Semantic versioning can be used to indicate the compatibility of front-end and back-end. The *never remove, only add strategy* which JSON:API uses may also be a viable approach to ensure backward compatibility. It may be necessary to repeat the process from the previous chapter starting from Step 3 to update the front-end so that it is compatible with the new back-end version.

5.3.3 Evaluation

Problem 2, (b), i. and Problem 2, (b), ii. from Chapter Analysis are addressed by the specific features of the UI execution environment. JSON:API features are used to reduce communication between front-end and back-end. Additionally, these features are used to reduce configuration overhead for front-end developers. The MDSD method may offer support to handle Problem 2, (a), iii., but overall this problem can be better solved by the development team than by the MDSD tool-chain.

6 Conclusion

MDSD and related approaches are likely to become more and more important for software development as software becomes larger and more complex. The additional abstraction provided by MDSD and related approaches allows application developers to focus mostly on the problem domain. It also enables people lacking extensive programming background to construct applications, once a suitable MDSD method has been chosen and established. [46], [51]

For a practical development scenario, a set of requirements for a MDSD method based on JSON Forms and JSON:API was compiled in Chapter Analysis. Subsequently, a MDSD method was proposed in Chapter Implementation that meets all the identified requirements. This example also serves to illustrate the MDSD terminology from Chapter Overview of Model-driven Software Development.

There are further topics worth consideration, but not covered in this thesis. Verification and validation can be considered in context of MDSD to build dependable systems [28]. Additionally, version control systems such as Git are not appropriate for the version control of models, since they are focused on text-based artifacts. Advanced features such as the merging of two model versions may lead to an inconsistent model [9]. Furthermore, the intricate details of software for M2M- and M2T- transformations were not considered in this thesis. In summary, there are many aspects of MDSD that can be covered in much greater depth than shown in this thesis.

Since parts of the MDSD tool-chain outlined in Chapter Implementation is not implemented, the next step would be to actually build the proposed MDSD tool-chain. Moreover, it was implicitly assumed that entities from the initial domain model are not split up in the back-end representation. Future changes to the initial domain model may also introduce some difficulties for the MDSD method.

A partial implementation of the UI execution environment can be found in the GitHub repository for this thesis (https://github.com/alexander-korzec/MDSD_Bachelor_Thesis).

References

- [1] R. Acerbis, A. Bongio, M. Brambilla, and S. Butti, “Model-Driven Development Based on OMG’s IFML with WebRatio Web and Mobile Platform”, in *Engineering the Web in the Big Data Era*, ser. Lecture Notes in Computer Science, Cham: Springer Int. Publishing, 2015, pp. 605–608.
- [2] A. A. H. Alzahrani, “4GL Code Generation: A Systematic Review”, *Int. J. Adv. Comput. Science Appl.*, vol. 11, no. 6, 2020.
- [3] P. Bachmann, “Static and metaprogramming patterns and static frameworks: A catalog. an application”, in *Proc. 2006 Conf. Pattern Lang. Programs*, ser. PLoP ’06, New York, NY: ACM, Oct. 2006, pp. 1–33.
- [4] M. Baciková, J. Porubán, and D. Lakatos, “Defining Domain Language of Graphical User Interfaces”, in *2nd Symp. Lang., Appl. and Technologies*, ser. OpenAccess Series in Informatics (OASICS), vol. 29, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 187–202.
- [5] F. P. Basso, R. M. Pillat, T. C. Oliveira, F. Roos-Frantz, and R. Z. Frantz, “Automated design of multi-layered web information systems”, *J. Syst. and Software*, vol. 117, pp. 612–637, Jul. 2016.
- [6] C. Bernaschina, “ALMOsT.js: An Agile Model to Model and Model to Text Transformation Framework”, in *Web Engineering*, ser. Lecture Notes in Computer Science, Cham: Springer Int. Publishing, 2017, pp. 79–97.
- [7] C. Bernaschina, S. Comai, and P. Fraternali, “IFMLEdit.org: Model Driven Rapid Prototyping of Mobile Apps”, in *2017 IEEE/ACM 4th Int. Conf. Mobile Software Engineering and Syst. (MOBILESoft)*, May 2017, pp. 207–208.
- [8] A. C. Bock and U. Frank, “Low-Code Platform”, *Bus. and Inf. Syst. Engineering*, vol. 63, no. 6, pp. 733–740, Dec. 2021.
- [9] M. Brambilla, J. Cabot, and M. Wimmer, “Model-Driven Software Engineering in Practice, 2nd Edition”, *Synthesis Lectures on Software Engineering*, vol. 3, no. 1, pp. 1–207, Mar. 2017.
- [10] M. Brambilla, A. Mauri, and E. Umuhoza, “Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End”, in *Mobile Web Information Syst.*, ser. Lecture Notes in Computer Science, Cham: Springer Int. Publishing, 2014, pp. 176–191.
- [11] J. Cabot, “Positioning of the low-code movement within the field of model-driven engineering”, in *Proc. 23rd ACM/IEEE Int. Conf. Model Driven Engineering Lang. and Syst.: Companion Proc.* 76, New York, NY: ACM, Oct. 2020, pp. 1–3.
- [12] K. Czarnecki and S. Helsen, “Feature-based survey of model transformation approaches”, *IBM Syst. J.*, vol. 45, no. 3, pp. 621–645, 2006.
- [13] K. Czarnecki, “Overview of Generative Software Development”, in *Unconventional Programming Paradigms*, ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2005, pp. 326–341.

