Universiteit Leiden
The Netherlands

Opleiding Informatica

An effective User Experience for Prototype Generation in Model Driven Engineering

Pieter de Hoogd, s2958392

Supervisors:
Guus Ramackers & Joost Visser

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 18/04/2024

Acknowledgements

This thesis would not have been possible without several individuals whom I would like to thank. First I would like to thank my supervisors Guus Ramackers and Joost Visser for their support, motivation, and insights. Secondly, I want to thank Max Boone for his insights and assistance in this project. Lastly, I want to thank Lucas Willemsens and Sven van Dam for their contribution and cooperation.

Abstract

In this bachelor thesis, the goal is to improve and implement design-time support for a prototype generator. This project is an extension of a larger project: Prose-to-Prototype (P2P), started by the AI4MDE organization. With two other students, we will extend and improve the already existing functionality. The focus of this research lies in the improvements of the design and run-time for the prototype application generation. We will evaluate multiple alternatives with the help of 'test users' to empirically identify the best solution.
By extending existing metadata, creating a user-friendly interface, and adding a smart default technique to minimize user input, this study tried to improve and enhance shortcomings in the AI4MDE platform. Test users expressed overall positivity towards the implemented solution, highlighting its efficacy in addressing key concerns. However, their feedback also underscored several areas for improvement, suggesting avenues for refining this solution's functionality.

Contents

1 Introduction

This bachelor thesis investigates the design and implementation of a prototype generator that uses metadata of diagrams in the Unified Modeling Language (UML). This project will be part of the larger Prose-to-Prototype (P2P) project at Leiden Institute of Advanced Computer Science (LIACS). Next to this thesis, there are two other bachelor theses which together form a large main project. The project will extend and improve what has already been designed and programmed by other students in earlier years. This thesis will primarily focus on improving an interface to support the generation of prototype applications.

1.1 Context

This project uses a framework which is provided by the AI4MDE organization, formerly ngUML. In this organization, work is already done by students in earlier years. That means, AI4MDE provides a framework in which this project can fit. The overarching aim of this collective research is to advance the AI4MDE project by developing an open-source platform that guides developers and entrepreneurs in designing systems, leveraging UML metadata for enhanced prototype generation. Additionally, the aim is to investigate and implement contemporary Software Engineering (SE) techniques, contributing to the broader SE literature. Through practical evaluation with user testing, the goal is to refine and validate the solution, ultimately creating a comprehensive solution for seamless system development from conceptualization to prototyping. This particular part of the research aims to develop a user-friendly interface.

1.2 Problem Statement

How can the design-time and run-time shortcomings in prototype application generation be effectively addressed and improved to enhance overall usability and user satisfaction?

What metadata is needed in prototype application generation to provide a user-friendly environment?

Which alternative solutions yield the most substantial benefits as determined through empirical evaluation with test users?

To answer these questions, research is done in the AI4MDE framework. This framework consists of a Natural Language Processor (NLP). This part of the framework is able to translate user input to UML diagrams. Next to the NLP, there is support for creating and adjusting diagrams by dragging, dropping, and connecting elements of the diagram. Additionally, with a runtime modeler, some basic web pages in the application could be made. However, this solution was not user-friendly and needed numerous adjustments to be used by business owners or software developers for example. Furthermore, the final functionality that could be generated by this solution missed some major parts to be a real prototype.

1.3 Example Driven Development

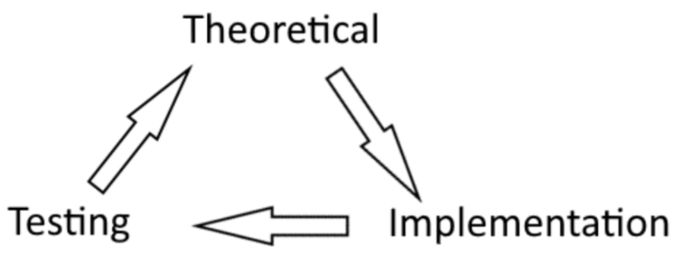


Figure 1: Phases of Example Driven Development

This project relies on the Example Driven Development approach. Each period in the Example Driven Development approach has three phases: a theoretical phase, an implementation phase, and a testing phase. When using models (in this case UML diagrams) Example Driven Development is an efficient solution in the process. Example Driven Development uses the power of abstraction and automation to streamline development processes. Making use of the testing phase, the efficiency and robustness of the final software will be improved [SJM03]. The choice to utilize Example Driven Development is particularly advantageous in this context as it facilitates the synchronization of progress and complications across the three concurrent projects.

