



Universiteit  
Leiden  
The Netherlands

# Opleiding Informatica

A flexible, template-driven generation framework  
for Model-Driven Engineering

Sven van Dam (s2634376)

Supervisors:

Dr. G.J. Ramackers & Prof.dr.ir. J.M.W. Visser

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)

[www.liacs.leidenuniv.nl](http://www.liacs.leidenuniv.nl)

17/04/2024

### **Acknowledgements:**

This research is part of the AI4MDE research project at Leiden Institute of Advanced Computer Science. I would like to express my gratitude to Lucas Willemsens, Pieter de Hoogd, and Max Boone for their skillful contributions in this research. Also great thanks to Dr. G. J. Ramackers and Prof.dr.ir. J.M.W. Visser for their supervision.

## Abstract

The recent rise in popularity and advancements in artificial intelligence has developers and researchers wondering whether software could be written by artificial intelligence. State-of-the-art chat-based language models such as ChatGPT 3.5 lack determinism and scalability in software generation and do not provide the user with an intuitive design studio to alter its output. This thesis aims to implement deterministic generation of prototype web applications in such a studio by extending the generation architecture in the research project *Artificial Intelligence for Model-Driven Engineering* (AI4MDE). The project's metadata was extended, scripts were written to parse this metadata into abstract objects and templates were created that could be combined with these objects to generate output files for prototypes. The extended architecture is effective in generating simple prototypes that can have multiple applications and pages, on which multiple classes can be operated upon using either *Create/Read/Update/Delete* (CRUD) operations or custom-written operations. The generation of prototypes is quick and consistent. However, practicality is still lacking before real end-users can use the environment to generate real prototypes.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Context and problem statement . . . . .	6
1.2	Research objectives . . . . .	6
1.3	Academic contribution . . . . .	7
1.4	Structure . . . . .	7
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	OMG Unified Modeling Language . . . . .	7
2.2	Model Driven Development . . . . .	8
<b>3</b>	<b>Earlier findings</b>	<b>8</b>
3.1	UML and automatic software generation . . . . .	8
3.2	Model Driven Development platforms . . . . .	9
<b>4</b>	<b>Related work</b>	<b>9</b>
4.1	ngUML . . . . .	9
4.2	Architecture . . . . .	9
<b>5</b>	<b>Research methodology</b>	<b>10</b>
5.1	Research by design and example driven development . . . . .	10
5.2	Case studies . . . . .	10
<b>6</b>	<b>Design</b>	<b>11</b>
6.1	Generic algorithm . . . . .	11
6.2	Implementation of the algorithm . . . . .	12
6.3	Section component data structure . . . . .	13
6.4	MVT generation . . . . .	15
6.5	Reusable application components . . . . .	17
6.6	Integration into ngUML environment . . . . .	18
6.7	Maintaining database entries and abstraction problems . . . . .	18
<b>7</b>	<b>Worked example</b>	<b>18</b>
<b>8</b>	<b>Security</b>	<b>21</b>
<b>9</b>	<b>Testing</b>	<b>22</b>
<b>10</b>	<b>Conclusion</b>	<b>23</b>
<b>11</b>	<b>Future work</b>	<b>23</b>
11.1	Technical improvements . . . . .	24
11.2	Abstract extensions . . . . .	24
	<b>References</b>	<b>26</b>

<b>A</b>	<b>Testing questions and responses</b>	<b>27</b>
A.1	First questionnaire . . . . .	27
A.2	Second questionnaire . . . . .	28
<b>B</b>	<b>User script</b>	<b>31</b>

# 1 Introduction

## 1.1 Context and problem statement

The recent rise in popularity and advancements in artificial intelligence has developers and researchers wondering whether software could be written by artificial intelligence. The realization of this concept could reduce workloads on developers by increasing speed, efficiency, and scalability in software development. These improvements would lead to cost reduction in business models. However, state-of-the-art chat-based language models such as ChatGPT 3.5 lack determinism and scalability in software generation and do not provide the user with an intuitive design studio to alter its output. There are currently no pragmatic environments in which a user can both intuitively prompt a language model to automatically generate software and in which the user also has full control during this generation process. The implementation of such an environment would close the gap between developer-driven development and AI-assisted development by providing a studio in which the user can not only prompt language models for AI-assisted programming but in which the user can also deterministically alter the output of such a language model in an intuitive editor, regaining full control of the end-product. This thesis aims to implement deterministic generation of prototype web applications in such an editor by extending the generation architecture in the research project *Artificial Intelligence for Model Driven Engineering* (AI4MDE), previously known as *next-generation UML* (ngUML), at Leiden Institute of Advanced Computer Science. This thesis will refer to this research project using its old name ngUML.

## 1.2 Research objectives

This thesis is an extension of the generation architecture of ngUML that was implemented by R. Driessen [Dri20] and B. van Aggelen [vA22]. This implementation made use of a simple What You See Is What You Get editor that solely acts on UML class diagrams that are produced by ngUML's language model. The implementation does not directly output a functional web application with multiple pages and underlying interactions but only renders simple pages that are manually created by the end user. This lack of automation leads to the main research objective:

**ngUML's generation architecture must be extended to support model-driven automatic generation of a web application prototype with functional behavioral flow.**

This implementation must be connected to the already existing design studio, resulting in one prose-to-prototype interface in which the user can generate projects and edit their corresponding metadata. Preferably, the newly implemented design should be one component that has a single JSON file as input and a prototype as an output, and the already present design studio should be revised to comply with the new generation architecture. To achieve the research objective the following questions will have to be answered:

- **What should a web application prototype with functional behavioral flow look like?**

Before any adjustments to the generation architecture should be considered, it must be clear what a desired output prototype looks like.

- **What metadata must be used to facilitate the generation of a web application prototype with functional behavioral flow?**

The currently present metadata is likely insufficient to consistently implement behavioral flow. New data structures will have to be added to ensure the solidity and flexibility of the prototypes.

- **How can metadata be used to generate prototypes?**

Presumably, the values that are provided in the design studio by the end user will need to be parsed into an intermediate form to prevent the new generation architecture from being too clear-cut for *ngUML*.

- **How must the already present design studio be edited to comply with a new generation architecture?**

When the generation architecture is extended, the design studio unquestionably will not be compatible anymore and needs to be altered.

### 1.3 Academic contribution

The extension of *ngUML*'s generation architecture combines questions from multiple academic fields. Firstly, the used metadata must be minimally extended to enable the generation of behavioral functionality in a prototype. This metadata must be retrieved from a user-friendly user interface in the design studio and parsed to intermediary objects using multiple traversal and parsing algorithms. Finally, these objects must be used to generate output files using a templating language. Above all, this thesis is a continuation of earlier theses of R. Driessen [Dri20] and B. van Aggelen [vA22].

### 1.4 Structure

This thesis will provide relevant background information about UML and model-driven development and explain *ngUML*'s architecture as of September 2023. After that, a research approach and design will be given for the extension of *ngUML*'s generation. This design will be clarified in a worked example and evaluated and discussed using real test users in Section 9. Finally, a conclusion is given and possible future extensions will be explained.