- [14] L. P. da Silva and F. Brito e Abreu, “Model-driven GUI generation and navigation for Android BIS apps”, in *2014 2nd Int. Conf. Model-Driven Engineering and Software Development (MODELSWARD)*, Jan. 2014, pp. 400–407.
- [15] V. De Castro, E. Marcos, and J. M. Vara, “Applying CIM-to-PIM model transformations for the service-oriented development of information systems”, *Information and Software Technology*, vol. 53, no. 1, pp. 87–105, Jan. 2011.
- [16] V. Deufemia, C. D’Souza, and A. Ginige, “Visually modelling data intensive web applications to assist end-user development”, in *Proc. 6th Int. Symp. Visual Information Communication and Interaction*, ser. VINCI ’13, New York, NY, USA: ACM, Aug. 2013, pp. 17–26.
- [17] Eclipse Foundation. (Aug. 2022). Eclipse IDE, [Online]. Available: <https://eclipseide.org/> (visited on 08/08/2022).
- [18] —, (Aug. 2022). Eclipse Modelling Project, [Online]. Available: <https://www.eclipse.org/modeling/> (visited on 08/08/2022).
- [19] —, *Eclipse modelling framework*. [Online]. Available: <https://www.eclipse.org/modeling/emf/> (visited on 08/08/2022).
- [20] EclipseSource. (Aug. 2022). JSON Forms, [Online]. Available: <https://jsonforms.io/> (visited on 08/08/2022).
- [21] M. EL Omari, M. Erramdani, and A. Rhouati, “A Model Driven Approach for Generating Angular 7 Applications”, *Int. J. recent contributions engineering, science & IT*, vol. 8, no. 2, pp. 36–45, 2020.
- [22] R. France and B. Rumpe, “Model-driven Development of Complex Software: A Research Roadmap”, in *Future Software Engineering (FOSE ’07)*, IEEE, 2007, pp. 37–54.
- [23] G. Dennis. (Jun. 2022). JSON Schema: A Media Type for Describing JSON Documents, [Online]. Available: <https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-01> (visited on 08/08/2022).
- [24] L. Gaouar, A. Benamar, and F. T. Bendimerad, “Model Driven Approaches to Cross Platform Mobile Development”, in *Proc. Int. Conf. Intelligent Information Processing, Security and Adv. Communication*, ser. IPAC ’15, New York, NY: ACM, Nov. 2015, pp. 1–5.
- [25] A. Gerasimov, J. Michael, L. Netz, and B. Rumpe, “Agile Generator-Based GUI Modeling for Information Systems”, in *Modelling to Program*, ser. Communications in Comput. and Information Science, Cham: Springer Int. Publishing, 2021, pp. 113–126.
- [26] M. Hamdani, W. H. Butt, M. W. Anwar, and F. Azam, “A Systematic Literature Review on Interaction Flow Modeling Language (IFML)”, in *Proc. 2018 2nd Int. Conf. Management Engineering, Software Engineering and Service Sciences*, ser. ICMSS 2018, New York, NY: ACM, Jan. 2018, pp. 134–138.