1.4 Methodology

To implement example-driven development in the project, a structured approach was adopted involving three case studies of increasing complexity. Initially, a simple case study was undertaken to introduce the concept and establish foundational understanding. This case study was also important to understand working with the AI4MDE platform. The next case studies continued to build on the previous one with extensions and improvements. Within a case study, weekly meetings with S. van Dam and L. Willemsens were done to keep up with the progress and to learn from each other's knowledge and insights.

- Case study 1: To-do list application
  A simple application where users can create multiple to-do lists. Users can add, remove, or check items (tasks) and set a priority for each task.

- Case study 2: Form application
  Multiple-user application: a user identifies as a filler or editor. Fillers can only answer questions on forms and editors can create, edit, and remove forms. Editors also have insights into the answers.

- Cast study 3: Webshop application
  Complicated web pages with multiple sections on one page. With custom methods and queries, advanced web pages can be generated.

1.4.1 Overview

The work of the overall project was divided into three parts: each part corresponds to this thesis, L. Willemsens' thesis [Wil24] or S. van Dam's [vD24] thesis. In short, the overall project consists of the extension of the 'design studio' from AI4MDE (that what existed already in the platform). From the design studio, metadata could be generated which had to be parsed into the generation process. This generation process uses Django to generate a prototype application. In addition, the choice was made to implement a smart defaulting technique so user input could be minimized. In figure 2 an overview of the overall project is given. This thesis gives information and insights on the *Design Studio* and the green blocks.