## 2 Background

### 2.1 OMG Unified Modeling Language

*Unified Modeling Language* (UML) is a modeling language that visualizes the design of object systems and software. The language is specified by the *Object Management Group* (OMG). As of the 2017 official documentation [Gro17], UML comprises 14 diagram types, which are to be divided into 2 groups. Structural UML diagrams are UML diagrams that describe the static structure of a system. Behavioral UML diagrams are UML diagrams that describe the dynamic flow of a system. Three broadly used types of UML diagrams are Class, Use Case, and Activity Diagrams. Class diagrams are structural UML diagrams that describe classes with their attributes and relationships between these classes. Use Case diagrams are human language-based behavioral diagrams that

describe separate ways in which distinct types of users can interact with the system. Activity diagrams are behavioral UML diagrams that describe the logical flow of an algorithm or system.

## 2.2 Model Driven Development

Model-driven development is software development using a *Model Driven Architecture* (MDA). Such an MDA was defined by the Object Management Group in the early 2000s [Gro14] and can be summarized as an architecture in which software is developed based on models. Firstly, a concept that is to be implemented is visualized using abstract *Platform-Independent Models* (PIMs). These PIMs can then be converted to platform-specific models, which consider technologies used in the development process. The advantages of using model-driven development are a greater use of abstraction, more consistency, higher reusability, and clearer communication. Disadvantages are that real-world scenarios often require too complex models and that consistency is not always straightforward to maintain.

## 3 Earlier findings

### 3.1 UML and automatic software generation

The primary reason for OMG to adopt UML was to standardize methods to describe the structure of an object-oriented system and to have visual aids in the construction of such systems. Thus, the scope of UML has always been software development. A considerable amount of research has been done on the automation of this process and whether UML diagrams can be used to deterministically generate systems automatically instead of only using these diagrams as a visual aid for the developer. In 2019, M. Mukhtar and B. Galadanci published a paper analyzing the state-of-the-art methods of automatic code generation based on UML [MG19]. The main conclusion from this research paper is that automatic code generation has been achieved largely for only single stand-alone UML diagrams and that much of this generation is performed in Java. Research on the possibilities of automatic software generation using multiple types of UML diagrams has been done but still lacks practicality. For example, B. Hussein and A. Salah have proposed a framework for Java code generation based on Class and Activity diagrams [HS13]. However, this framework is only suitable for lightweight uncomplicated algorithms and, more importantly, requires the activity diagrams to fully describe the algorithm in low-level individual steps. In practice, activity diagrams used in business operations usually contain nodes with human-like text and thus cannot be used for code generation in this framework. S. Burmester et al. describe in *Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code* [ea05] how PIMs can be used to automatically generate code. This translation is done by extending the metadata of existing UML diagrams.

Research has also been done on the reversed process of model-driven development. The papers *A Model-Driven Reverse Engineering Framework for Generating High-Level UML Models from Java Source Code* [ea19] and *Designing and implementing a tool to transform source code to UML diagrams* [ea21] describe how UML diagrams can be constructed from source code as an input. This reversed process can be analyzed when designing a non-reversed process.



## 3.2 Model Driven Development platforms

As B. van Aggelen discusses in his thesis [vA22], platforms that accommodate model-driven development exist, but such platforms usually have the disadvantages of high costs as well as a lack of standardization. Another reason why pragmatic model-driven development environments have not been popular yet is discussed in the conference paper *Cognifying Model-Driven Software Engineering* [ea18]. This paper argues that the benefits of such frameworks do not outweigh the costs.

# 4 Related work

## 4.1 ngUML

The goal of *next-generation Unified Modeling Language* (ngUML) is to provide a framework for prose-to-prototype automatic model-driven generation that can be used with practical, real-world UML diagrams. ngUML is a research project at the Leiden Institute of Advanced Computer Science which aims to automatically generate prototypes based on human input. As of present, UML Class, Use Case and Activity diagrams are generated using NLP models based on human text or speech input. These diagrams can be edited in a web-based studio within the environment. Prototype generation, however, has only been partially implemented for UML Class diagrams. In 2022, B. van Aggelen implemented a What You See Is What You Get editor in which the user can manually assemble pages [vA22]. Behavioral flow, including automatic page and URL generation based on Use Case and Activity Diagrams, is however not implemented yet.

## 4.2 Architecture

As of September 2023, the *ngUML.runtime* repository (the prototype generation part of nguML) only provides means to generate a Django project with a single application and a single *models.py* file that corresponds to a UML class diagram. This application is generated without any Django views, URLs, or templates. This means that functionality to edit the database through either an API or a user interface is lacking. A What You See Is What You Get Editor was implemented by B. van Aggelen to try to counteract this problem. Using this editor, the user can manually create pages that have HTML input fields corresponding to the attributes of the classes in the UML class diagram. This is, however, time-consuming for larger applications and requires the user to have a full understanding of classes, attributes, and HTML. Besides that, one could argue that a prototype is not automatically generated when the user must manually build each page for the corresponding application. To make ngUML more accessible and practical for users who have no or little experience in web development, it must be extended with the generation of pages within an application and the generation of backend functionality that can be called by user actions on these pages. This generation must still be adaptable at design time to allow adjustments from the user if the prototype does not yet satisfy the needs.

## 5 Research methodology

### 5.1 Research by design and example driven development

The strategy used in this research can be classified as research by design. The research was performed in cooperation with L. Willemsens and P. de Hoogd, who also actively worked on this project for their bachelor's research thesis. From September 2023 up to April 2024, ngUML's repositories have been extended iteratively to try to resolve the project's problems which were mentioned earlier in Section 4.2. Multiple case studies were worked out to imitate the input that real-world users might feed to the project. During the implementation of these case studies, multiple problems arose, both in the new commits as well as in the design studio that was already present priorly, that had to be resolved to make ngUML more accessible and practical for the end users.

Case studies are a fundamental concept in example-driven development. In example-driven programming, programmers start designing software by first analyzing possible inputs and outputs. In this research, the UML diagrams of a case study can be regarded as the input and the desired prototype as the output. When these two are analyzed, an abstract idea can be planned on how to convert the input into the output deterministically. This research used 3 real-world case scenarios. A more detailed description of these case studies can be found below.

### 5.2 Case studies

In total, 3 case studies were conducted to extend the generation part of ngUML. For all three case studies, UML Class, Activity, and Use Case diagrams were built in the environment's design studio. For the first two cases, a target Django application was manually programmed as well. These target applications were built to act as a desired output, which could be compared to the output of the designed generation architecture.

#### Case Study 1: To Do

The first case study mimics a simple to do application in which the user can create and delete lists. Each list has corresponding items which can be created, deleted, or completed. The class diagram consists of two classes: **List** and **Item**. The use case diagram has 12 use cases. The manually implemented target application consists of a single Django application.