- [27] R. Hebig, C. Seidl, T. Berger, J. K. Pedersen, and A. Wąsowski, “Model transformation languages under a magnifying glass: A controlled experiment with Xtend, ATL, and QVT”, in *Proc. 2018 26th ACM Joint Meeting European Software Engineering Conf. and Symp. Foundations Software Engineering*, ser. ESEC/FSE 2018, New York, NY: ACM, Oct. 2018, pp. 445–455.
- [28] A. Hovsepyan, D. V. L, S. O. D. Beeck, W. Joosen, G. Rangel, J. Fern, and E. Briones, “Model-driven software development of safety-critical avionics systems: An experience report”, in *Proc. 1st Int. Workshop Model-Driven Development Processes and Practices (MD2P2)*, 2014, pp. 28–37.
- [29] H. Hoyos Rodriguez and B. Sanchez Piña, “JSOI: A JSON-Based Interchange Format for Efficient Model Management”, in *2019 ACM/IEEE 22nd Int. Conf. Model Driven Engineering Lang. and Syst. Companion (MODELS-C)*, Sep. 2019, pp. 259–266.
- [30] IETF. (Aug. 2022). JSON Schema, [Online]. Available: <https://json-schema.org/> (visited on 08/08/2022).
- [31] L. Jonathan. (Nov. 2017). BNF Grammars for SQL-92, SQL-99 and SQL-2003, [Online]. Available: <https://ronsavage.github.io/SQL/> (visited on 08/08/2022).
- [32] S. Jörges, *Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach*. Springer, Jan. 2013.
- [33] B. W. Kernighan, *The C Programming Language / Brian W. Kernighan ; Dennis M. Ritchie*. 2. ed., 6. [print.] Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [34] J. Kramer, “Is abstraction the key to computing?”, *Commun. ACM*, vol. 50, no. 4, pp. 36–42, Apr. 2007.
- [35] G. Macher and C. Kreiner, “Model transformation and synchronization process patterns”, in *Proc. 20th European Conf. Pattern Languages Programs*, ser. EuroPLoP ’15, New York, NY: ACM, Jul. 2015, pp. 1–11.
- [36] A. P. F. Magalhaes, A. M. S. Andrade, and R. S. P. Maciel, “Model driven transformation development (MDTD): An approach for developing model to model transformation”, *Information and Software Technology*, vol. 114, pp. 55–76, Oct. 2019.
- [37] OMG. (Jun. 2014). OMG Document – Ormsc/14-06-01 (MDA Guide Revision 2.0), [Online]. Available: http://themeforest.net/user/dan%5C_fisher (visited on 08/08/2022).
- [38] —, (Feb. 2015). Interaction Flow Modeling Language, [Online]. Available: <https://www.omg.org/spec/IFML/About-IFML/> (visited on 08/08/2022).
- [39] M. Overeem and S. Jansen, “An Exploration of the ‘It’ in ‘It Depends’: Generative versus Interpretive Model-Driven Development:” in *Proc. 5th Int. Conf. Model-Driven Engineering and Software Development*, Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2017, pp. 100–111.
- [40] M. Overeem, S. Jansen, and S. Fortuin, “Generative versus Interpretive Model-Driven Development: Moving Past ‘It Depends’”, in *Model-Driven Engineering and Software Development*, ser. Communications in Comput. and Information Science, Cham: Springer Int. Publishing, 2018, pp. 222–246.