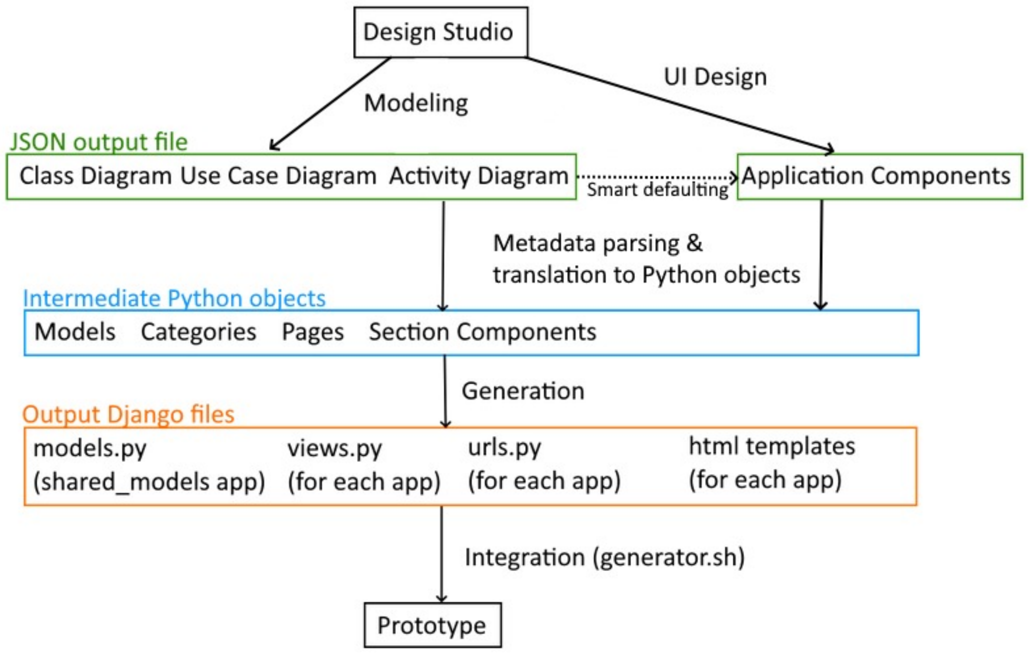

Design Studio
Modeling
UI Design
JSON output file
Class Diagram Use Case Diagram Activity Diagram
Smart defaulting
Application Components
Metadata parsing & translation to Python objects
Intermediate Python objects
Models Categories Pages Section Components
Generation
Output Django files
models.py (shared_models app)
views.py (for each app)
urls.py (for each app)
html templates (for each app)
Integration (generator.sh)
Prototype

Figure 2: Overview of the overall project

2 Background

In this thesis, there are some foundational concepts to understand each part:

- Unified Modeling Language (UML)
- The AI4MDE Project
- User Interface (UI)
- User Experience (UX)

In the next subsections, information is given about these concepts.

2.1 Unified Modeling Language (UML)

The Unified Modeling Language is a language for structuring and modeling software systems. The idea is that every software system consists of UML diagrams. In this thesis, we focus on three of the many diagrams that UML provides: Class diagrams, Activity diagrams, and Use Case diagrams. From these diagrams, a metadata model can be created that gives us information about what kind of software customers want [SO08].

2.1.1 Class Diagrams

Class diagrams, as it is in the name, are used to define class objects that have to be in the software. Every class has attributes, such as *name* or *priority*, and operations that users can perform on the class, such as *getName()* which will return the name. An example of a class diagram for a simple to-do list is given in Figure 3. Class diagrams are the most common UML diagrams to use in software [KEBP21].

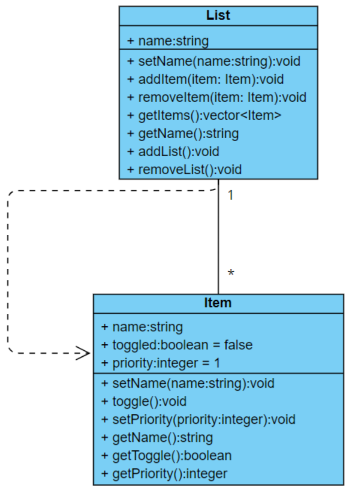

List
+ name:string
+ setName(name:string):void
+ addItem(item: Item):void
+ removeItem(item: Item):void
+ getItems():vector<Item>
+ getName():string
+ addList():void
+ removeList():void
1
*
Item
+ name:string
+ toggled:boolean = false
+ priority:integer = 1
+ setName(name:string):void
+ toggle():void
+ setPriority(priority:integer):void
+ getName():string
+ getToggle():boolean
+ getPriority():integer

Figure 3: Class diagram of a simple to-do list application

2.1.2 Activity Diagrams

Activity diagrams show different states of the software system. They also show how users come from one state to another: the flow. Every node in the diagram represents a state and the lines represent transitions which you can correspond to clicks in an application for example. An example of an activity diagram for a simple to-do list is given in Figure 4.

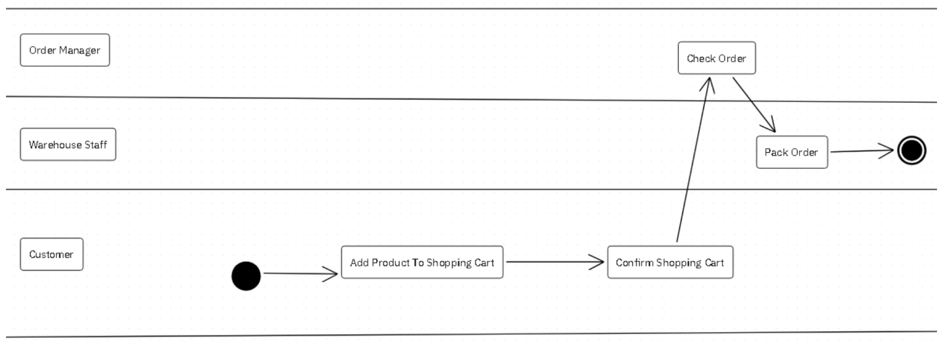


Figure 4: Activity diagram of a webshop application

2.1.3 Use Case Diagrams

Use Case diagrams are used to understand the functionality of software systems. It consists of actors who can perform actions. This corresponds to users who can perform actions in the application. An example of a use case diagram for a simple to-do list is given in Figure 5.

