Universiteit Leiden
The Netherlands

Computer Science & Economics

UML class models as first-class citizen: Metadata at design-time and run-time

Ralph Driessen

Supervisors:
Dr. G.J. Ramackers (first supervisor) &
Dr. A.W. Laarman (second supervisor)

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

23/08/2020

Acknowledgements

This research is done as part of a bachelor thesis at LIACS, the computer science institute of Leiden University, and with great thanks to dr. G.J. Ramackers and dr. A.W. Laarman for supervising and contributing to the result.

Abstract

Developing large-scale software systems requires a well-defined approach to the design of such complex systems. The Unified Modeling Language (UML), defined by the Object Management Group standards consortium, provides a standard way of expressing such designs, and is supported by a large number of commercially available tools. These tools allow for the visual expression of models at design-time, and subsequent generation of skeleton code that is then fully implemented, compiled and deployed to provide applications . While such a Model Driven Architecture (MDA) approach shortens development time in the design phase of a project, it can actually slow maintenance tasks, such as implementing new feature requests, which are performed when the system is operational. To implement feature requests, the design and implementation phases have to be re-entered and executed in full. This is a costly and time-consuming process, requiring expert developers.

In order to address these issues, we propose to augment the classic MDA approach by using UML models not just as design-time artifacts, but by also deploying them as run-time artifacts alongside the application. By structuring applications in such a way that the UML model functions as metadata that is reflected upon at run-time by a "model interpreter" component, the functionality of the application can be rapidly evolved by directly modifying the model metadata in the system whilst it is running. The feasibility of this approach is demonstrated in a prototype implementation that focuses on UML Class models as an example of metadata that defines the structural aspects of a system. This implementation allows the user to define UML class models at design-time, and additionally enables editing of that metadata at run-time by the developer or user (by means of the same editing screens being accessible). Due to model interpretation in the run-time environment developed to support the deployed models, the application immediately changes functionality in terms of showing the new fields and links. In order to facilitate this, the UML Class model has been extended with the concepts of "domain model" and "application model". In addition, a limited dynamic aspect has been added to the meta data in the form of "model scripting": the user can add snippets of Python code to methods which are stored in the model, and executed at run-time. This thesis discusses lessons learned, and outlines future work in extending the set of UML models deployed to have more dynamic aspects, such as state machines, and activities, which would require a more comprehensive run-time model execution engine.

Contents

1 Introduction

1.1 Context

In order to increase efficiency and functional accuracy in software development, industry often utilises the Model Driven Architecture (MDA) approach, sometimes also referred to as Model Driven Development (MDD). MDA makes abstract models of the system-under-development as a leading artefact in software development, using them to generate (parts of) the source code for the implementation. Several models can be used to describe the system under development from various perspectives: UML[1] class models describe the structural aspects, UML state machines, activity models and interactions describe behavioral(dynamic) aspects.

In practice, code generation also requires the availability of metadata[2] other than UML models, such as, configuration files for deployment, BPEL files for worklows, etc. The ultimate vision of MDD is to make all models and other design-time data, as leading artefacts in the development process. This means that the source code can be generated from this metadata at any time. This is a non-trivial exercise, as feature requests and maintenance tasks (e.g., bug resolving) lead to a continuous evolution of both the codebase and other design-time artifacts.

1.2 Problem statement

Currently, when a business or end user of an application requests a new feature there is a number of process steps to be completed to implement the feature. The problem here is that these steps are time consuming and require expert developers to perform them. While the MDA approach shortens development time in the design phase of the project, it can actually slow maintenance tasks, such as implementing new feature requests, which are performed when the system is operational. In most businesses the process of implementing feature requests will mostly will take between a few weeks and a few months. Often the requirements will have changed in that time or the result is not what the original business expert or end-user had hoped for. To complete feature requests, the design phase has to be re-entered and executed in full. The design phase consists of multiple steps from the "waterfall model" as illustrated in Figure 1.

[1] https://www.omg.org/spec/UML

[2] metadata: 'data about data'

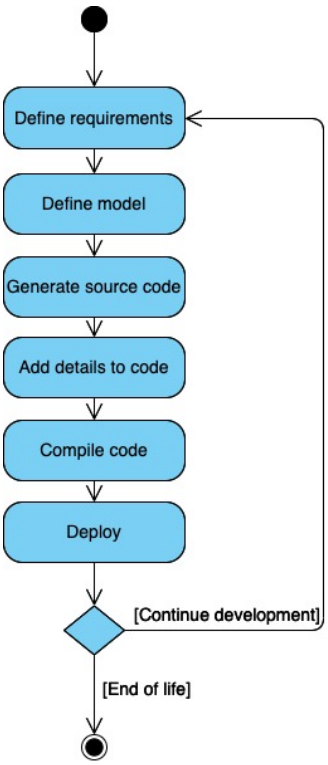

Figure 1: UML activity diagram of the current software development flow

1.3 Approach

This thesis proposes to make UML class models first-class citizens by deploying them as metadata at run-time. Our vision is that the metadata is deployed together with the application and can be modified to change the behavior of the system while it is running. This creates a different workflow than the one shown in Figure 1. This new workflow, as illustrated in Figure 2, lacks the aspects of regenerating the code, compiling and deploying the codebase. In this flow, the metadata is continuously interpreted during the definition of the model and the annotation of the model to modify the application at run-time.

Because UML is a widely known language and supported in many existing applications we chose to focus on UML models, specifically on UML class models as core structural aspects of the UML specification. To demonstrate that it is indeed possible to update a running system based on modifications to the UML class model, we implement a prototype system that contains a UML class model definer and an interpretation environment that performs reflection on the UML class model using its metadata. This system is used to build a simple example enterprise application. The prototype application should thus support defining a UML class model and interpreting that UML class model metadata to generate an enterprise application. To realise this approach we propose to use a small metamodel to capture the model metadata and to implement a prototype application that supports the evolution of software through continuous metadata modification and interpretation. An overview of the main concepts of the UML class metamodel is illustrated in

Figure 3.

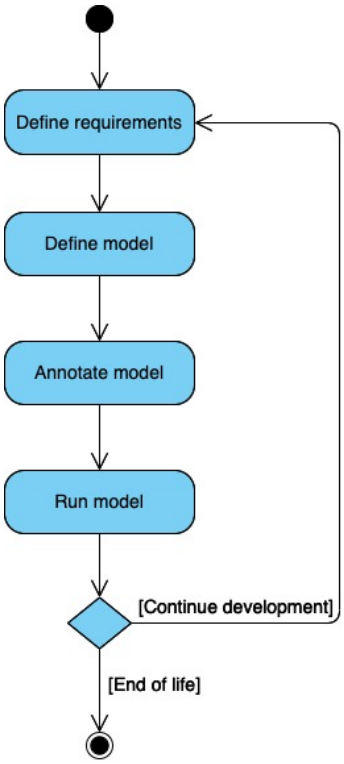

Figure 2: UML activity diagram of the software development flow with the proposed approach

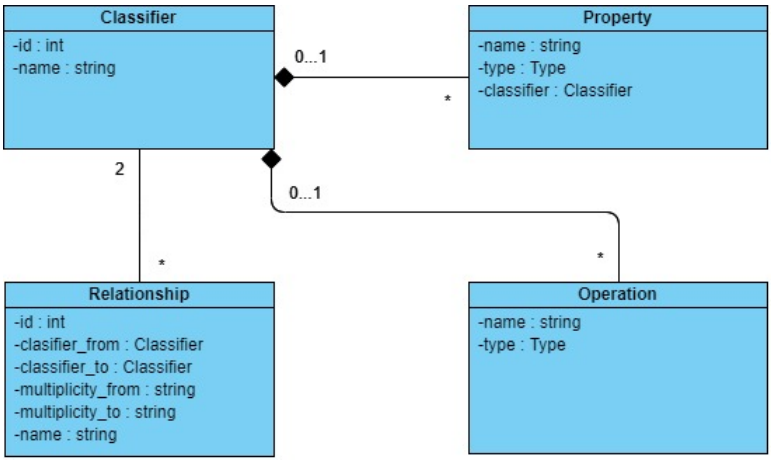

Figure 3: UML class diagram of the overview of the main concepts of the UML metamodel

1.4 Result

We implemented a UML class model definer and run-time metadata interpreter in Python using the Django framework. This is done by creating a web application that stores the metadata in a SQLite database. This supports creating a UML class model by creating classifiers, properties, operations and relationships. It also supports creating applications that are able to use the defined classifiers and perform CRUD (Create, Read, Update, Delete) operations on those objects. At run-time it is possible to edit the UML class model by adding new classifiers, properties, operations or relationships by using the same class model definer (as depicted in Figure 4).