#### Case Study 2: Forms

For the second case study a more complex target application was built that mimics a two-user application in which one can create and alter multiple forms, and the other can fill out a form. The editing user can also see the responses that were given to a form. This target Django application consists of four apps: an authentication app, an editor app, a filler app, and a *shared models* app in which the classes are defined which subsequently can be fetched in the other three applications.

## Case Study 3: Webshop

The last case study is the most complex. For this case study, no target application was built. This was done to review the architecture created during the first two case studies. For this last case study, UML diagrams that replicate the design of a webshop were drawn in the environment. The UML use case diagram consists of 3 actors: **Customer**, **Order Manager**, and **Warehouse Staff**. There is a total of 25 use cases, some reachable by multiple actors. The activity diagram consists of 4 main components: authentication and a segment of activities for all 3 different actors. The class diagram consists of 10 distinct classes, including subtyping, inheritance and enums. Besides the diagrams, 4 application components were created in the enhanced studio environment. Application components and this enhanced studio will be discussed in Section 6. The generated metadata was used as input for the new generation architecture.

# 6 Design

## 6.1 Generic algorithm

The already present *ngUML.runtime* repository retrieves a JSON file from the design environment that contains the metadata of a class diagram, use case diagram, activity diagram, and some configuration parameters. It would be favorable to design a generation architecture that solely uses this JSON file as input and outputs a working Django prototype, considering this would be simple to connect to different pipelines. The current design environment, however, does not accommodate information about single pages within the desired output. For example, it is unknown at design time what the order and location of the use cases or actions on a page are. Also, the metadata does not explicitly maintain on what page a specific action or use case must be rendered. This could be resolved by using messy hacks such as merge-point-based page generation on the activity diagram metadata during generation time, but this approach limits the user from being able to alter these pages.

An extension of the metadata is a more effective manner for resolving this problem. New data structures called categories, pages, and section components are added to the metadata, and the design environment is extended to ensure that users can explicitly define the values of these structures. P. de Hoogd explains the implementation of these three data types more thoroughly in his thesis [dH24]. L. Willemsens describes in his thesis [Wil24] how these values can be defaulted using smart defaulting techniques based on the corresponding UML diagrams. Such techniques would prevent the end user from tediously inputting unnecessary values. The set containing all these data types is called an application component. One application component will map to a single Django application within a Django project in the output prototype.

When this metadata is extended, the distinct parts of the metadata can be mapped to the different desired Django output files. For example, classes in the class diagram are mapped to a *models.py* file that contains Django models, and application components are used for the generation of URLs, views, and templates. A valuable advantage of extending the metadata with application components is that the UML diagrams will not need any more information than they already have. When considering the purpose of requirements engineering in model-driven development, which

is to use models that conceptualize a system based on human ideas, it is desirable that the UML metadata only contains fields that are relevant to the corresponding models. By adding application components, the process of designing a user interface is separated from the modeling process.

## 6.2 Implementation of the algorithm

The implementation of the new generation architecture can be found on GitHub [nrg24]. An abstract visualization of the generation architecture can be seen in Figure 1. The root level of the designed generation architecture is a single shell script that has two input parameters: the location of the JSON file that was generated in the design environment, and a name for the output project. When these are given, the script calls a Python method which returns a list of application names for the output. After that, the script runs Django commands to build a Django project and to build a Django application for each application name that was returned by the previously discussed Python script. For each application, additional Python scripts are called that generate one *views.py*, one *urls.py*, one *base.html*, one *styling.css* and multiple HTML templates for each application. One *models.py* file is generated in a *shared\_models* application. These Python scripts do not act on the metadata directly, but rather on abstract datatypes that have been generated based on the metadata. These scripts generate the desired output files by rendering a Jinja2 template using a different data structure for each type of file.

For *models.py* this data structure is a list of classes, attributes, and relations that is generated based on the class diagram. For *urls.py* and *views.py*, this is a list of section components. This section component data structure will be discussed in Section 6.3. Lastly, for the HTML and styling files both the application components that P. de Hoogd discusses [dH24] and its corresponding section components are used for generation. L. Willemsens describes thoroughly [Wil24] how the studio's metadata is parsed to Python objects which are used during the generation process. The generation of this prototype is discussed in Section 6.4.

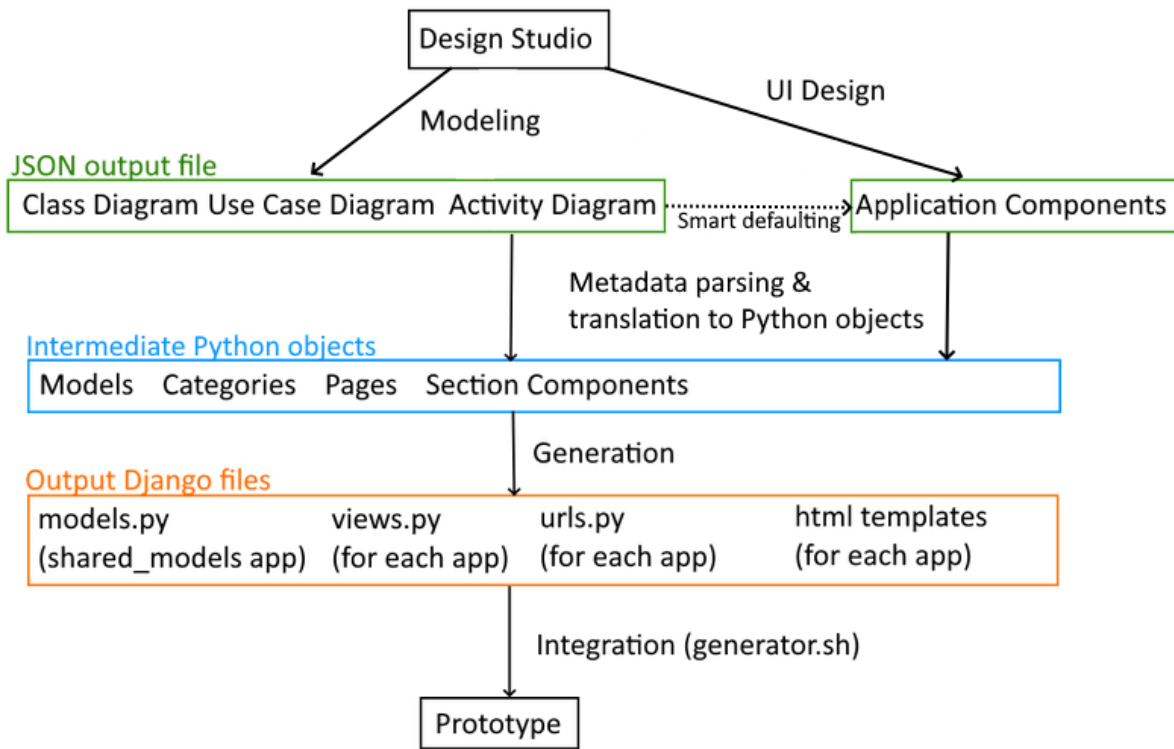


Figure 1: An abstract visualization of the generation architecture.