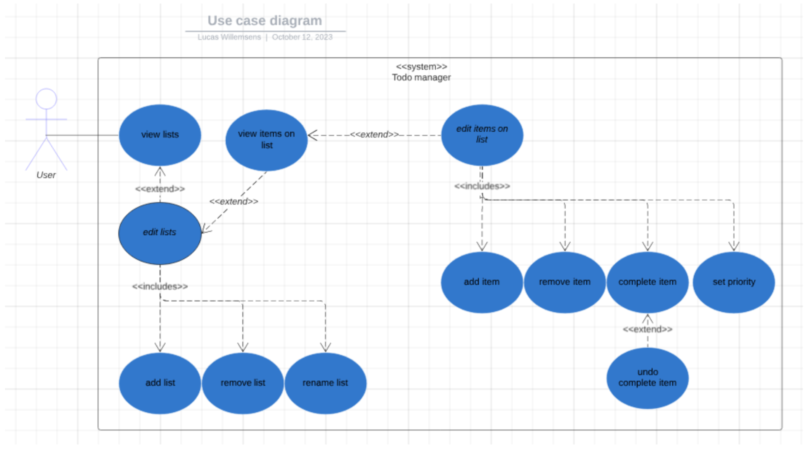


Figure 5: Use Case diagram of a simple to-do list application

2.2 The AI4MDE Project

The AI4MDE project is a software research project at LIACS (Leiden Institute of Advanced Computer Science). It has the goal to automate software development. Currently, programming new software is hard and requires abundant time and effort. Another problem is the fact that it is not always fully clear what the customer wants. The AI4MDE project aims to make the process of creating new software more efficient.
To do this, UML diagrams are generated from speech-to-text using NLP models. The diagrams are stored in JSON format files. From these JSON files, a web-based prototype can be generated [Ram21].

2.3 User Interface (UI) and User Experience (UX)

The concepts UI and UX of a software system are about usability. The user interface (UI) on one hand is the layout for software systems or applications. The user experience (UX) on the other hand is more about the entire experience of users including emotions, perceptions, and satisfaction.

Expanding on the importance of UI and UX in software systems, these concepts primarily revolve around usability. The UI encompasses the design and layout of software systems and applications. It serves as the visual gateway for users, influencing their initial interaction and engagement.

On the other side, the UX delves deeper into the overall experience users have with a system. Beyond visual elements, UX is about emotions, perceptions, and overall satisfaction throughout the user journey. It goes beyond the surface, focusing on creating an experience that not only attracts users but also keeps them satisfied and coming back.

In essence, while UI acts as the visual ambassador, UX takes a comprehensive approach, shaping the user's journey and perception. Both UI and UX are integral facets in ensuring not only the initial allure of a software system but also its enduring success through user satisfaction. As such, a harmonious integration of UI and UX principles is important for crafting software systems that stand out in today's competitive landscape[Pra20].

3 Related Work

3.1 UML Metadata

A thesis by R. Driessen laid the foundation for the current metadata which is used for prototype generation in the UML application. This foundation, later extended by other students, was excellent for simple prototype applications but had no support for user experience[Dri20]. An overview of the metadata used in the work of R. Driessen can be seen in figure 6