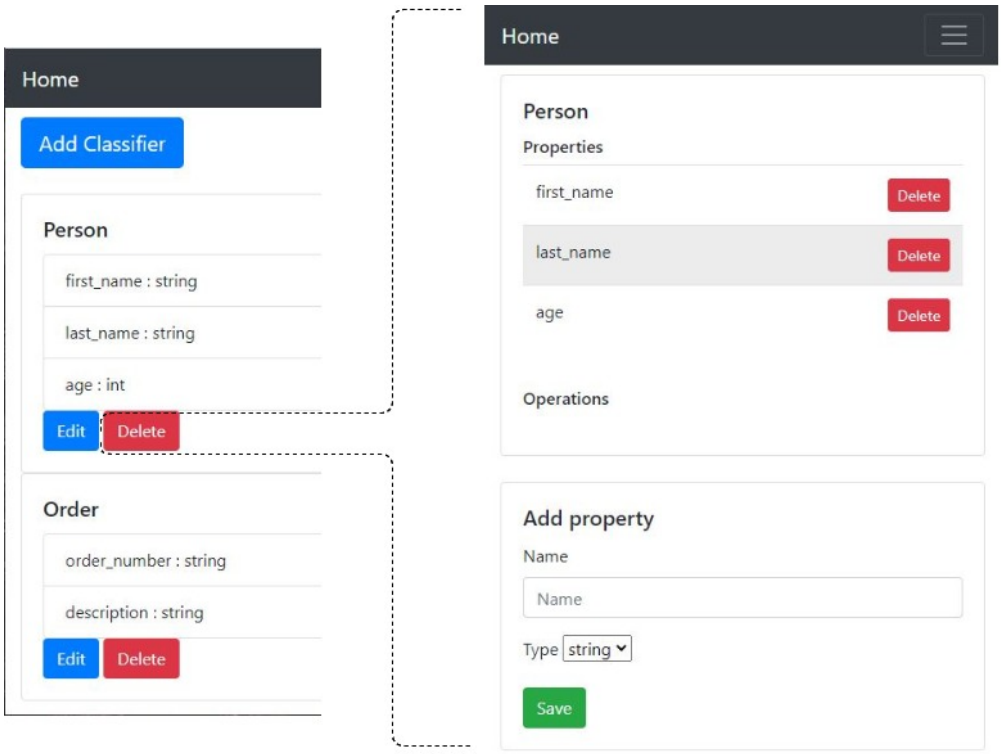

Figure 4: Model definer screenshots

These changes then directly evolve the running applications that make use of the classifiers that were affected by the changes. This is possible because the run-time environment contains a model interpreter. The run-time environment is structured as illustrated in Figure 5. In this illustration we see that the user can talk to a web application, containing the model definer and the run-time applications. The model definer saves the metadata in objects when a changes is made. The model interpreter generates source code for the run-time application under the hood (not visible for the user). The run-time applications are able interact with the classifiers from the metamodel because of the work of the model interpreter.

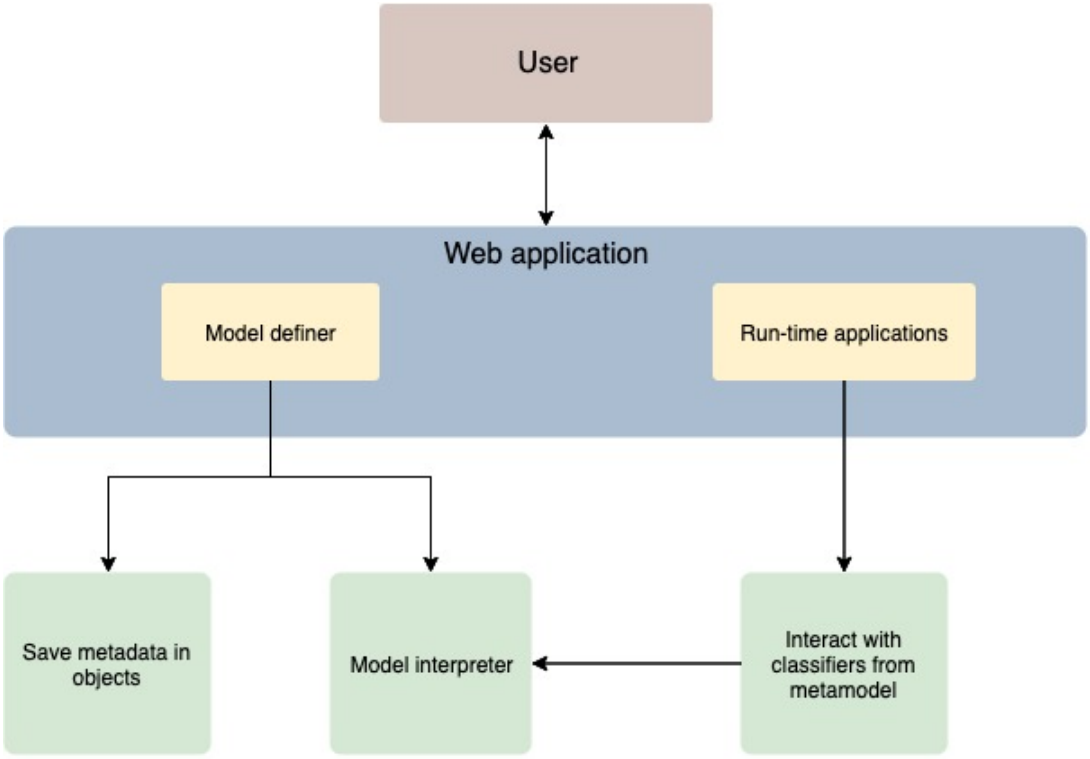

Figure 5: Run-time environment of the prototype implementation

1.5 Overview

Chapter 2 provides a background of the system development concepts central to this thesis, and contains definitions of the terminology. In addition a number of use cases to which this approach is relevant is described based on the background of these concepts. Chapter 3 describes the previous work in the field of Model Driven Development and Model Driven Architecture, and how this differs from the approach outlined in this research. In addition, this chapter describes current development tools, and their market segments. Chapter 4 contains information on the details of the approach suggested in this thesis and how a resulting application should work. Chapter 4 also contains the details of the model used, and how this is translated to be used in the run-time interpretation of the meta data. Chapter 5 contains information on how the implementation is set up. Chapter 6 illustrates how such an application would look like from a end user's point of view and demonstrates how to use it. Chapter 7 contains all aspects that are not completely or totally not implemented in the created prototype application. This chapter could give guidance to a potential continuance of research on this topic and provides all of the currently thought of limitations and shortcomings.

2 Background

The following concepts are central to the approach presented by this thesis:

- OMG UML
- Model Driven Architecture
- Class model metadata
- First-class citizen

The approach presented in this thesis is build upon these concepts and they thus play an important role in the definition of this approach.

2.1 OMG UML

The Object Management Group (OMG) created the Unified Modeling Language (UML) to have a standardized modeling language which can be used to visualize system designs. UML contains a lot of different definitions for diagrams, such as:

- Class diagram
- Use case diagram
- Activity diagram

We use the UML definition of the class diagram to create our own metamodel that contains parts of the class diagram definition plus some extensions. Class diagrams are used to define the class objects a system consists of. It shows the system's classes, with their attributes and operations as well as the relations between mutiple classes. A simple example of a class diagram is illustrated in Figure 6.

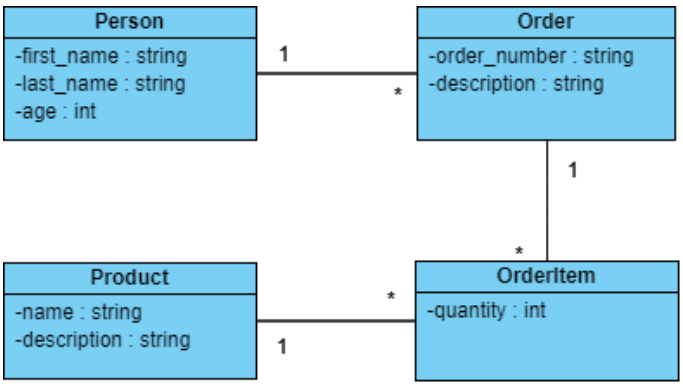

Figure 6: UML class diagram example

Figure 6 shows that a system has four classes, viz. Person, Order, OrderItem and Product, and also lists their attributes and relationships. UML class diagrams are orginally Platform Independent Models, which are models that can be defined without reference to a platform, and therefore without describing a particular form of technology[9]. However, adding some aspects that are only possible in certain languages or frameworks can make them in fact platform specific.

2.2 Model Driven Architecture

The above terms are widely used in different ways. As for this research, 'Model Driven Architecture' is interpreted as defining the architecture of an application by interpreting a model, or in this case a UML Class Diagram. The MDA is defined and trademarked by OMG[21]. The MDA defined by OMG is used as a basis in many different researches[20],[10],[13]. The term MDA is entangled with MDD (Model Driven Development) as the term is mostly used for development based on models, rather than only using models to define the architecture.

A lot of work within the software development field is done on model driven architecture or rather on approaches based on the model driven architecture. Especially within business context, the model driven architecture based approaches are very popular. This is because business concepts can be directly translated to models. This leads to a codebase focused on the specific business aspects it is created for and thus an application that is an exact fit to the business processes.

The approach we present is also based on the model driven architecture. However, where classic MDA has no role fore models at run-time. The approach we present augments the classic MDA approach by using UML models not just as design-time artefacts, but by also deploying them at run-time artifacts alongside the application. This creates an environment where software can be rapidly evolved by changing UML class models at run-time.