### 6.3 Section component data structure

A new data structure is used that is used for the generation of *urls.py*, *views.py*, and HTML templates. This data structure is a so-called section component. A section component can be defined as a segment of a page, on which the user can perform one or multiple actions or use cases. A section component has the attributes specified below. L. Willemsens explains [Wil24] how these section component objects are made up by parsing ngUML’s metadata. An example of a section component in the generated prototype can be seen in Figure 2.

#### Name

A sanitized string with the name of the section component. For example, “*viewProductsInShoppingCart*”.

#### App

A string with the name of the application component that the section component belongs to.

#### Page

A string with the name of the page on which the section component should be rendered.

#### Text

A list of free text objects that should be rendered in the section components. A text object has

both a tag type and a content string. For example, *h1* and “*Hello World*”.

### Links

A list of page names. If a page name is present in this list, a button that redirects to this page will be rendered in the section component.

### Primary model

The primary model that the section components act upon. The objects of this model will be shown in the table view of the section component. In Figure 2, *Banana* and *Coffee* are objects of the primary model **Product**.

### Attributes

A subset of the primary model’s attributes. This subset should be shown in the section component.

### Has Create

A Boolean parameter. If this value is true, the section component renders a button that can be used to add a new element to the table of the section component.

### Has Delete

The same as Has Create, but a delete button will be added for each object.

### Has Update and updatable attributes

If the Boolean value of Has Update is true, the objects in a section component can be updated by the user. Only the attributes that are defined as updatable can be updated.

### Is Query and query condition

If the Boolean value of Is Query is set to true, the section component will only contain the objects of its primary model for which the given query condition is true.

### Custom methods

A list of custom methods as defined in the class diagram for the primary model of the section component.

<b>name</b>	<b>price</b>	<b>producttype</b>	<b>inventoryLevel</b>	<b>reorderLevel</b>	<b>availability</b>	<b>Warehouse</b>		
Banana	1	Fruit	1	8	<input checked="" type="checkbox"/>	1		
Coffee	2	Drinks	0	3	<input type="checkbox"/>	1		
								

Figure 2: A section component with read, create, update, and delete functionality.

## 6.4 MVT generation

The previously discussed intermediate data structure as discussed in Section 6.1 are combined with Jinja2 templates to generate the desired output Django files. Jinja2 is a templating language that can be used to dynamically generate files by combining template files with context [Pro24]. Jinja2 was chosen because it is simple to integrate with Python and Django and because of its clear readability and one-to-one mapping to output files. Furthermore, when compared to line-by-line generation that previously has been used in ngUML, templating languages are less likely to crash, easier to maintain, and less prone to errors.

### Models

It has been chosen to define all Django models in one shared application that can be accessed by all other applications. The advantage of this design decision is that underlying dependencies between models and application components can be ignored when generating files. Using this method, only one *models.py* file is generated in a distinct *shared models* application that has the sole use of containing these models. All other Django applications can consequently import these models from this application if they are needed. This prevents duplicate model definitions and no dictionaries that map models to files have to be considered. An abstract visualization of this generation process can be seen in Figure 3.

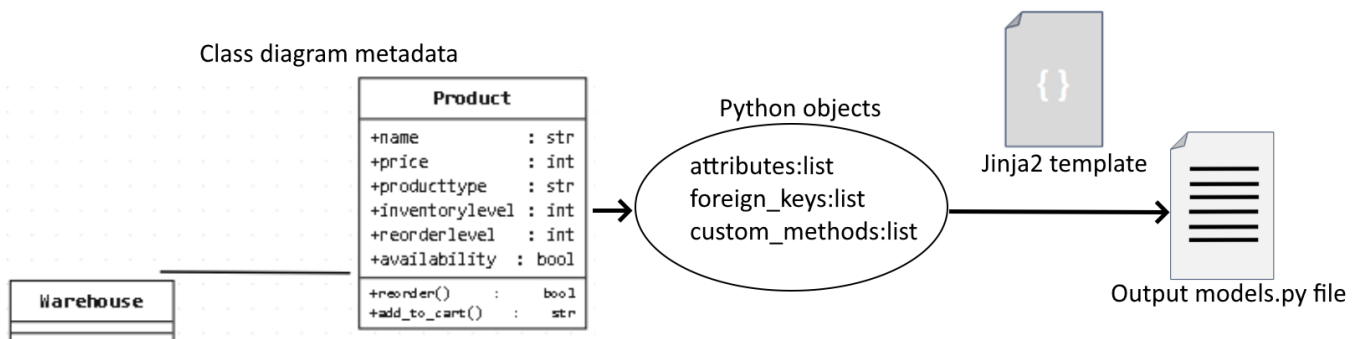


Figure 3: An abstract visualization of model generation.

Individual models are generated for each distinct class in the UML class diagram. Attributes can have the type of string, Boolean or integer. Relations between classes are treated as foreign key attributes. Custom methods can be added to a model by defining them in the UML class diagram using raw code. These raw code snippets are then added to the model in *models.py* by the Jinja2 template.

### Views

One *views.py* file is generated for each Django application that is not the *shared models* application; meaning, one for each application component in the metadata. This is done by combining a Jinja2 template with a list of relevant section components (see Section 6.3). Relevant section components

are section components that have the current application component name as their app attribute. Also, a list of strings containing the page names of the current application is given as context. The Jinja2 template iterates over the section components and generates custom views based on its attributes. After that, it iterates over the list of page names and generates views that render these pages when they are called. An abstract algorithm of the Jinja2 template can be found below.

```
for SectionComponent in RelevantSectionComponents:
    if SectionComponent.HasCreate:
        generate(CreateFunctionality)
    if SectionComponent.HasDelete:
        generate(DeleteFunctionality)
    if SectionComponent.HasUpdate:
        generate(UpdateFunctionality, UpdatableAttributes)
    for CustomMethod in SectionComponent.CustomMethods:
        generate(CustomMethodFunctionality, CustomMethod)

for Page in RelevantPages:
    generate(RenderPageFunctionality, Page)

generate(RenderPageFunctionality, HomePage)
```

## URLs

One `urls.py` file is also generated for each application other than *shared models*. In the same manner, another Jinja2 template file is combined with a list of relevant section components, a list of page names, and a root page. The root page is the name of the home page that should be rendered when the user launches the current Django application.

The Jinja2 template iterates over the section components and creates a URL for all non-read functionalities of a component. Read entries do not require a URL because of their passive nature; they are performed when a page is rendered rather than when a button is clicked. For each page, a URL is generated for rendering this page. Finally, a special URL is generated for the home page.

## Templates

For each application, a Python script is called to generate HTML templates. This script first generates a *css* styling file for the application. Secondly, a *base.html* file which contains a navigation menu and a logo is generated. After that, custom HTML templates are generated by combining a Jinja2 template with a list of relevant section components. The natures of these section components differ based on their parameters. Pseudocode for the generation of templates for an application component can be found below.



```
generate_templates(ApplicationComponent):
    generate_css(ApplicationComponent)
    generate_base_html(ApplicationComponent)
    for ReusableComponent in ApplicationComponent.ReusableComponents:
        generate_reusable_page(ReusableComponent)
    for Page in ApplicationComponent.Pages:
        generate_page(Page)
```

## 6.5 Reusable application components

Reusable application components can be considered for application components that are used frequently and which require more complex functionality than *Created/Read/Update/Delete* (CRUD) operations or custom-defined class methods. A reusable application component is a pre-written application component that might be slightly customized and is copied to the prototype when a Boolean value is set to true in the metadata.