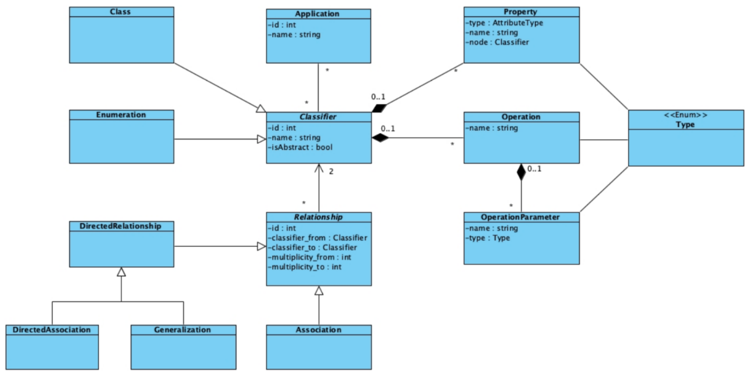


Figure 6: Foundation for the metadata[Dri20]

As shown in figure 6, the metadata consists of classifiers that are generated from a class diagram. From there on, a prototype can be generated. Although this solution was robust and efficient, the need for more functionality and a better-looking UI became bigger within AI4MDE.

3.2 Wireframe Prototyping

The data structure of R. Driessen[Dri20] was adjusted by B. van Aggelen with 3 new subclasses as seen in Figure 7. These subclasses were important for user experience in the design-time choices. First, a new subclass Category was added for the grouping of pages so that pages are easier to find. Secondly, two subclasses Page and Section were added for the styling of different pages in the generated application. A 'Page' was described as a collection of sections, displayed on a single HTML page. A single section was described as a collection of HTML elements and data that can be parsed[vA22]. A good overview of this is given in Figure 8.

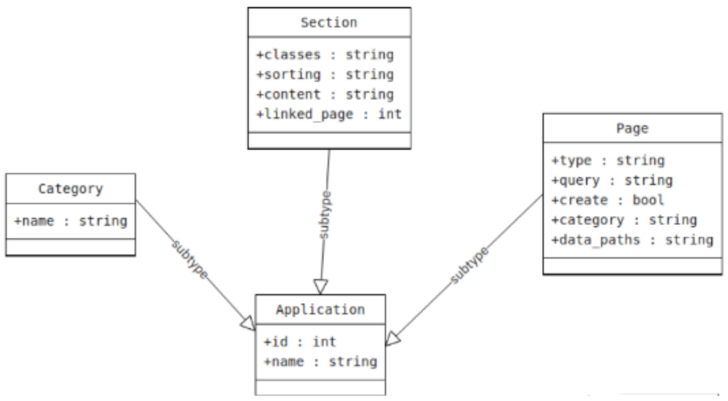


Figure 7: Extension of the existing datastructure by B. van Aggelen[vA22]

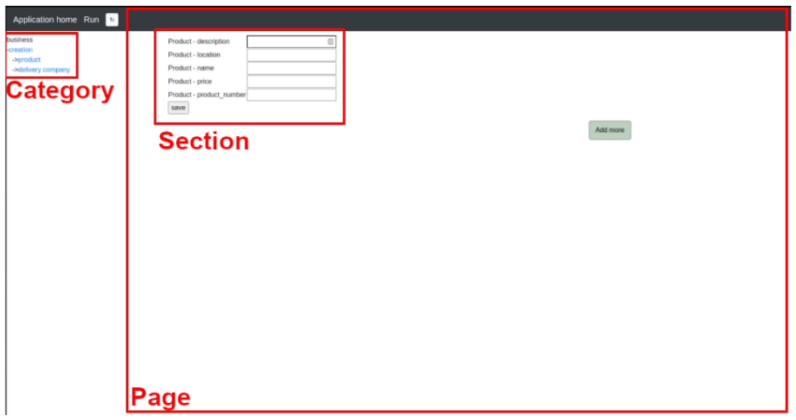


Figure 8: Overview of metadata on a generated webpage[vA22]

The AI4MDE project aims to empower users to effortlessly create their own applications by offering them the freedom to make choices regarding various sections. Users are granted the flexibility to determine the display positions of HTML elements, facilitated by a 'What You See Is What You Get' (WYSIWYG) editor. This tool enables users to easily insert and position HTML elements, allowing them to personally design the application according to their preferences.

The extension by B. van Aggelen was robust but needed many improvements for a real user-friendly application. The main problem was that the WYSIWYG editor was hard to use for customers without code experience. In addition, the generated pages could be improved in style.