2.3 Class model metadata

The term metadata is mostly defined as 'data about data'. Metadata is used as the data about a model, where the model is a model defined according to an extended version of the UML standards. This model is used to save the metadata of a class diagram model that users can define. The metamodel used for this approach, as illustrated in Figure 3, is an abstraction of the UML class diagram specification. When looking back at the example UML class diagram shown in Figure 6, the way that the given model's metadata is saved is done according to the structure of the metamodel shown above. The classes Person, Order, OrderItem and Product are saved as Classes which extend the Classifier class, their properties are saved in Property objects which are linked to a classifier and their relationships are saved in, in this case, Relationship objects that are linked to both the classes and contain the information such as multiplicity. An example of the resulting metadata can be illustrated in many types of data, in Listing 1 is shown how the metadata of the class model from Figure 6 could be expressed in JSON format.

```
1 {
2     "classifiers": [
3         {
4             "id": 1,
```

```
            "name": "Person"
        },
        {
            "id": 2,
            "name": "Order"
        },
        {
            "id": 3,
            "name": "OrderItem"
        },
        {
            "id": 4,
            "name": "Product"
        }
    ],
    "properties": [
        {
            "id": 1,
            "name": "first_name",
            "type": "string",
            "classifier_id": 1
        },
        ...
        {
            "id": 4,
            "name": "order_number",
            "type": "int",
            "classifier_id": 2
        },
        ...
    ],
    "relationships": [
        {
            "id": 1,
            "name": "Person orders Order",
            "classifier_from_id": 2,
            "classifier_to_id": 1,
            "multiplicity_from": "*",
            "multiplicity_to": "1"
        },
        ...
    ]
}
```

Listing 1: "Metadata example in JSON"

2.4 First-class citizen

The concept 'first-class citizen' could be the most significant concept because it defines the run-time use of metadata which is the core of the approach we present. The term 'first-class citizen' is defined as 'an entity which supports all the operations generally available to other entities' where the operations mentioned are:

- Being passed as an argument
- Being returned from a function
- Being modified
- Being assigned to a variable

In the proposed approach the metadata of a model is a first-class citizen. This means that all the metadata entities have the availability of the above described operations. The first-class citizenship is not only at design-time but also at run-time. This means that the properties of first-class citizenship specified above apply at run-time too. The metadata can be changed at run-time and also be used by the applications that use it. The use of metadata as a first-class citizen is what makes this approach 'metadata driven', which simply means that the application is driven by metadata. An example of an existing metadata driven technology is BPEL(Business Process Execution Language)[11]. Here the metadata of business processes is used for multiple applicational activities such as data flow and manipulation. These metadata driven approaches are also a great factor in No Code Platforms(further mentioned in 3.2). These platforms also use metadata as a way to define the workings of an application. The difference with the approach presented by this thesis is that the metadata in other approaches is most oftenly not deployed at run-time and is not a subset of the widely known modeling language UML.

2.5 Use cases

A development approach based on the described concepts above will be an environment that allows for rapid software evolution. This creates the possibility for this approach to be used in several ways. The most important use cases are discussed in this chapter.

2.5.1 Large Scale Enterprise Systems

When this approach would be expanded to full on implement the UML specification, it can be used for Large Scale Enterprise Systems. Most systems like this are primarily used for things as order handling or other, quite simplistic, operations. The benefit of this setup in contrast to a extremely complex system as SAP, is that because every change in the models are handled runtime, the agility is much higher. Every small change can be done within minutes without having to dive into the source code. Even very specific operations can be implemented by adding a custom operation to a class. The disadvantage of this system would be that it is very generic and is not designed specifically for a company. However, because the system is based on the models the user provides, the base architecture would be an exact fit to the wishes of the user.

2.5.2 Rapid prototyping and application development

Rapid prototyping via web based systems provide a better availability for rapid prototyping tools[16]. This approach provides a service to rapidly create (prototype) applications by only creating a UML models. This could higly reduce the cycle and cost of the standard development flow.

2.5.3 Model validation

One of the most viable possibilities for rapid prototyping would be prototyping a domain model to get feedback from a customer about the structure. This way the customer is engaged in reviewing the work in a early stage of the project cycle and it is less likely to get miscommunication issues. If the customer notices a mistake in the structure it can easily be edited at runtime. This could be a perfect setup for a live editing session with customers to find all possible mistakes in the domain model.

3 Related Work

3.1 Metadata mapping

In [19], the author proposes a vision for the evolution of MDA. The near term vision is that "of an environment in which efficient and nearly seamless interoperability between diverse applications, tools and databases is achieved through the interchange of shared models". The use of model instances to save and use metadata as done in the approach we propose is also what they defined in their vision of MDA evolution. They also define a structure of how different language should map to each other.

The difference between this work and our work is that their vision of using objects to save and distribute model metadata is something we have implemented in our approach. Our work thus adds the implementation of aspects from their vision for the evolution of MDA. Because this work already overlaps with our approach, the outcomes of their (future) research can be beneficial to our work. When their work on mapping metadata from different languages or frameworks, such XMI(XML Metadata Interchange) and CWM(Common Warehouse Metamodel), is defined, it could result in the ability for our approach to create more input/output types. For example, they refer to mapping from XMI to a UML metamodel. When a user wants to define metadata using XMI that would be possible, because after that it can be mapped to the UML metamodel the rest of the environment uses. But because that UML metamodel could also be mapped to XMI, the metadata visualization can be done with XMI if a user prefers that.

3.2 Current markets

An important aspect within enterprise systems and development approaches aimed for enterprise systems is the existing market and the expectation on how this market will grow. There are a few markets the solution of this thesis can be applicable to:

- No- or Low Code Platforms
- Packaged application software
- In house development

Because this approach requires less coding than normally it could be seen as a no- or low code application platform. Of course it can also be packaged and be used within the packaged application software market. Because it can also be used for prototyping and rapid application development, this approach can be a great benefactor within in house development.

3.2.1 Market leaders

As for market leaders, only the low code platforms and the packaged application software markets will be discussed as in house development is of course specific for a company itself. For the low code application platforms market, Gartner created a magic quadrant showing the market position of the companies within that market, as shown in Figure 7.
According to Gartner, the leaders within this market are:

Figure 7: Magic quadrant low code application platforms market

- Salesforce[7]
- Microsoft[5]
- OutSystems[6]
- Mendix[4]
- Appian[1]

All these companies provide a low code application platform. The platforms they provide are all focused on quickly creating apps to automise processes or improve customer experience. The development is platform independent and done by modeling rather than actual coding and thus realises a short time to market.

As for the packaged application software market, the leaders according to gartner are Microsoft, SAP and Oracle[15]. These companies provide full solutions for enterprise systems. They all created a large amount of packaged applications that can be used as Enterprise Resource Planning and Customer Relationship Management and many more.

Microsoft
Microsoft is one of the largest players in both of these markets with their Power platform and Dynamics 365. The Power platform contains for example Powerapps and Power BI. Just like the other platforms, Powerapps provides an easy to work with environment where apps can be created in a short timespan. One of the advantages for Microsoft is that they have a great amount of integration with their own services such as Azure[3]. Because microsoft is a leader in both of the above mentioned markets, they have a unique position where they can provide a company with every type of software it needs.

3.2.2 Trends

According to [3], the market size of low code application platforms in 2019 is around 10.3 billion US dollars and has an expected growth rate of 31.1% between the years 2020 and 2030. The largest drivers for this growth rate are an increasing demand for business digitization and a yearning for less dependency on IT professionals. According to [2], the market size of packaged application software was valued at 8.48 billion US dollars in 2017. The expected value in 2023 is 11.69 billion US dollars, growing at a rate of 5.4%. Because the approach in this thesis can belong to both of these markets as well as being able to be used with in house development, the market size of this type of applications can expectedly grow fast within the upcoming years. This is because the same aspects that drive the growth for the low code application market are used in this approach.

[3]https://powerapps.microsoft.com/nl-nl/build-powerapps/

4 Metadata at design-time and run-time

Based on the problems discussed before, and the missing features in the solutions within the current markets, a new approach to software development is presented in this thesis. The basis for this new approach is the structure of the metadata objects at design-time and run-time.

4.1 Metadata definition

The resulting software application will need to contain a front-end as well as a back-end component. In the front-end component the editing of the UML metadata will be done, where as in the back-end component the resulting changes will be translated to the correct metadata objects which will be used to generate the corresponding source code.

The metadata of a UML class model will be saved with an OOP approach, where every aspect of the diagram such as `Classifier`, `Relationship` or `Property`, has their own corresponding class. As the metadata of a UML model can get quite complicated and extensive, we will start with a simple structure and expand it when necessary. The structure of how metadata is saved is illustrated using a UML class model in Figure 8, which is an expansion of the model illustrated in Figure 3. The metadata of a UML class model is deployed alongside the application using the structure defined here and is created as a first-class citizen. This means that at runtime the metadata objects are still being used in standard operations where other normal entities would too. All changes in these metadata objects would incur changes in subsequently called actions.