- [41] G. Paolone, M. Marinelli, R. Paesani, and P. Di Felice, “Automatic Code Generation of MVC Web Applications”, *Comput.*, vol. 9, no. 3, p. 56, Sep. 2020.
- [42] C. Preschern, N. Kajtazovic, and C. Kreiner, “Applying patterns to model-driven development of automation systems: An industrial case study”, in *Proc. 17th European Conf. Pattern Lang. Programs*, ser. EuroPLOP ’12, New York, NY: ACM, Jul. 2012, pp. 1–10.
- [43] T. Preston-Werner. (Aug. 2022). Semantic Versioning Specification, [Online]. Available: <https://semver.org/> (visited on 08/08/2022).
- [44] S. Raemaekers, A. van Deursen, and J. Visser, “Semantic versioning and impact of breaking changes in the Maven repository”, *J. Syst. and Software*, vol. 129, pp. 140–158, Jul. 2017.
- [45] A. Rauf, M. Ramzan, M. A. B. U. Rahim, and A. A. Shahid, “Extending UML to model GUI: A new profile”, in *2010 The 2nd Int. Conf. Comput. and Automation Engineering (ICCAE)*, vol. 1, Feb. 2010, pp. 349–353.
- [46] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, “Supporting the understanding and comparison of low-code development platforms”, in *2020 46th Euromicro Conf. on Software Engineering and Adv. Appl. (SEAA)*, Aug. 2020, pp. 171–178.
- [47] H. Stachowiak, *Allgemeine Modelltheorie / Herbert Stachowiak*. Vienna: Springer, 1973.
- [48] T. Stahl and M. Völter, *Model-Driven Software Development: Technology, Engineering, Management*. Chichester, England ; Hoboken, NJ: John Wiley, 2006.
- [49] Y. Sun, Z. Demirezen, M. Mernik, J. Gray, and B. Bryant, “Is my dsl a modeling or programming language?”, in *Domain-Specific Prog. Development*, 2008, p. 4.
- [50] t2informatik. (Aug. 2022). t2informatik, [Online]. Available: <https://t2informatik.de/> (visited on 08/08/2022).
- [51] H. Tufail, F. Azam, M. W. Anwar, and I. Qasim, “Model-Driven Development of Mobile Applications: A Systematic Literature Review”, in *2018 IEEE 9th Annu. Information Technology, Electronics and Mobile Communication Conf. (IEMCON)*, Nov. 2018, pp. 1165–1171.
- [52] M. Völter, “Best practices for dsls and model-driven development”, *J. Object Technology*, vol. 8, no. 6, pp. 79–102, 2009.
- [53] WebRatio. (Nov. 2017). IFML Quick Reference Card, [Online]. Available: <https://my.webratio.com/learn/lmsservlet/321/1/IFML%5C%20Quick%5C%20Reference%5C%20Card.pdf> (visited on 08/08/2022).
- [54] N. Wirth, *Algorithms and Data Structures Are Programs*. Ser. Prentice-Hall Series in Automatic Computation. Englewood Cliffs, N.J: Prentice-Hall, 1976.
- [55] Y. Katz, D. Gebhardt, G. Sullice. (Aug. 2022). JSON:API, [Online]. Available: <https://jsonapi.org/> (visited on 08/08/2022).

References

- [56] N. Yousaf, F. Azam, W. H. Butt, M. W. Anwar, and M. Rashid, “Automated Model-Based Test Case Generation for Web User Interfaces (WUI) From Interaction Flow Modeling Language (IFML) Models”, *IEEE Access*, vol. 7, pp. 67 331–67 354, 2019.

A Appendix

```

export interface RemoteLocation {
  protocol?: string,
  port: number,
  address: string
}

export interface Resource {
  type: string,
  service: RemoteLocation,
  schemaPath: string,
  uischemaPath?: string,
  createPath?: string,
  remotePath?: string,
  updatePath?: string,
  deletePath?: string
}

export interface ResourceFile {
  resources: Resource[]
}

```

Listing 4: TS definition for resource files. A resource file is an array of resources to be displayed by the UI execution environment. A resource is at least specified by the attributes **type** (= unique identifier for resource), **service**, which gives information about the location of the API endpoint containing the resource, and **schemaPath**, the specific path of the JSON Forms schema for the given resource. For instance, the UI execution environment will send a GET request to `protocol://address:port/type` to retrieve the concrete instances of a resource. Further attributes can be specified for the UI layout (**uischemaPath**) or if CRUD operations have different API paths deviating from **type**.

```

{
  "type": "HorizontalLayout",
  "elements": [
    {
      "type": "VerticalLayout",
      "elements": [
        {
          "type": "Control",
          "label": "Staff data",
          "scope": "#/properties/staff_mgmt/properties/staff_data"
        },
        {
          "type": "Control",
          "label": "Address",

```

```

        "scope": "#/properties/staff_mgmt/
        properties/address"
    },
    {
        "type": "Control",
        "label": "ID's",
        "scope": "#/properties/staff_mgmt/
        properties/ids"
    }
]
},
{
    "type": "VerticalLayout",
    "elements": [
        {
            "type": "Control",
            "label": "Work",
            "scope": "#/properties/staff_mgmt/
            properties/work"
        },
        {
            "type": "Control",
            "label": "Phone",
            "scope": "#/properties/staff_mgmt/
            properties/phone"
        },
        {
            "type": "Control",
            "label": "Comments",
            "scope": "#/properties/staff_mgmt/
            properties/comments"
        }
    ]
}
]
}

```

Listing 5: Example of a JSON Forms UI schema.