3.3 Django and Jinja2

For the generation process, a combination of Django and Jinja2 can be used. Django is a web framework for Python and makes it accessible to create web applications. Django has numerous useful things for developers such as an admin interface for applications, support for databases, and the possibility to generate web pages [Dja].

Jinja2 is a template engine that allows it to write custom Python code. These templates can be rendered in Python which makes it possible to render HTML or CSS with custom Python code. This is very useful for this project because with template-driven development it is possible to generate advanced web applications [Jin].

The use of templates allows for a flexible and customizable approach during generation. Layouts are specified, without requiring additional code. Similarly, hierarchical layouts can be automated using templates. This allows software developers to create and iteratively improve complex layouts without having to deal with the technical details of implementation [PEH23].

4 User Interface Specification

The AI4MDE application initially provided support for creating the relevant UML diagrams: class, activity, and use case. This application is called *The Interface*, previously referred to as *Design Studio* in this thesis. The interface had to be extended with the integration of advanced user interface (UI) and user experience (UX). Before this thesis, support for UI and UX was very minimal. Users could add pages and sections with a WYSIWYG editor, but this solution was not user-friendly: Users were writing raw HTML code. To solve this problem, many adjustments were made to the metadata and the interface. The following section provides information about:

- How the interface metadata was extended
- Which new functionality was added to the interface
- A smart defaulting technique to minimize user-input
- Insights about information exchange in the system
- Validation to guarantee a correct output for generation

4.1 Metadata adjustments

First, some adjustments to the latest version of the metadata (see Figure 7) were made. The application model was renamed to *Application Component* which initially contained categories, page styling, and sections. Categories kept the same function as first: a fluent navigation between pages in the application. The 'page' component not only consisted of single pages anymore but also contained styling elements like background color or text alignment. The most complex component, sections, was totally refactored: the WYSIWYG editor was removed and an advanced system to create a user interface was implemented. Sections would work with rows and columns to place them on, called a grid. More about this grid is explained in section 5.4.2. An overview of the described new metadata is given in figure 9.

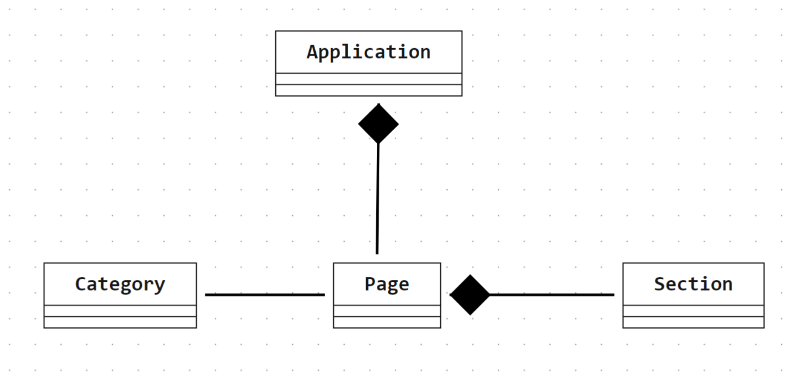


Figure 9: New metadata model: *The Application Component*

Insufficient information within the sections of the application component impeded the generation of advanced web pages. To solve that problem, a new component was added: Functionalities. Functionalities would describe the operation on a class on an attribute. Operations were based on CRUD: Create, Read, Update, and Delete [cru]. In addition, an advanced Read was needed to implement charts or show limited items of a class attribute. A Functionality could handle a query that could be typed in by the user. So, eventually, a functionality was part of a section, which made a section a more complex component. Therefore, a Section was renamed to a Section Component because a section now contained a lot of information.

Because there was more information needed from the diagrams, the choice was made to connect all items from the diagrams to the Section Components. So, a Section Component now had classes, attributes, actions, and use cases to operate on. Together with linked functionalities, it formed a strong information component to use in generation. For example, when a user creates a class *Order* and a use case and action *Create order*, the Section Component would exist of class *Order*, action *Create Order* and use case *Create Order*. The related functionality would be a *Create* operation on class *Order*.