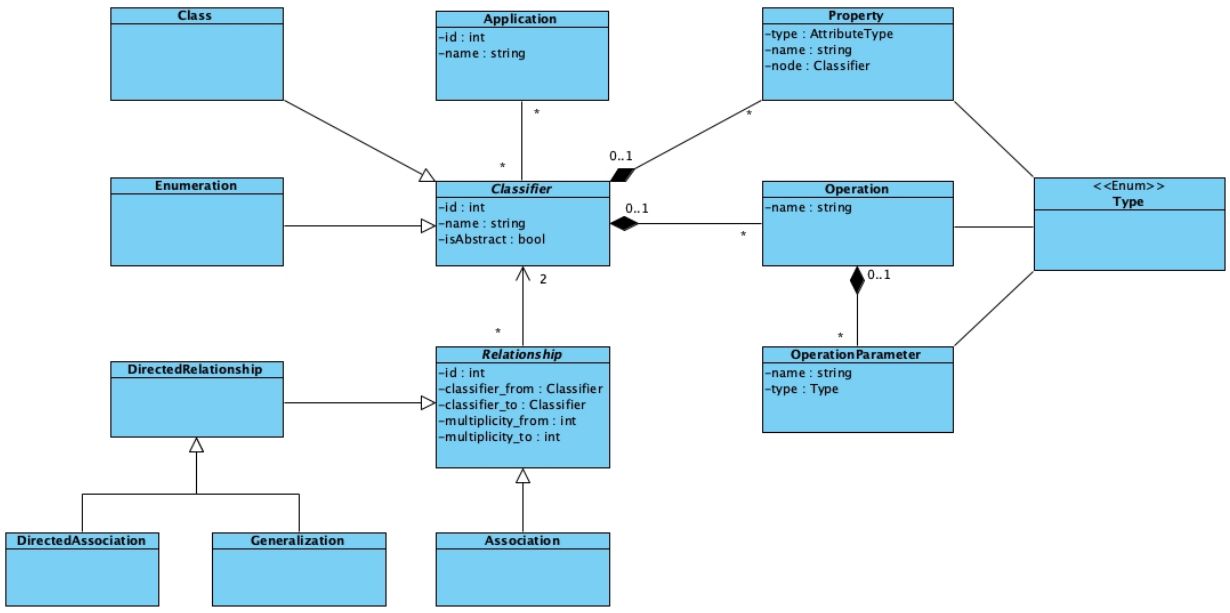

Figure 8: Structure of the metadata

As seen in Figure 8, this structure is not the complete according to the UML specification[18]. This for instance does not contain all the relationships that exist. Rather this is based on the UML

classifier specification and thus expanding is possible. When the UML class model is interpreted by the application at run-time, the source code for the definition of the classes, that is continuously generated based on the saved metadata of the UML class model, will look like the code in Listing 2 when implemented in the Django framework:

```
class Person(models.Model):
    first_name = models.CharField()
    last_name = models.CharField()
    age = models.IntegerField()


class Order(models.Model):
    order_number = models.CharField()
    description = models.CharField()
    person = models.ForeignKey(Person,
                               on_delete=models.CASCADE,
                               related_name='person')


class OrderItem(models.Model):
    quantity = models.IntegerField()
    order = models.ForeignKey(Order,
                              on_delete=models.CASCADE,
                              related_name='order')


class Product(models.Model):
    name = models.CharField()
    description = models.CharField()
    order_item = models.ForeignKey(OrderItem,
                                   on_delete=models.CASCADE,
                                   related_name='order_item')
```

Listing 2: "Resulting source code"

4.1.1 UML extension concepts

Even though the UML specification is not completely implemented, an extension to this specification is actually implemented. In this approach there are two additions to the UML standards:

- Domain model
- Application model

The domain model contains objects that are part of a specific domain. Work based on the existence of a domain is commonly used in business logic and is the basis of Domain Driven Design[12]. Domain Driven Design is a concept where the structure of an application should match the business domain it is created for. For example, when an application is created for a Sales domain, the classes the application consists of might be `SalesOrder`, `Customer`, `Product`, etc. The availability of mulitple domain models ensures the separation of classes that have no relation whatsoever. Applications however, are able to pick classes from different domains if the application use case

spans over multiple domains.

The application model defines the scope of a specific application. An application model can consist of multiple domain models and can handpick parts of a specific domain model whenever possible. The application model is a detailed model where not only different classes can be used, but also specific properties of a class. However, because multiple applications can share the use of a class, the definition and logic of a class remains within the domain model. The advantages of an application model is the ability to customly design what is needed for a specific application. Therefore it is also possible to select different properties per class; why would you use a property if it is not needed?

4.2 Dynamic aspects

The metadata discussed above focuses on the structural aspects of an application, but all applications also contain dynamic aspects. These define the internal workings of an application such as operations and class methods. There are a few possible approaches to defining the dynamics of an application:

- Action language to define operational implementations
- Activity / state machine models to define operations
- Model scripting in implementation language(approach used in this thesis)

First of all, there is the action language defined by OMG[17]. The specification defined by OMG contains an approach to make models executable. With this approach, it is possible to define operational implementations. Different UML action languages are created of which all have their different advantages. To use this implementation, one of these languages should be chosen.

It is also possible to define operations with activity- or state machine models. These models are also specified in UML and give the ability to define behavior of a class or instance. In classic MDA there would be a mapping from these models to the generated source code. In our approach we would need a run-time interpreter for these models too to be able to implement this.

The approach used in this thesis is model scripting in the implementation language. As seen in the metadata structure in Figure 8, it is possible to define operations. These operations are class methods and thus can use everything that is available to that class. The operation class has a textfield property containing the source code of that operation. The code provided has to be in the implementation language. This approach gives a structured and relatively safe way for adding dynamics to an application. When for example the `__str__` operation is added to the Person class from Figure 6, to make sure the name of the object will be the first and last name, this will be added as a class method by simply giving the implementation as shown in Listing 3.

```
1 return self.first_name + ' ' + self.last_name
```

Listing 3: "str operation implementation"

4.2.1 Adding custom code for real life applications

In real life applications, there is more operational code than class methods. To create a platform for real life applications an approach to adding custom code should be provided. There are a few different approaches on inserting custom code into an application. For example by using some kind of API or simply editing the source code files. With adding custom code there are a some notable problems:

- Changing source code could break the logic of the existing platform for interpreting the metadata structure.
- It is also possible when working with objects defined in the metadata that changes in those objects are not handled well by the existing logic and could break database related logic.

Adding custom code will most likely result in a lot of dirty code which is complex, conflicting, and after some time nearly impossible to maintain. Working with custom code should be an exception because of this and it lays a great responsibility on the developer. The platform defined in the approach this thesis provides is not a platform containing solutions for adding custom code and thus is not responsible for failures e.g. after adding custom code.

5 Prototype implementation

For this project we chose to create a web application. Most web applications use a Model View Controller architecture[14], where the Model is a database or application model containing the data of classes, the View is the part a user will see in the front-end such as a GUI, and the Controller is an object that lets you manipulate the views in some ways. When mapping the MVC architecture to Python, in particular to the Django framework, the terms used are a bit different. Views in the Django framework can be seen as the controller in the MVC model and it uses Templates as MVC's Views. Thus in the Django framework, the Template will render the front-end part and can be manipulated by Views.

5.1 Web implementation framework

For this project, the use of a web development framework is very logical because of the need for a front-end and back-end that can communicate with each other. When choosing a framework to use, multiple aspects come into play. There are a large amount of possibilities to choose from, differing per programming language and/or features. For this thesis the following aspects were of influence in the decision of choosing a framework to use for the prototype implementation:

- Language
- Ease of use
- Documentation
- Run-time interpretation

As the application needs to be able to generate code while running and changing application behavior at the same time, the code will need to be interpreted at run-time. This will drop a lot of options as every framework using a compiled language such as `C#` is not applicable here. Two languages that were took into consideration were Python and PHP. With PHP there is the possibility to use the Laravel framework which has an extensive feature set and is well documented. For Python there are frameworks as Django, Flask, CherryPi and Web2Py.

As Python is the language that is used at LIACS, the decision to use this language was obvious. This will provide the opportunity for other students to continue to work on this project while also making it easier to integrate with other projects within LIACS. The framework that will be used is Django[8]. This decision is mostly based on the enormous userbase which brings a large amount of information online. Considering that the need to look up how certain aspects can be implemented would probably be high, a userbase such as this one is useful.

5.2 Technical design

As discussed in section 5.1, the Django framework will be used. This provides the option of defining multiple applications within a project. In this case we have a `model` project and a `shared` project, where `model` contains everything for the model definer and interpreter and `shared` is the application

containing the result of the code generation done by the model interpreter. Outside of these applications, the run-time applications created by adding an `Application` object are receiving the name that the user gives them. These created applications only contain specific views for all their classifiers.

The front-end of this application will consist of forms that make it possible to create new applications, classes, properties, operations and relationships. Because of the way Django works, it is possible to set up different applications that are able to use the same models from a shared application. This is how this system will work too. A user can create multiple applications in which he can assign classes to be used in this application. All classes are defined in a shared application and thus all have the same database table, no matter in which application they are used. Within every class that is assigned to an application, the user will be able to select which properties of the class that specific application should use. Only the selected properties are shown and used in that application.