The architecture specified above has been extended to include the possibility of generating reusable application components. Additionally, an example of a reusable application component has been added and can be chosen by the end user in the design studio. This component is a pre-written Django application that handles the authentication of users. In this application, users can log in or be registered. This pre-written application is copied to the prototype if a checkbox "*authentication*" is selected in the design studio. In the prototype, the reusable application component is slightly customized by checking which user types are present in the class diagram. For each type of user, a radio button is added to the registration form. This customization is done by combining a Jinja2 template with a list of user types. A screenshot of the reusable application component as rendered in the prototype can be seen in Figure 4. More reusable application components can be considered in the future.

The image shows a web form with two main sections: 'Login' and 'Register'. The 'Login' section contains a 'Username' input field, a 'Password' input field, and a dark 'Login' button. The 'Register' section contains a 'Username' input field, a 'Password' input field, two radio buttons labeled 'OrderManager' and 'Customer', and a dark 'Register' button.

Figure 4: An authentication form as a reusable application component.

## 6.6 Integration into ngUML environment

The generation architecture has been designed in a new repository that can be pipelined to the design studio. When models and a user interface have been created in the studio, a JSON file containing the metadata is downloaded to the downloads folder of the used web browser. The user must then call the *generator.sh* shell script in the runtime repository and provide the path to the JSON file. A new Django project is then generated in this repository.

## 6.7 Maintaining database entries and abstraction problems

One inconvenience of the current generation architecture is that the prototype is fully generated again after every single edit in the design studio. This causes the database to be empty after each edit, which in return requires the user to tediously enter all data repeatedly after each edit. L. Willemsens describes in his thesis how this problem has been tackled [Wil24].

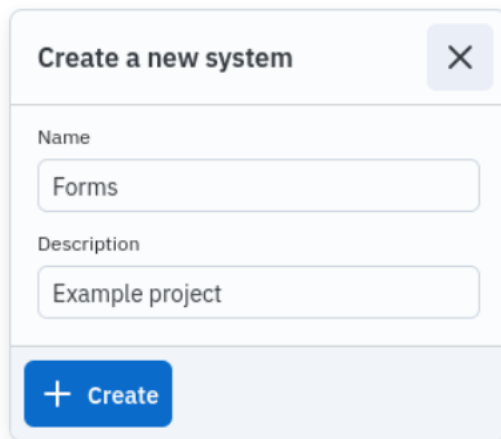
A risk of using this specific generation architecture is that abstraction problems might occur because of the nature of using templating languages. The implemented architecture relies heavily on single templates, and if one of these fails global problems are likely to occur which will not be understandable for the end user. P. de Hoogd describes this risk more thoroughly in his thesis [dH24].

## 7 Worked example

This section will briefly describe how an end user can use the new design architecture to generate a prototype based on UML diagrams.

## Creating a new project

To start a blank project, a new project and a new system must be created by pressing the **Create** button on the design studio project's page.



**Create a new system** [X]

Name  
Forms

Description  
Example project

+ Create

## Modeling

In this newly generated project, UML class, activity and use case diagrams must be modeled using the diagram editor. First, a diagram must be created in the modeling menu. When a diagram is created, its diagram editor can be opened by clicking on the diagram.

Projects · Systems · System (1)

### [system 1]: Model

#### Class Diagram

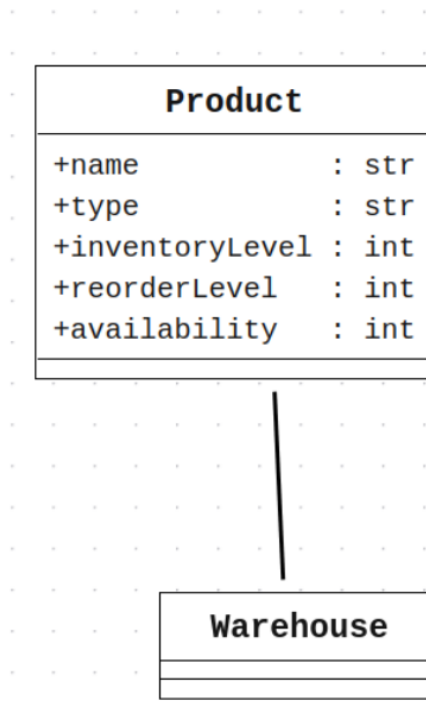
**Initialize**  
Class Diagram

#### Activity Diagrams

**Create New**  
Activity Diagram

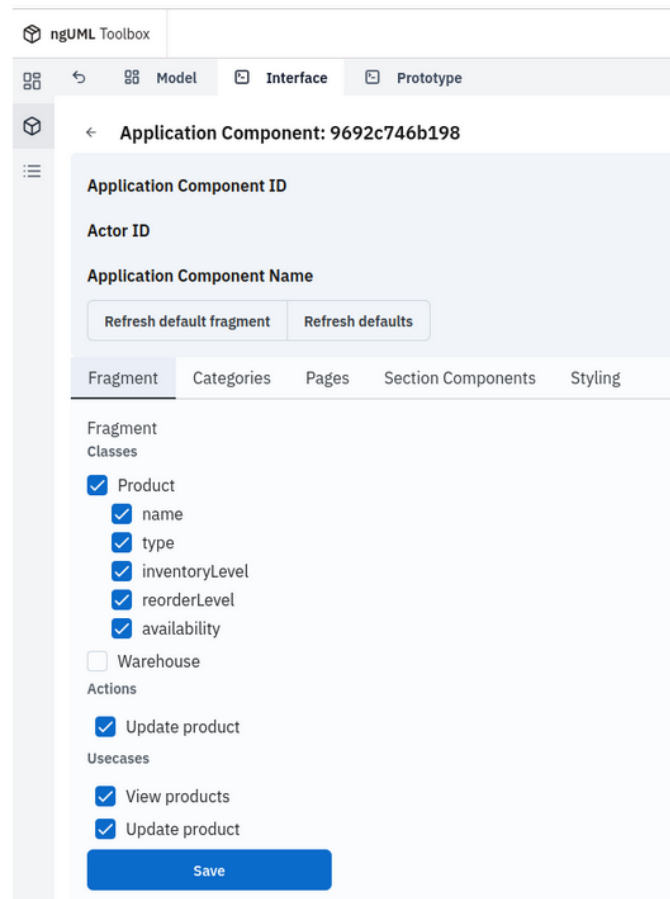
#### Use Case Diagrams

**Create New**  
Use Case Diagram



## Adding a user interface

When the models are done, the user interface can be built under the **interface** tab. The easiest way to implement a user interface is by using the interface that is generated by default. This is done by clicking on the **refresh default fragment** button and pressing the **refresh defaults** button. If the end user wants to tweak the user interface, this can be done under the tabs next to **fragment**. When the interface is done the user can press the **Prototype** tab to download the generated prototype.