The next issue that came up was that we had to create multiple application components for multiple users in the application. Thinking of a webshop, there would be an application for an *Order Manager*, a *Customer*, and *Warehouse staff* for example. The problem was that not all users had to access the same classes, actions, and use cases. Therefore, a new component: Fragment was defined. A fragment is a part of the diagrams that could be used in Section Components and Functionalities. *Warehouse staff* for example have to see every attribute from class *User* (name, address) but a *Customer* does not have to see all other customer data from class *User*.

Because users now had to define various properties in different components (Section Component, Functionality, Fragment, etc.) the choice was made to integrate Functionalities into the Section Component. This made the Section Component more powerful and user-friendlier because the user input was reduced. In the end, the final metadata model can be seen in figure 10.

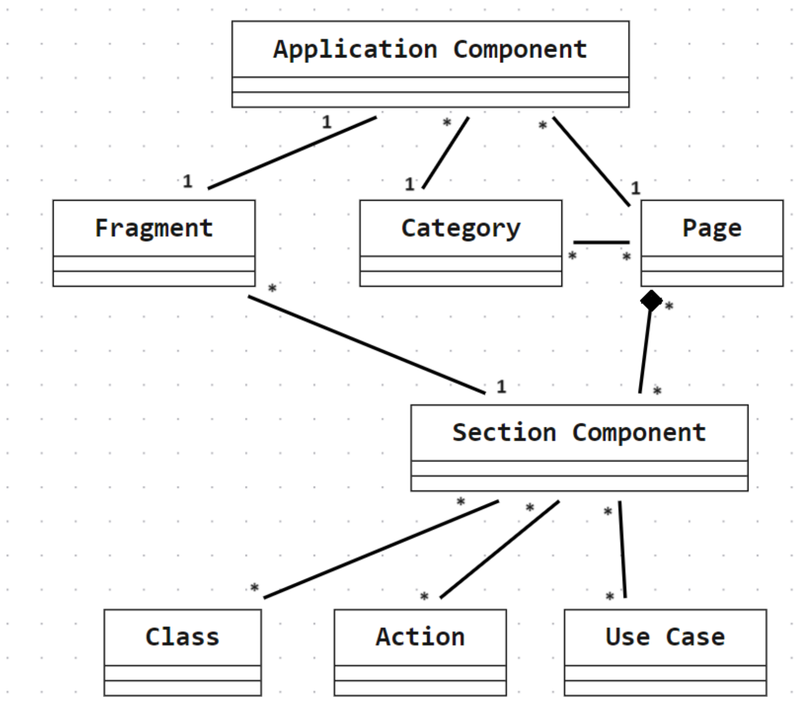


Figure 10: Final metadata model: The Application Component

4.2 New features in Interface

The already existing interface from the AI4MDE framework needed numerous adjustments. First, a new environment for the new metadata (The Application Component) needed to be implemented. Users could add and remove Application Components and give them a name and actors from the Use Case diagram to link it to. In addition, the possibility to edit an Application Component needed to be implemented so there could be Categories, Pages, and Section Components defined. All these components need various information for generation which the user can set up.

4.2.1 Fragment

A Fragment defines which data from the diagrams can be used in the Application Component. Therefore, this component is the first component the user gets to see. When a user adds nothing to the Fragment, all other components are greyed out. This was done to force users to start with the Fragment. A Fragment needs all elements (classes, actors, actions, etc.) from the diagrams and each element has to be checked or not checked. To keep the checkboxes clear, attributes are added as 'childs' under the matching class (parents). The same applies to extended use cases: they are added as 'childs' under the parent use case.