5.3 Model execution

Interpretation of the metadata will be done based on the info in the metadata objects that are saved in the database. Adding or changing anything in the UML class diagram will trigger this generation. The generation will be split into generating the classifiers themselves from the node and attribute data, and the relations between them based on the link data. The first successful generation was for generating the code for a new `Class` with it's attributes. Where there were three different attribute types; integer, string and boolean. A part of the code for this is shown in Listing 4.

```
def generate_classifier(self, classifiers, model_file):
    file_content = model_file.read()
    if file_content == '':
        model_file.write('from django.db import models\n\n')
        file_content = 'from django.db import models\n\n'

    for classifier in classifiers:
        if isinstance(classifier, Class):
            if self.generate_class_declaration(classifier.name) not in file_content:
                self.generate_new_class(classifier, model_file)
```

Listing 4: "Basic class generation"

The functions this loop uses are defined as in Listing 5.

```
def generate_class_declaration(self, name):
        return '\nclass ' + name + '(models.Model):\n'


def generate_new_class(self, classifier, model_file):
    model_file.write(self.generate_class_declaration(classifier.name))
    for attribute in Attribute.objects.filter(classifier_id=classifier.id, updated=True):
        model_file.write(self.generate_attribute_declaration(attribute.name, attribute.type))


def generate_attribute_declaration(self, name, type):
```

```
    declaration = '    ' + name + ' = '
    if type in Type._value2member_map_:
        if type == Type.String.value:
            return declaration + 'models.CharField(max_length=255)\n'
        elif type == Type.Int.value:
            if name == 'id':
                return declaration + 'models.IntegerField(primary_key=True)\n'
            else:
                return declaration + 'models.IntegerField()\n'
        elif type == Type.Bool.value:
            return declaration + 'models.BooleanField()\n'

    return declaration + 'models.CharField(max_length=255)\n'
```

Listing 5: "Basic class generation (2)"

In listings 4 and 5 we can see that for every `Node` in the database a class definition is generated after checking if it already exists and for every `Attribute` in this class an attribute declaration is appended after the class declaration. This is a simple example and does not contain the complete source code for the model execution. Aspects that are not showed here are:

- Writing at a certain line instead of appending to the file
- Check if an attribute already exists per class
- Generation of relation declarations
- Generation of operation declarations

The implemented relationship interpretation does not contain every possible relationship within the UML class model specification, but rather is focused on one relationship type to demonstrate the approach. Operation are generated as class methods and the implementation given by the user is added as the class method code.

5.4 View generation

To make an application workable, there still needs to be a front-end where users can work with the given data. In this project, for every class defined in the metadata, a view is generated for CRUD(Create Read Update Delete) operations, such that users can work with the given objects. Whenever a class is created, a view will be generated in de shared application. These views make use of a set of generic html pages that can:

1. Show a list of all instances of that class
2. Add a new instance of that class
3. Edit an existing instance
4. Select which properties of the class should be used in this application

At first, This does not contain much because there are no properties attached to the class yet. The source of for example the index function when there are no properties yet is shown in Listing 6:

```
def index(request):
    properties = []
    objects = Order.objects.all()
    for object in objects:
        object.properties = []
        object.properties.reverse()
    context = {'application': __get_application(request),
               'properties': properties,
               'objects': objects,
               'object_name': 'Order',
               'object_name_lower': 'order'}
    return render(request, 'view_list.html', context)
```

Listing 6: "View index generation)"

For each property created for the class, extra code is generated to be able to handle all the logic around properties. When a property 'Person' is added because of adding a relationship between the Order and Person class, the index view source code will be updated to the code in Listing 7

```
def index(request):
    properties = []
    objects = Order.objects.all()
    for object in objects:
        object.properties = []
        object.properties.append({'name': 'Person', 'type': 'association', '
   value': object.person.__str__})
        object.properties.reverse()
    properties.append({'name': 'Person'})
    context = {'application': __get_application(request),
               'properties': properties,
               'objects': objects,
               'object_name': 'Order',
               'object_name_lower': 'order'}
    return render(request, 'view_list.html', context)
```

Listing 7: "View index generation with property)"

Because an application can use a limited amount of properties of classes, the shared view is copied to the applications views to be able to edit it without changing anything for other applications. This way it is possible to use different sets of properties of a class between multiple applications.

6 Worked example

A prototype application for this development approach is created using Python with the Django framework. This application generates code based on changes in the application and domain model(s). At the homepage of this application, displayed in Figure 9, a simple menu is displayed containing the different parts of the models that are editable.

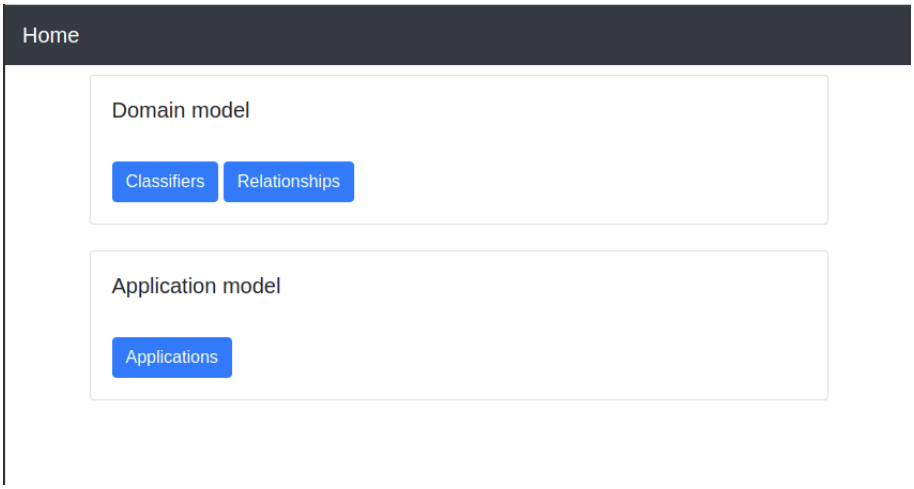

Figure 9: Application homepage

6.1 Creating classifiers

To begin creating a domain model, one or more classifiers need to be added. This is possible by opening the 'classifiers' page and clicking 'Add classifier'. On the next page, fill in a name and the type of classifier. When this is done the application will generate the corresponding source code for this classifier. Because the classifier is saved as an object in the database, the 'classifiers' page now contains all the created classifiers as shown in Figure 10. To update the database containing the tables for the domain model, database migrations are created and executed. This happens under the hood after the source code generation.

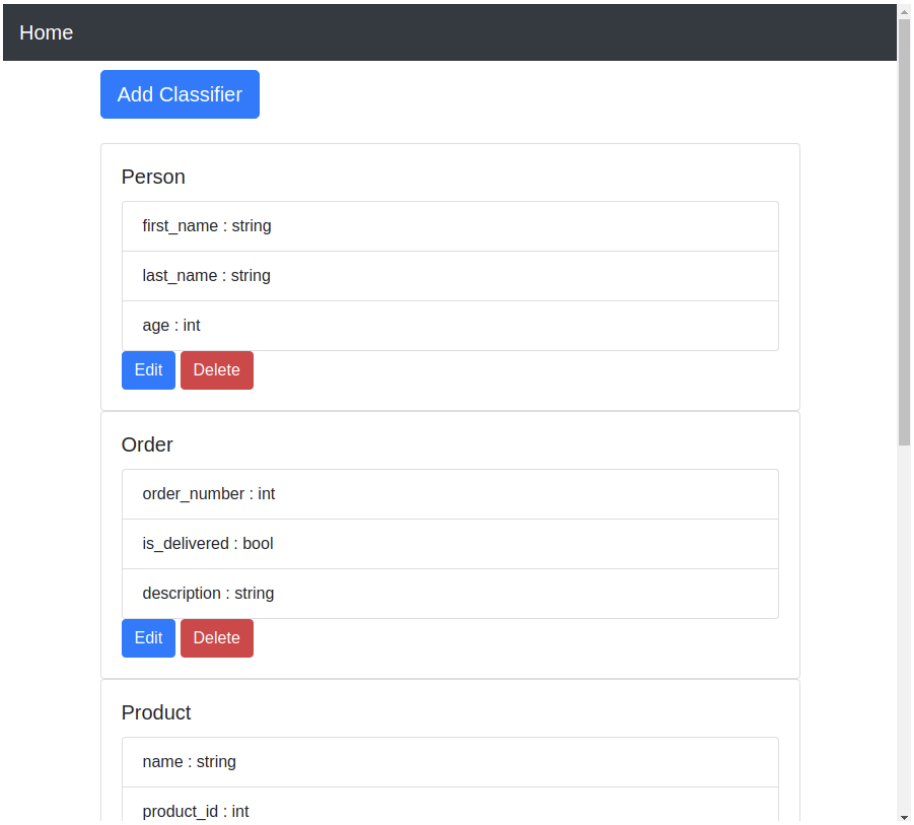

Figure 10: Classifiers page

6.1.1 Adding properties

To add properties, navigate to the edit page of a classifier. On this page, as shown in Figure 11, you will see all properties and operations the classifiers has. To add new properties simply fill in the name of the property and select which type it is.