Staff

Staff data

Last name: Alviar
 First name(s): Paul
 Form of address: Mr.
 Title:
 Birth date:
 Nationality: British
 User name: Paul Alviar
 Address: Street: 64 Orchard road, Postcode/City: ME 76289 Mount Everest, State/Country: UK
 ID's: Staff-ID 1: 52CB4646, Staff-ID 2: DE4EF0

Work

Profession:
 Function: Nurse
 Position: Nurse Director
 Department: Renal
 Phone: Telephone Work: 05403 6360589, Private 1:, Private 2:, Fax:
 Comments:

Last Name	First Name(s)	Form of address	Title	Birthdate	Nationality	User name	Profession	Function
Alviar	Paul	Mr.			British	Paul Alviar	Nurse	Nurse
Baloun	David	Mr.		26/05/1959	british	David Baloun	dialysis support worker	Nurse
Batenga	Mercy	Mrs.		25/04/1964	Indian	Mercy Batenga	Nurse	Nurse
Challinor	Randi	Mr.		28/06/1973	British	Randi Challinor	Senior Dialysis Nurse	Nurse
Concepcion	Patience	Mrs		03/04/1976	British	Patience Concepcion	Nurse	Nurse
Dixon	Kathryn	Ms.		08/07/1957	British	Kathryn Dixon	Nurse	Nurse
Jones	Lisa	Mrs	Dr.	16/12/1987	British	Lisa Jones	Physician	Physician

Medications Catalog

Details

Trade name: Alfacalcidol 0.25mcg, Manufacturer: generic
 Category: , National code: , ATC code:
 Package data: 30, Preparation form: capsule, Dosing strength: 0.25, Dosing unit: ug
 Remarks: , Active / Obsolete: Active

Trade name	Category	Manufacturer	Package data	Preparation form	Dosing strength	Dosing unit	National code	ATC code
Alfacalcidol 0.25mcg		generic	30	capsule	0.25	ug		
Alfacalcidol 1.0mcg		generic	30	capsule	1	ug		
Alfacalcidol 2.0mcg Amp		generic	10	ampule	2	ug		
Aranesp (Darbopoeitin...)	Erypo	Amgen	6	pre-filled syringe	130	ug		
Aranesp (Darbopoeitin...)	Erypo	Amgen	6	pre-filled syringe	300	ug		
Aranesp (Darbopoeitin...)	Erypo	Amgen	6	pre-filled syringe	40	ug		
Aranesp (Darbopoeitin...)	Erypo	Amgen	6	pre-filled syringe	100	ug		
Aranesp (Darbopoeitin...)	Erypo	Amgen	6	pre-filled syringe	60	ug		
Aranesp (Darbopoeitin...)	Erypo	Amgen	6	pre-filled syringe	10	ug		
Aranesp (Darbopoeitin...)	Erypo	Amgen	6	pre-filled syringe	80	mg		
Aranesp (Darbopoeitin...)	Erypo	Amgen	6	pre-filled syringe	30	ug		
Aranesp (Darbopoeitin...)	Erypo	Amgen	6	pre-filled syringe	20	ug		
Chlorphenamine	generic		28	tablet	4	mg		
CitraFlow		MedXL Inc. Montreal, ...		pre-filled syringe	46.7	%	10682686004113	38143-1