## Generation

After the JSON has been downloaded, the prototype can be generated by calling the generation shell script and providing a project name and the path to the downloaded JSON file using the following command in the *ngUML.runtime* repository:

```
./generation_with_path.sh <PATH_TO_JSON> <PROTOTYPE_NAME>
```

## Generated prototype

When the shell script is done, the generated prototype can finally be found under localhost.

### my\_project\_name application

#### Login

#### Register

WarehouseStaff  OrderManager  Customer

## 8 Security

As in all other software development projects, the security of the implementation must be addressed. Security measures should be taken in all steps of the generation architecture to battle faulty corner cases and injections as soon as possible so they cannot trickle down through the process. It can be seen in Figure 1 that there are 4 global streams in the generation architecture:

- *Modeling and UI design*: Design studio → JSON file
- *Metadata parsing*: JSON file → Python objects
- *Jinja2 generation*: Python objects → Output Django files
- *Integration*: Output Django files → Prototype

These 4 global streams each have their security measures as described below.

### Modeling and UI design

Input fields in the design studio are validated before they are added to the JSON file. This validation handles most of the corner cases. For example, integer fields can only contain valid integers, and invalid inputs are thus never used during generation. Strings do not have to be unique and preferably

do not have to be. Double values are given a unique ID in the JSON. Furthermore, delays have been put on some buttons to prevent the overflowing of the API endpoints. Protection against malicious names such as *adminis*, however, not implemented and should be considered in the future.

## Metadata parsing

During the parsing process of the JSON file to the Python objects, validations are performed on the data again. Names are sanitized, and integers are checked to be valid. Custom methods in the class diagram are only checked for syntax errors, but not for malicious injections. The end user must be trusted to not inject faulty code in the class diagram, which will be a problem if the studio is to be deployed.

## Jinja2 generation

Context variables given to Jinja2 templates are not checked for code injections within the templates themselves. Some variables are however sanitized to only contain alphanumeric characters. This sanitation can be extended to all context in the future, and some additional checks could be done within the Jinja2 files as an additional layer of security. Injections should however be protected against firstly in the design studio and during the metadata parsing. Adding more checks in this layer would only be an additional layer of security.

## Integration

The shell script that integrates all output files only has 2 input parameters; the name of the prototype and a path to the JSON file. No checks are performed on these parameters. This might be a problem when the user tries to inject faulty paths. Security has not been considered in this layer because it is a temporary solution, as discussed in Section 6.6.

# 9 Testing

The new generation architecture has been tested by test users outside of the ngUML team to analyze the practicality and straightforwardness of the new system. Different test users with different technical backgrounds have been exposed to the environment and have been asked to answer multiple questions about the system. The script that was given to the test users, the questions that the users have been asked and the answers to these questions can all be found in appendix A and B. A total of 7 test users have been asked for feedback and comments.

## Analysis of the results

Averages and standard deviations of the answers to ordinal assessment questions can be found in the appendices. Considering the small pool of test users, these numerical values must be taken with a grain of salt due to their low statistical significance.

The responses to open questions are more interesting. One of the main points that can be found in all responses is that the practicality of the platform is missing. Most respondents cherish the

abstract idea of ngUML, but the environment requires further improvements and extensions before it can be used in practice. Extensions such as information tiles or a tutorial, automated input in the modeling applications, and automated saving would all improve user experience. It would also be favorable to not require the end user to manually add each actor in the use case diagram to the class diagram. Other notable comments are that the fragment field in the interface editor is unclear, that styling in the prototype needs improvements and that accessibility for, for example, visually impaired end users does not meet accepted standards. The most important and most common point of criticism is that the invocation of the shell script is too difficult and should be replaced with a generation button in the design studio.

## 10 Conclusion

The main objective of this research was to extend ngUML's generation architecture to support model-driven automatic generation of a web application prototype with functional behavioral flow. Multiple real-world scenario case studies were used to devise a new architecture. First, the metadata was extended with the Application Component and its child data structure, the Section Component. Intelligent manners of defaulting the values of these new data structures based on the metadata in the corresponding UML diagrams were implemented, and the design studio was updated to provide the end user with the means to edit these values and create new components. Multiple scripts were written in Python to parse this metadata into abstract objects and the generation architecture was extended with the templating language Jinja2 so the abstract Python objects could be combined with templates to generate Django output files for the prototype. Finally, a shell script was written that combines all formerly said operations.

The extended architecture is effective in generating simple prototypes that can have multiple applications and pages, on which multiple classes can be operated using either *CRUD* operations or custom-written operations. The generation of prototypes is quick and consistent. Practicality is however still lacking before real end users can use the environment to generate real prototypes. The design studio is confusing for the end user and guidance is missing. The end user is also limited to *CRUD* operations if no custom-written methods are manually added in the class diagrams, and these methods are likely to fail due to syntactic or behavioral errors. The end user is also required to explicitly follow naming rules when creating use cases, which is not favorable when taking into account that use cases should be described using natural human language. The shell script should be fully integrated into the design studio and the end user should not use any terminal. In conclusion, the extended architecture is effective but still requires enhancements in multiple fields before it can be used in practice.

## 11 Future work

The newly implemented generation architecture can be broadly extended in all different aspects and repositories. Future work is to be divided into two main categories: **technical** improvements of the prototype and environment and **abstract** extensions of the architecture. Some ideas for future work that fall under these categories can be found below.

## 11.1 Technical improvements

### Integration in the studio

As described in Section 6.6, the user must manually run the shell script for each generation. Preferably, the generated project should be reachable within the studio. This could be done by adding a generate button rather than a download button which then generates the JSON file and pipelines it to the runtime repository. The shell script must then be called automatically, and the user should be redirected to the port that hosts the generated project.

### Dynamic pages

The current output files are solely using Django without JavaScript or Ajax. A result of this is that for each action, the website must be rendered again. Due to this, a call is performed to the database after each click of a button in the prototype, which becomes problematic for larger datasets. In the future, dynamic pages could be generated to avoid this problem

### Docker deployment

There is currently no deployment for the prototype. The end user must have Django installed for the prototype to work correctly. In the future, the prototype could be wrapped and deployed in a Docker container to ensure stability.

### A structured API

The current generation architecture only generates Django views files that operate directly on the database. It would be neater to use a more structured API by using something like Django REST.

## 11.2 Abstract extensions

### Non-CRUD functionalities in section components

As of now, section components can only have standard CRUD buttons that act on the primary model of the section component. Buttons for custom methods can be rendered, but these custom methods must be written explicitly in the class diagram by the end user. If these custom methods contain errors, which are likely to occur, the prototype will crash when its corresponding button is clicked. Future developers could try to either deterministically generate such custom code using components or use a non-deterministic *large language model* (LLM) prompt to generate the code. S. Saqlain Zeidi [Zei23] has investigated this before in another context, but his work could be used as a reference. Activity diagrams could also be potentially very useful to deterministically generate custom methods.