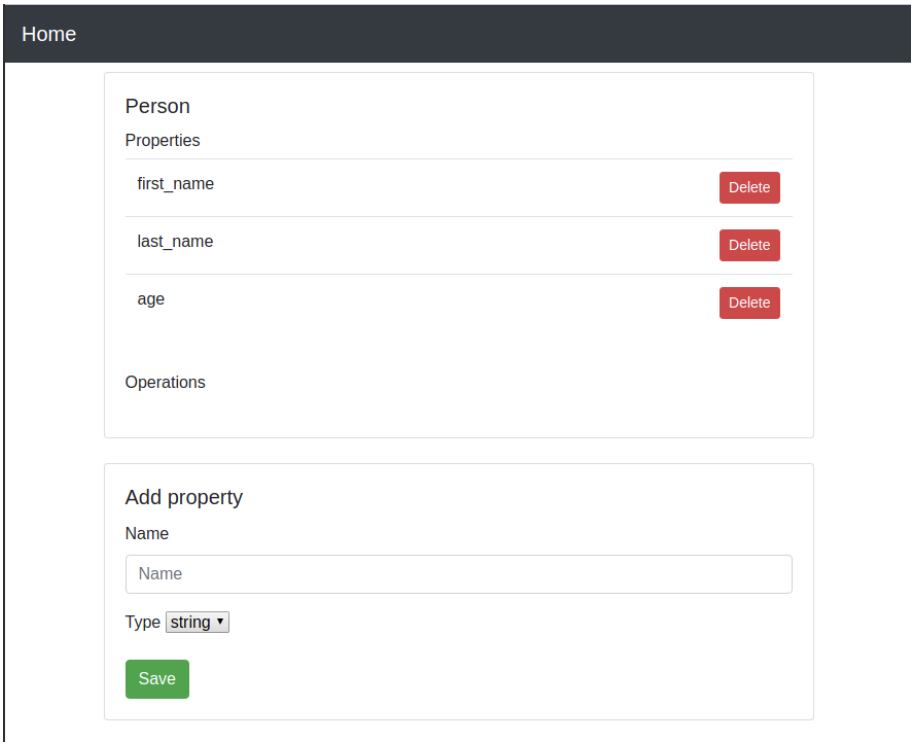

Figure 11: Classifier edit page

6.2 Adding relationships

After the classifiers are created, it is possible to add relationships between them. This can be done by navigating to the relationships page and clicking on 'Add Relationship'. In the following screen you choose what type of relationship you want to add. In this example application the only supported relationship is an association, which can be 'One to Many' or 'One to One'. To add classifiers, select the classifiers you want to link and fill in the names and multiplicity of the relationship. The 'classifier_from name' is what you want to name the relationship within the classifier_from model. For example, in a relationship from 'Order' to many 'Persons', the classifier_from name would be 'person'. This name will be added in the 'Order' model. After adding the relationships, the relationships page will contain all new additions as shown in Figure 12.

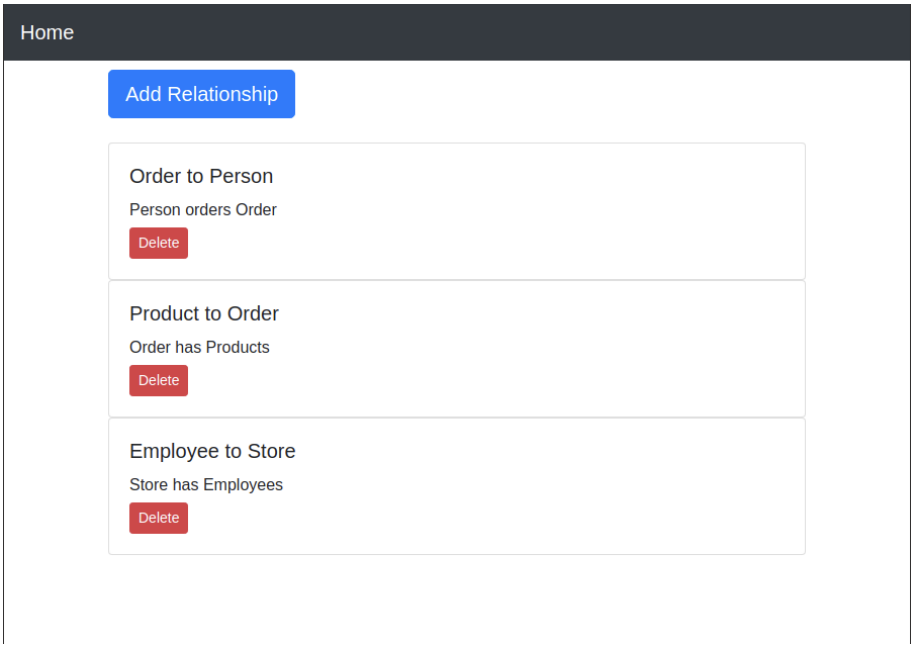

Figure 12: Relationships page

6.3 Adding operations

The last aspect of the domain model that is implemented is operations. Operations can be added to a specific class as class methods. Adding an operation is mostly the same as adding a property. The difference is with operation parameters. These can be added by editing an operation. For every operation and operation parameter it is possible to select the basic types as parameter type or return type. When the `__str__` operation is added, the classifier edit page will show it under the properties as shown in Figure 13.

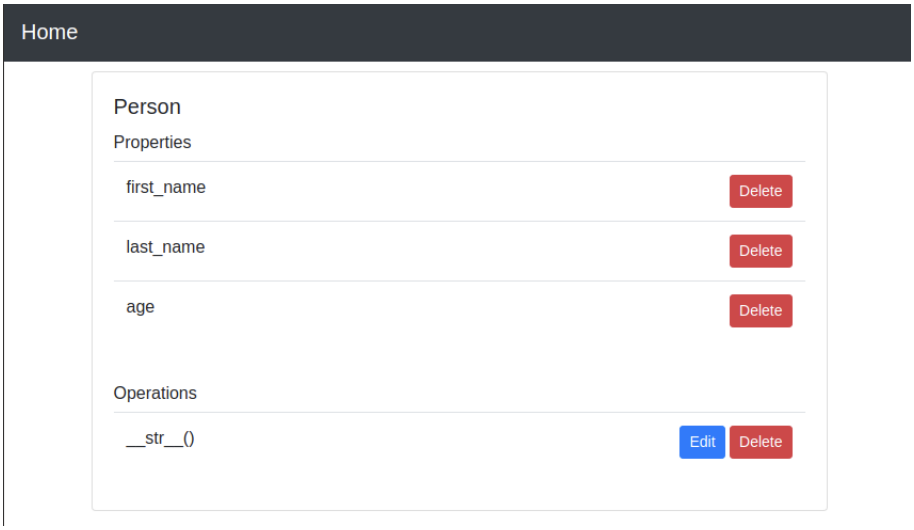

Figure 13: Operation added

6.4 Domain model diagram

With the classifiers and relationships added, the domain model is now set up. By navigating to '/model/diagram', a UML class diagram is generated based on the domain model. The diagram from this example application is shown in Figure 14.

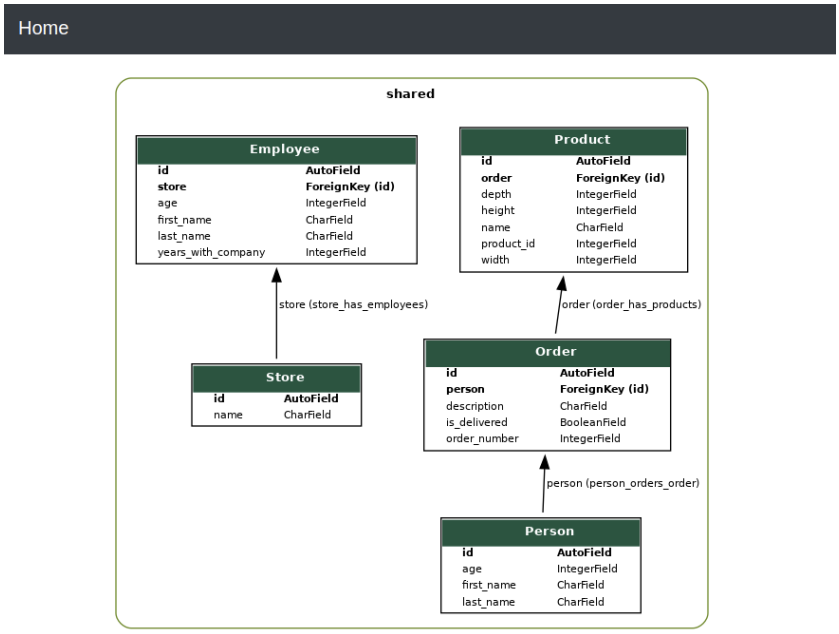

Figure 14: Domain model diagram

6.5 Creating applications

Next to the domain model, it is possible to define an application model. This can be done at the application model section of the homescreen, shown in Figure 15. At the applications page it is possible to add a new application. After confirming the name of the desired application the corresponding source code is generated. This contains some settings within the django project and the base route of the application. Also, to be able to have different views per application seperate files are created.

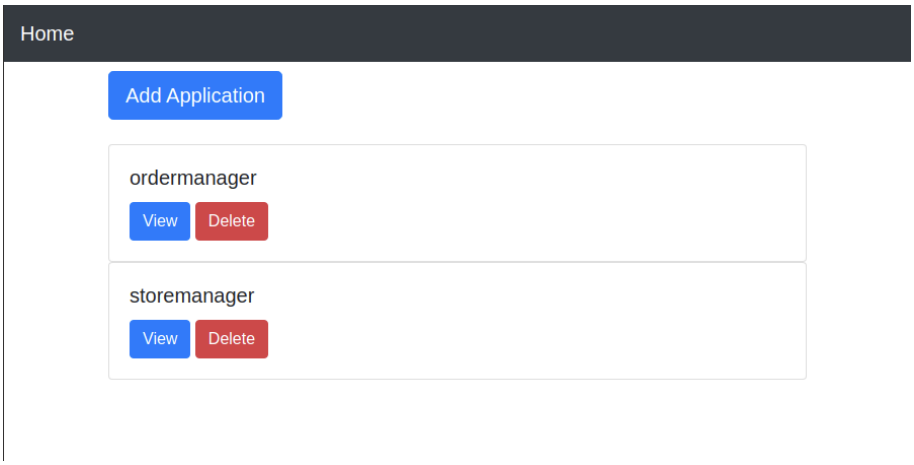

Figure 15: Application page

6.5.1 Linking classifiers to an application

Classifiers can be linked to different applications, as seen in Figure 16. To link one or multiple classifiers to an application, view the concerned relation and add the desired classifier. After the desired classifiers are linked, all of them will be listed at the application's view page. By linking a classifier to an application, an extra view file is created that is used for viewing the classifier's data and performing crud operations on the objects. This is all possible after running the created application.