Center, wards, rooms, beds

Organisation

- 8. Braun Avitum Renal ...
 - Ward 1
 - Bay 1 Bed 1
 - Bay 1 Bed 2
 - Bay 1 Bed 3
 - Bay 1 Bed 4
 - Bay 1 Bed 5
 - Bay 1 Bed 6
 - Bay 1 Bed 7
 - Bay 1 Bed 8
 - Bay 2 Bed 1
 - Bay 2 Bed 2
 - Bay 2 Bed 3
 - Bay 2 Bed 4
 - Bay 2 Bed 5
 - Bay 2 Bed 6
 - Bay 2 Bed 7
 - Bay 2 Bed 8
 - Ward 2
 - Bay 3 Bed 1

Details

In order to add a center, select <Organisation> in the tree and press <Add center>.
 In order to add a ward, select a center in the tree and press <Add ward>.
 In order to add a room, select a center or a ward in the tree and press <Add room>.
 In order to add a bed, select a ward or a room in the tree and press <Add bed>.

Name:
 Organisation:
 Comment:
 (Name cannot be modified)

Figure 9: Example screenshots of a management system for multiple hospitals. The screenshots show an UI for managing staff members, an overview of a medication catalog, and an interface for the management of Location objects, e.g., Center, Ward, Room, and Bed. In the first screenshot, the user can set a number of attributes for staff members. The next screenshot shows domain-specific information that needs to be filled out by a domain expert or more specifically by a medical expert, i.e., a doctor. The last screenshot is a configuration screen for Location objects, which is perhaps not used often in the application.