### Queries

As of now, section components can only be queries on a single attribute of the primary model. Preferably, queries should be implemented using a recursive formal language. Such a formal language can be found below. This formal language is based on a paper by P. Guagliardo and L. Libkin [GL17] in which the semantics of SQL queries are described.



- Query  $\rightarrow$  Comparison
- Query  $\rightarrow$  Negation
- Query  $\rightarrow$  Logical
- Comparison  $\rightarrow$  Leaf Operator Leaf
- Operator  $\rightarrow$   $>/</</<=>/=/==/$ SUBSTRING
- Leaf  $\rightarrow$  Value/Attribute
- Logical  $\rightarrow$  Query AND Query
- Logical  $\rightarrow$  Query OR Query
- Negation  $\rightarrow$  NOT Query

Python classes for this formal language have already been defined in the project, but the design studio would have to be extended to accept such a language. Smart defaulting methods based on the diagrams should also be considered to prevent the end user from manually defining large query trees. The Jinja2 templates must also be extended to accept such a formal language.

## References

- [dH24] P. de Hoogd. An effective user experience for prototype generation in model driven engineering. *bachelor thesis at LIACS*, 2024.
- [Dri20] R. Driessen. Uml class models as first-class citizen: Metadata at design-time and run-time. *bachelor thesis at LIACS*, 2020.
- [ea05] S. Burmester et al. Model-driven architecture for hard real-time systems: From platform independent models to code. 2005.
- [ea18] J. Cabot et al. Cognifying model-driven software engineering. 2018.
- [ea19] U. Sabir et al. A model driven reverse engineering framework for generating high level uml models from java source code. 2019.
- [ea21] Rasha Gh. Alsarraj et al. Designing and implementing a tool to transform source code to uml diagrams. 2021.
- [GL17] P. Guagliardo and L. Libkin. A formal semantics of sql queries, its validation, and application. 2017.
- [Gro14] Object Management Group. Mda guide revision 2.0, 2014. Accessed on April 14th, 2024. Available at: <https://www.omg.org/cgi-bin/doc?ormsc/14-06-01>.
- [Gro17] Object Management Group. Omg unified modeling language specification, 2017. Accessed on April 5th, 2024. Available at: <https://www.omg.org/spec/UML/2.5.1/PDF>.
- [HS13] B. Hussein and A. Salah. A framework for model-based code generation from a flowchart. 2013.
- [MG19] M. Mukhtar and B. Galadanci. Automatic code generation from uml diagrams: The state-of-the-art. 2019.
- [nrg24] ngUML research group. nguml.runtime repository, 2024.
- [Pro24] Pallets Projects. Jinja documentation, 2024. Accessed on April 5th, 2024. Available at: <https://jinja.palletsprojects.com/en/3.1.x/>.
- [vA22] B. van Aggelen. Enabling data-driven wireframe prototyping using model driven development. *bachelor thesis at LIACS*, 2022.
- [Wil24] L. Willemsens. Utilising use case models for multi-actor prototype generation. *bachelor thesis at LIACS*, 2024.
- [Zei23] S. Saqlain Zeidi. Synergizing uml class modeling and natural language to code conversion: A gpt-3.5-powered approach for seamless software design and implementation. *bachelor thesis at LIACS*, 2023.

# A Testing questions and responses

## A.1 First questionnaire

1: Rate the ease of use for generating a prototype using the AI4MDE platform (1-5)

Average: 4.57

$\sigma = 0.49$

2: How clearly are the different steps in the generation process (MODEL - INTERFACE- PROTO-TYPE) presented? (1-5)

Average: 4.14

$\sigma = 0.99$

3: How effective is the AI4MDE platform in aiding users to generate prototypes? (1-5)

Average: 4.00

$\sigma = 0.53$

4: How accessible is the prototype generation process using the AI4MDE platform? (1-5)

Average: 4.00

$\sigma = 0.76$

5: Do you have any suggestions that could help increase clarity during the prototype generation process?

- Duidelijkere omschrijving over wat je nou eigenlijk aan het doen bent. Bijvoorbeeld bij het uitvoeren van de json file. Tussen proces tussen design en generatie makkelijker maken.
- Als je over een item heen gaat met je muis tips weergeven of uitleg wat er precies gebeurt als je dat item invult.
- Accessibility kan beter, vooral links en tab navigatie is karig. information icons bij actie elementen voor verduidelijking, kleine tutorial voor nieuwe gebruikers
- Before generating, I was not asked what I would like to have in my webshop. I would like to have a sort of shopping list of items I would like to see in my webshop. Now, I had nothing to say what would be generated. Seems like every generated webshop would look the same.
- Misschien uitleg over hoe je de diagrammen kunt ontwerpen. Wat mogelijk is en wat niet
- The application does not update automatically. The generation button action could be integrated in the platform.
- Voor de UI van de editor geldt het weglaten van informatie die die bijdraagt aan de app (onbruikbare informatie kan verwarrend werken, zoals: Application ID's). Voor de UI van de App geldt dat het opschonen van de links en knoppen het een nettere app maakt. Ook kan het behulpzaam zijn als er hover-text bij links en knoppen zichtbaar wordt met een korte omschrijving van de te verwachte actie van die link/knop. DB inhoudelijk is er ook nog wat ontwikkeling nodig zoals valuta records, maar dat is afwerking. Zou het mogelijk zijn om

ook een structured files te gebruiken om je webshop DB als manager eenvoudig te kunnen (bij)vullen?

## A.2 Second questionnaire

1: How effective are the diagrams in displaying what the prototype should do? (1-5)

- Class diagram

Average: 4.29

$\sigma = 0.70$

- Use Case diagram

Average: 4.14

$\sigma = 0.35$

2: How well were the changes made in the diagrams reflected in the updated prototype? (1-5)

- Class diagram

Average: 3.86

$\sigma = 0.99$

- Use Case diagram

Average: 4.14

$\sigma = 0.64$

3: How effective is each part of the interface step? (1-5)

- Fragment

Average: 4.00

$\sigma = 1.07$

- Categories

Average: 3.86

$\sigma = 1.12$

- Pages

Average: 3.71

$\sigma = 0.88$

- Section Components

Average: 3.43

$\sigma = 0.90$

- Styling

Average: 3.71

$\sigma = 1.03$

- Do you have any suggestions that could help increase clarity during this step?

- Als je iets aanpast zou het fijn zijn om te zien wat dat voor gevolgen gaat hebben voor andere componentjes
- Zorg ervoor dat de gebruiker zijn veranderingen en acties kan terugleiden, save knoppen zijn inconsistent of onduidelijk te vinden. user action validation/ error warning. spatie wordt uit strings gehaald dus bij prototype afwezig. interaction feedback is wel focus waardig
- Indicate which choices a user can make beforehand. Also: it would be very helpful if a user could see the changes it makes immediately. Such as: if I choose the colour blue: what is going to be blue? That was not clear from the outset on.
- Het verschil tussen de componenten is duidelijk en het is makkelijk om in een iteratief proces met prototypes het gewenste effect te krijgen.
- Changes made in diagrams should reflect in these pages and vice versa. Categories and Pages could be a single page. Design should not be able to divert from diagram restrictions.
- Nog te weinig gevoel bij. Meer mee werken om goed te kunnen begrijpen wat de invloeden zijn op het eindresultaat.