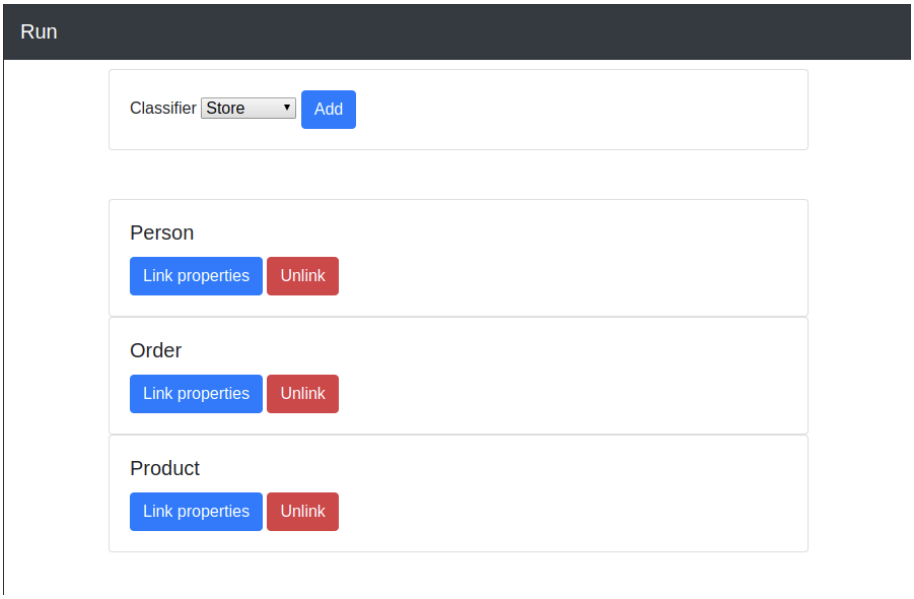

Figure 16: Application link classifiers page

6.5.2 Linking properties to an application

It is also possible to select different properties of a classifier that should be used within an application, as seen in Figure 17. This is possible at the 'Link properties' section of a classifier in an application. This will result in additional code in the view file of the regarding classifier that makes sure that all the properties selected are used in the crud operations on the objects at run-time. All relationships are added to the run-time application the same way as linked properties are.

Figure 17: Application link properties page

After all this is done, it is possible to work with the applications. All applications contain a list of objects for each classifier they are linked with. This is shown in Figure 18.

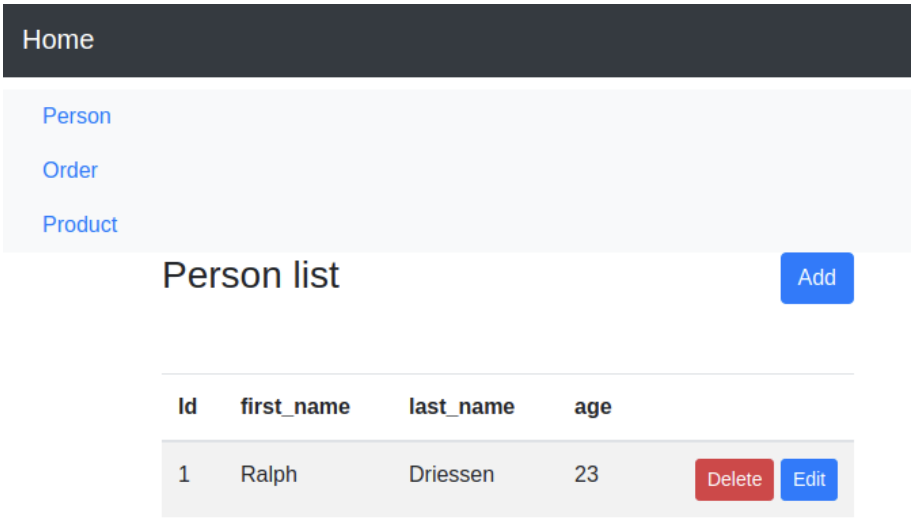

Figure 18: Application page

6.6 Evolution example

An important aspect of the approach presented in this thesis is of course not only the speed in which the initial design of an application results in a running application, but rather the speed of evolution after the application's initial state is running. Figures 19, 20, 21 and 22 illustrate how adding a few metadata objects at run-time result in changes in the running application. The initial orders page contains an order showing it's current properties. At this order the reference to Person is 'Person object (2)' because that is the current result of the default `__str__` operation. After adding a new property 'delivery_address' to the Order class and an implementation of the `__str__` operation to the Person class, the orders page is updated to use those as shown in Figure 22. **Note:** The address in the order object is added to clarify that adding the property actually works at run-time.

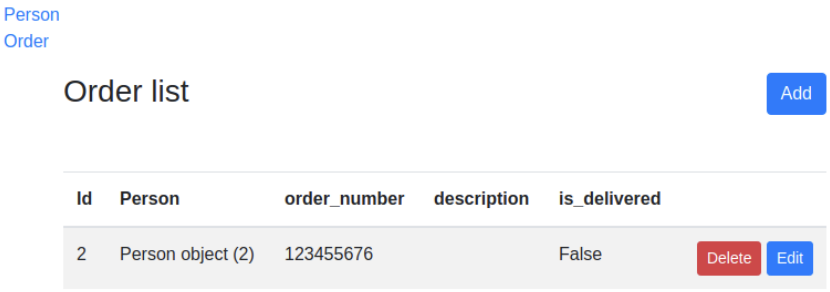

Figure 19: Initial Orders list page

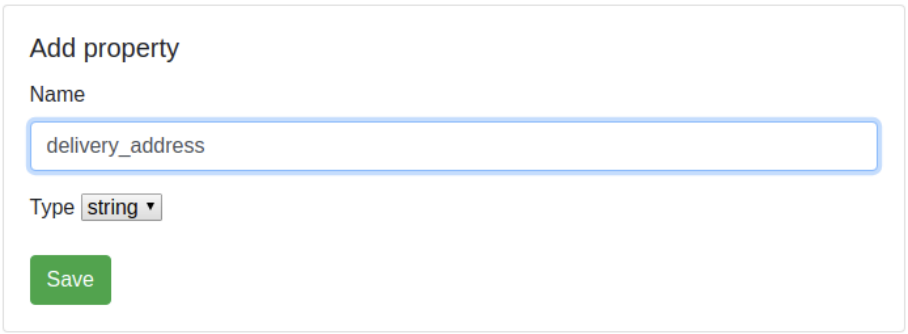

Figure 20: Add delivery_address property to Order class

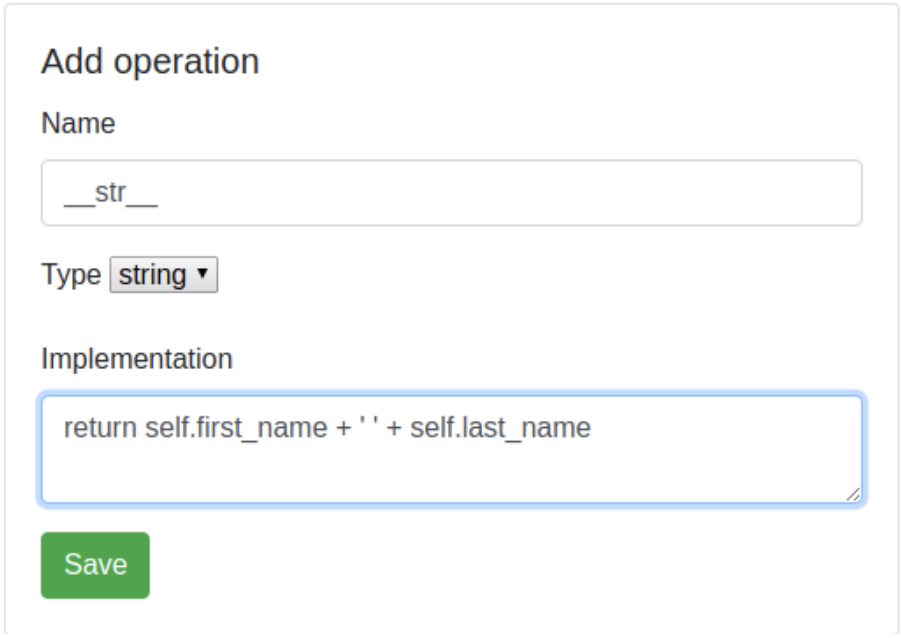

Figure 21: Add __str__ operation to Person class

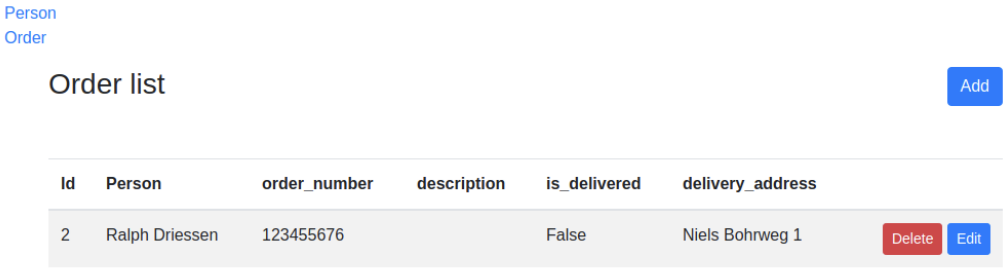

Figure 22: Resulting Orders list page

6.7 Change handling

Because it should be possible with this implementation to have multiple users working on the application, the handling of changes should be clearly defined. In this implementation changes to the UML metamodel can be done at the same time. For example when one user adds a new property to a class at the same time another user add a different property to that class, the result is that both properties will be added. However, as no duplicate check is implemented, this also means that if both users add the same property, the property is added twice which will result in failure at model interpretation because the database can't add two columns with the same name.