A. Appendix

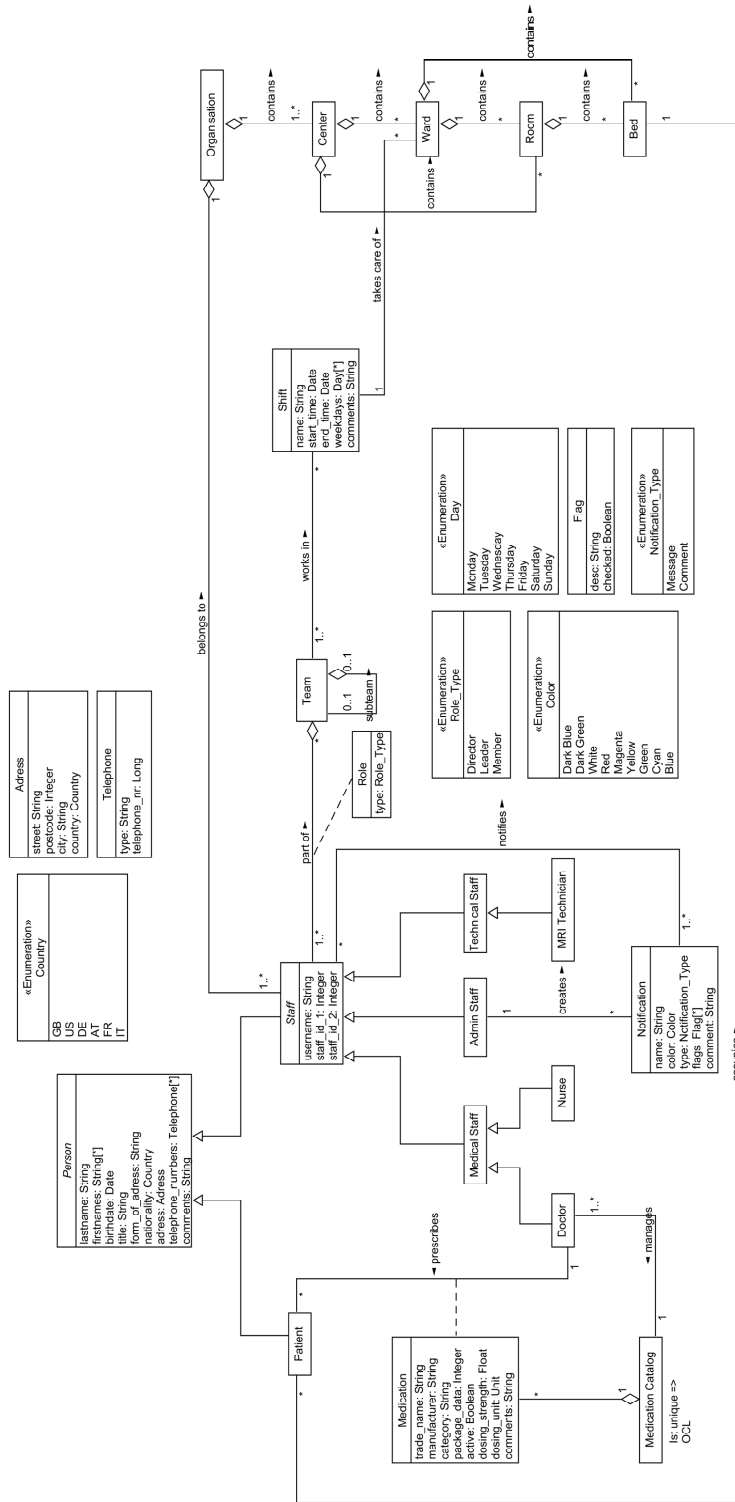


Figure 10: An example for a static conceptional model for a health care management system. Figure 9 does not fully represent the information from this UML class diagram. Note that this model is fairly general and does not include any domain-specific UML stereotypes.

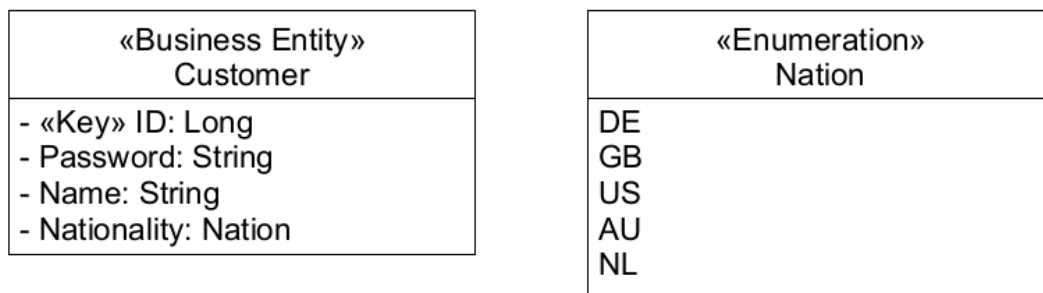


Figure 11: UML stereotypes are enclosed in so-called guillemets, i.e., the characters « and ». «Business Entity» and «Key» represent domain-specific abstractions and can be replaced or expanded in further model transformations. UML also provides a set of predefined general stereotypes such as «Enumeration» to define a data type and the corresponding collection of values that the data type can take.

A. Appendix

STAFF

MEDICATION

NOTIFICATION

SHIFT

Staff

Manage Item		Data											
Edit	Delete	Last name	First name(s)	Form of address	Title	Birth date	Nationality	User name	Street	Postcode/City	State/Country	Staff-ID 1	Staff-ID 2
		Alvar	Paul	Mr			British	palvar	64 Orchard Road	ME 76Z89 Mount Everest	UK	52CB4646	DE4EF0
		Batenga	Mercy	Mrs.		1964-04-25	Indian	mpatenga	62 Orchard Road	ME 76Z89 Mount Everest	UK	11FF2088	CC1CC2
		Challinor	Randi	Mr.		1973-06-28	British	rchallinor	61 Orchard Road	ME 76Z89 Mount Everest	UK	98AF3212	AC12AF43
		Conception	Patience	Mrs.		1976-04-03	British	pconception	32 Peanuts Road	ME 76Z89 Mount Everest	UK	66CD1212	CD12DC67
		Dixon	Kathryn	Ms.		1957-07-08	British	kdixon	98 Peanuts Road	ME 76Z89 Mount Everest	UK	96AC21AA73	AF61DE
		Jonas	Lisa	Mrs.	Dr.	1987-12-16	British	ljonas	13 Peanuts Road	ME 76Z89 Mount Everest	UK	53DE19EC	CE22DA

CREATE NEW ENTRY +

STAFF

MEDICATION

NOTIFICATION

SHIFT

Staff

RETURN TO PREVIOUS SCREEN ←

Staff data

Last name

First name(s)

Form of address

Title

Birth date

Nationality

User name

Address

Street

Postcode/City

State/Country

ID's

Staff-ID 1

Staff-ID 2

Work

Profession

Function

Position

Department

Phone

Telephone Work

Private 1

Private 2

Fax

Comments

Comments

CREATE NEW ENTRY >

Figure 12: Basic UI output for *Staff* resource produced by the UI execution environment.