4: How worthwhile is the addition of support for rules or queries in the generated prototype? (1-5)

Average: 4.57

$\sigma = 0.49$

5: How worthwhile is the addition of support for the unused activity diagram to dictate the flow of a prototype? (1-5)

Average: 4.00

$\sigma = 0.53$

6: How worthwhile would it be to use Categories in the sidebar of the prototype to group pages based on the data accessed on the page? (1-5)

Average: 4.00

$\sigma = 0.76$

7: Do you have any other feature in mind that is missing in the current form of AI4MDE, which you would have liked to see?

- Misschien nog verder automatiseren doormiddel van een script die vraagt wat je wilt aanpassen en dit dan dus ook voor je doet
- Zou mooi zijn als je bij de interface direct vkan zien hoe het er uit komt te zien, ivm juiste kleurkeuze
- tutorial, documentation/help, user log of actions and events

- Generate the website as you work on it in like a side screen would be very helpful to see exactly what/how my choices influence the website
- Ability to pick a subset of Class Diagrams as it could get quite large and ineligible. Color coding or visual differences in diagrams.
- Structured data uploaded

8: Rate the quality of the default values based on changes in the diagram found in the interface step. (1-5)

Average: 4.14

$\sigma = 0.64$

9: Rate the quality of the generated prototype.

Average: 3.57

$\sigma = 1.18$

10: Would you use AI4MDE to design and generate prototype software?

- Ja, je begint al met de basis opmaak en vanuit daar ga je verder bouwen. Je hoeft dus niet helemaal vanaf het begin te beginnen.
- Jazeker
- ja, ik geloof dat het concept van de applicatie en de huidige uitwerking een zinnige uitbreiding kan zijn voor iemand die bezig is met database architectuur. De diagram uitwerking omzetten in een prototype met zoveel gemak is duidelijk van waarde.
- In the future maybe. Main improvements: more explanation on the choices that could be made and sidebar with generated website
- It is very easy to create prototypes, so based on the diagrams it is easy to see whether they would work in a prototyping setting. Making this a good tool to verify the created diagrams as well.
- Definitely if you implement my suggestions.
- Nog niet... Behoeft nog wat werk voor onge oefende gebruikers.

## B User script

The user script that was used for testing contains guiding illustrations for every instruction noted below.

1. Go to the design studio: <http://localhost:5173/>
2. On the home page, click on “Get started”
3. Select the already present “Testing System” at the systems page of this project.
4. In this system, you can find the modeling diagrams corresponding to this prototype. We will not edit these now, but feel free to take a look at them.
5. Click on the “Interface” tab. This tab shows a list of different user interfaces for different types of users. You can inspect them, but do not edit them yet.
6. When you are done inspecting, press the “Prototype” button. A JSON file containing the information of the prototype will now be downloaded to your downloads folder.
7. Generate the prototype by running the generation script and providing a name for your prototype and the path to the previously downloaded JSON file. Please ask your instructor for help if this process is unclear.
8. After this generation is done, your prototype can be inspected. Go to: <http://127.0.0.1:8001/> to inspect the newly generated Webshop.
9. Now please answer the questions on the first page of the questions sheet.
10. Go back to the page containing the models of your prototype. We will first change the UML Class Diagram corresponding to this project. Do this by clicking on the “Class Diagram” tile.
11. The Class modeler will open. You can edit classes by clicking the right mouse button on a class (a rectangular box) and then clicking “Edit”.
12. A popup window will open. Try adding a new attribute by pressing “Add Attribute”.
13. A new attribute will appear. You can give this attribute a name (no spaces!!!) and a type.
14. When you are done press “Save”.
15. We will now add a new class. Right click on the empty canvas and click “New Node”.
16. A popup window will open. Select type “Class” and give the class a name. When you are done press “Add”.
17. Give the new class new attributes like you have done in steps 12-14.
18. We will now connect the new class to an already existing class. Right click the new class and press “Connect”.

19. A new line will follow your cursor. Left click a class you want to connect the new class to.
20. A new popup window will open. Select type “Association” and press “Add”.
21. The new class is now connected to the model. We will now leave the Class Diagram editor and start editing the Use Case Diagram.
22. On the System page, left click the “Use Case Diagram” tile to start editing the Use Case diagram.
23. The Use Case diagram editor will open. We will add a new use case for our new class by right clicking the canvas and pressing “New Node”.
24. A popup will open. Select type “Use Case” and name it “View” + the name of the class that you just created + “s”. When you are done click “Add”.
25. We will add the new Use Case to an existing actor. Right click the Use Case and click on “Connect”.
26. Connect the Use Case to an actor and select type “Interaction” in the new popup. Click “Add” when you are done.
27. The Use Case is now connected to the actor. We will also add a new Actor. Right click the empty canvas and click “New Node”.
28. In the popup select “Actor” and give the new Actor a name. When you are done click “Add”.
29. The new Actor has been created. Do not forget to also connect it to the new Use Case using an interaction like you did in step 26.
30. The new Marketing Manager actor should also be able to create new Discount Coupons. Create a new Use Case with the name “Create” + the name of your new class. Connect it to the new Actor using an “Interaction”. Then connect the Create Use Case to the View Use Case using an “Extension”. (From: Create, To: View).
31. Finally, we must add a new class as a subtype of user for our new actor. Create a new class with the name of the actor and connect it to “User” using a “Generalization”.
32. The models are done. We will now start editing the user interfaces by clicking the “Interface” tab in the system menu.
33. We can see that a new user interface has been automatically created for the newly added Actor. This interface is still missing some information. We will add this by pressing the “Edit” button.
34. Before we can add a user interface, we have to specify a fragment of the modeling diagrams that is relevant for this interface. We will use a default fragment by pressing the “Refresh default fragment” button.



35. Next, we will press the “Refresh defaults” button to generate standard Section Components and Categories. Feel free to take a look at these after you have pressed the button.
36. We will finally build a page on which the new actor can create new Discount Coupons. Go to the “Pages” tab and fill in the fields of a page. Leave the two numbers on 1. Select the “Create” Section Component. When you are done click “Add”.
37. If wanted, you can edit the styling for this user interface under the “Styling” tab.
38. When you are satisfied with everything, click the “Prototype” tab to download the new JSON file. Repeat steps 7-8 to generate the new prototype.
39. In the landing page we can see that a registration button for our new Actor was added. We can also see that a new user interface was generated in which the Marketing Manager can create Discount Coupons.
40. Feel free to play around more with the user interfaces and generate other prototypes before proceeding to the questions.
41. Please fill in the questions on the second page of your question sheet. Thank you for your service!