Updates in the UML class model while run-time applications are already using it result changes in the run-time applications as well. When for example a new property is added to a class, the existing objects of that class do not contain data for that property. In this implementation this is solved by making every field nullable. This simply means that an object will contain a `null` value when a new property is added. After the property is added, it can be given a value for an object when editing the object in one of the run-time applications.

7 Future work

7.1 Running application

7.1.1 User interface

The currently created user interface is simplistic and is not customized to the needs of real life application users. The user interface needs to be extended with at least the basic features of a real life business application such as filtering, sorting and searching. Also the structure needs some expansion, currently if there are a lot of classifiers, the navigation tab on the left will get very long and won't fit on the screen anymore. Navigation between classifiers should be possible and also the expansion of an object that is a property of another object. For example, if an Order has a Customer, clicking on the Customer object should initialize an expansion of all the info of that customer.

7.1.2 UML editor

Another feature of this approach of software development is to give users different ways to add or edit the metadata of the model. In the application made for this project, the only way to add metadata is to manually add the objects by filling in forms. Another way of providing metadata would be editing a UML diagram. This gives the user different approaches to changing the model thus the user can decide on which one to use according to it's preferences. Most of the time multiple people work on a software project and the fields of which these people have knowledge in can vary. Because of this, being able to edit the metadata from different approaches the chance of a better user satisfaction at editing the metadata is higher. One employee can have experience in UML modeling while the other rather edits it from an object view with forms. Providing different approaches better addresses people with different skillsets.

7.1.3 Extended UML specification

The currently implemented aspects of the UML specification only contain:

- The standard class with properties and operations
- The association relationship type
- Basic datatypes: integer, string and boolean

For a full blown platform it is needed to implement the complete UML specification. This includes other classifier types as Interface or Abstract Class, but also all the different types of relationships, in particular sub typing. The current list of basic datatypes should be extended to contain all of them, such as arrays, enumerations and floating point numbers. To be able to use different objects in other classes, it should also be possible to use a previously created classifier as datatype. This thesis provides an extension to the UML specification by adding Application and Domain. These should be integrated with the existing UML specification to work properly in a complete solution.

7.1.4 Editing existing metadata object properties

In the implementation provided in this project it is not possible to edit existing metadata properties such as the name of a classifier or property. When this possibility is implemented it provides a platform for quick fixes that can be done in a few seconds. Currently when a change in existing metadata properties is desired, the old version should be deleted and a new version created. This will result in loss of data. This is not a problem at the design-time, but as this approach is meant to provide a platform for software development at run-time this is a crucial aspect of a fully providing solution. Use cases of this aspect would for example be fixing a typo or quickly editing a name after an epiphany. For instance when a small change is thought of when commuting. At that moment it would be possible to grab a cellphone to make the quick fix.

7.2 Rules

Business logic can be added with rules. With these rules, business processes can be defined with modelling. A lot of enterprise systems contain an environment where rules can be added to an application. To make the approach from this thesis more complete, such an environment should be added here too. There are multiple types of models that can define businness processes, for example BPMN(Business Process Model and Notation). One model type should be picked and implemented in the application to provide business rule integration. Of course, the gist of the approach proposed in this thesis is metadata at runtime, thus the rule modelling should be in line with this too.

7.3 Custom code

As discussed in section 4.2.1, custom code needs to be supported to implement dynamic aspects of an application. This can be done in multiple ways which all have their advantages and risks. One of the risks of adding custom code to an application is breaking the logic of the metadata interpretation. Also, adding custom code will likely result in dirty code that is not easily managable and will only continue to build up and become completely impossible to maintain. Keeping the custom code within existing models would be more maintainable however it would provide with less opportunities and freedom. To support custom code completely, an environment should be worked out where the custom code can be added to the application and a specific structure should be thought to prevent unmaintainable code.

7.4 Multiple domain models

Currently the worked example shown in chapter 6 contains only one domain model. When working for a larger enterprise system this domain model would get enormous and complex. Therefor it would be better to be able to define multiple domain models, each with their own database tables and class models. This way the specification of the domains would be more structured and maintainable. The applications would be able to choose what classifiers to use from different domain models. To get this working it should be worked out how the different domain models would, if at all possible, work with eachother. There also a lot of small aspects that should be worked out such as multiple classes with the same name existing in different domain models. An application will then be able to pick for example 3 different types of a `Person` class. These edge cases should be worked out and handled within the domain model logic.

8 Conclusion

This thesis presented a new approach for application development based on MDA. It adds additional features to this approach, by demonstrating that metadata should not just be used at design-time, but also be deployed alongside the application at run-time. As a result of interpreting the metadata of a UML class model at run-time, the evolution of software systems can be done rapidly. When fully worked out, a platform based on this approach can be a solution to the existing problems in software development; A lack of IT professionals and a slow time to market.

To demonstrate the approach we developed a prototype application where a UML class model is deployed at run-time alongside the application by it's metadata. The application is structured as illustrated in Figure 5. It consists of a web application a user interacts with, containing a model definer and the run-time applications. This prototype application succesfully demonstrates the ease and speed of developing by editing UML class model metadata at run-time. Because the changes are done at run-time and then are interpreted under the hood, the changes immediately result in the change of behavior of the run-time applications. This prototype also showed that development can be done by a modeler instead of an expert developer and thus reduces the need for IT professionals.

From this prototype, we draw several lessons about the process of developing a system for this new approach. First of all, there are a lot of edge cases one should think of, most importantly to make sure that the source code resulting from the metadata changes are not creating conflicts with existing code. Also the error handling should be implemented very well so that the saved metadata will never be out of sync with the source code when something goes wrong while updating the source code. Lastly we learned that editing source code by hand which is also involved in the metadata interpretation is tricky and can mess up the metadata interpretation logic and thus should be done with great care.

As discussed in chapter 7, a lot is still missing in our prototype implementation. Outside of user experience aspects such as a UML editor and a fitting user interface for an enterprise system,, there are also some core features that need to be added to make the application actually usable by a business. Features such as process modelling with business rules and adding custom code are needed to actually have a fulfilling application that can be used within enterprises. Also the theory on how to work and interact with multiple domain models has yet to be worked out.

References

[1] Appian. https://www.appian.com/platform/.

[2] Application platform market. https://www.marketsandmarkets.com/Market-Reports/application-platform-market-28942023.html.

[3] Low-code development platform market forecast, 2030. https://www.psmarketresearch.com/market-analysis/low-code-development-platform-market.

[4] Mendix. https://www.mendix.com/platform/.

[5] Microsoft power platform. https://powerplatform.microsoft.com/nl-nl/.

[6] Outsystems. https://www.outsystems.com/platform/.

[7] Salesforce. https://www.salesforce.com/nl/products/.

[8] The web framework for perfectionists with deadlines, django. https://www.djangoproject.com.

[9] A. Billig, S. Busse, A. Leicher, and J. G. Süß. Platform independent model transformation based on triple. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 493–511. Springer, 2004.

[10] P. Caceres, E. Marcos, and B. Vela. A mda-based approach for web information system development. In *Workshop in Software Model Engineering*, 2003.

[11] M. Das and A. Yiu. Business process management and ws-bpel 2.0. *Oracle Corporation*, 2006.

[12] E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[13] A. Gavras, M. Belaunde, L. F. Pires, and J. P. A. Almeida. Towards an mda-based development methodology for distributed applications. In *Proceedings of the 1st European Workshop on Model-Driven Architecture with Emphasis on Industrial Applications (MDAIA 2004), CTIT Technical Report TR-CTIT-04-12, University of Twente, ISSN*, pages 1381–3625, 2004.

[14] I. Hatanaka and S. C. Hughes. Providing multiple views in a model-view-controller architecture, July 20 1999. US Patent 5,926,177.

[15] G. Inc. Market share: Enterprise application software, worldwide, 2018. https://www.gartner.com/en/documents/3906841/market-share-enterprise-application-software-worldwide-2.

[16] H. Lan. Web-based rapid prototyping and manufacturing systems: A review, Jun 2009.

[17] Object Management Group. Action language for foundational uml. 2017.

[18] Object Management Group. Unified modeling language. 2017.

[19] J. D. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *Workshop on Metamodeling and Adaptive Object Models, ECOOP*, volume 50. Citeseer, 2001.

[20] C. Sansores and J. Pavón. Agent-based simulation replication: A model driven architecture approach. In *Mexican International Conference on Artificial Intelligence*, pages 244–253. Springer, 2005.

[21] R. Soley et al. Model driven architecture. *OMG white paper*, 308(308):5, 2000.