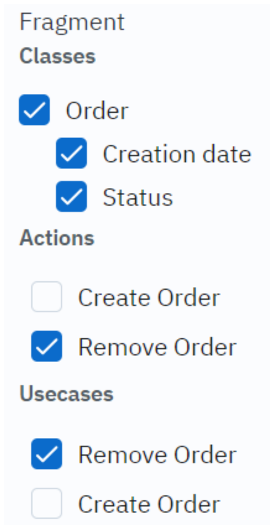
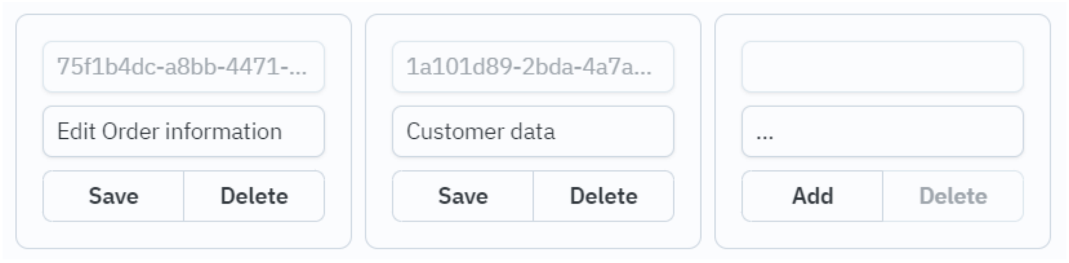

Fragment
Classes
✓ Order
  ✓ Creation date
  ✓ Status
Actions
☐ Create Order
✓ Remove Order
Usecases
✓ Remove Order
☐ Create Order

Figure 11: Editing a Fragment

4.2.2 Categories

The second component is Categories. Because this component defines the 'menu' for the application, it is important to define this before pages and sections. A Category is a simple component that has a name. Categories can be created or removed.


75f1b4dc-a8bb-4471-...
Edit Order information
Save | Delete

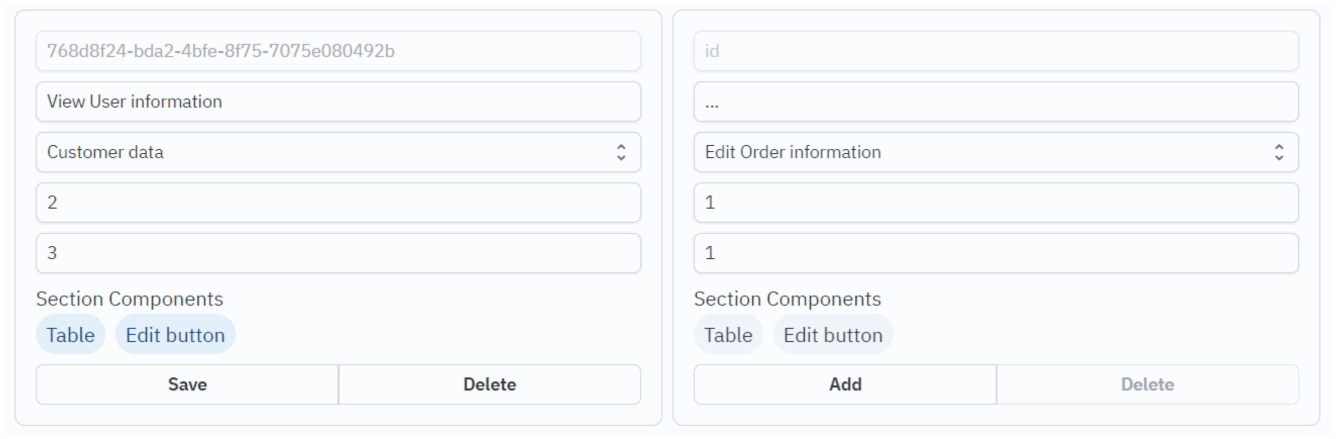
1a101d89-2bda-4a7a...
Customer data
Save | Delete

...
Add | Delete

Figure 12: Create or remove Categories

4.2.3 Pages

Pages have a name, linked Category, linked Section Components, and a row/column for the grid. There can be multiple Section Components on one page, so users must have the option to check multiple.


768d8f24-bda2-4bfe-8f75-7075e080492b
View User information
Customer data
2
3
Section Components
Table | Edit button
Save | Delete

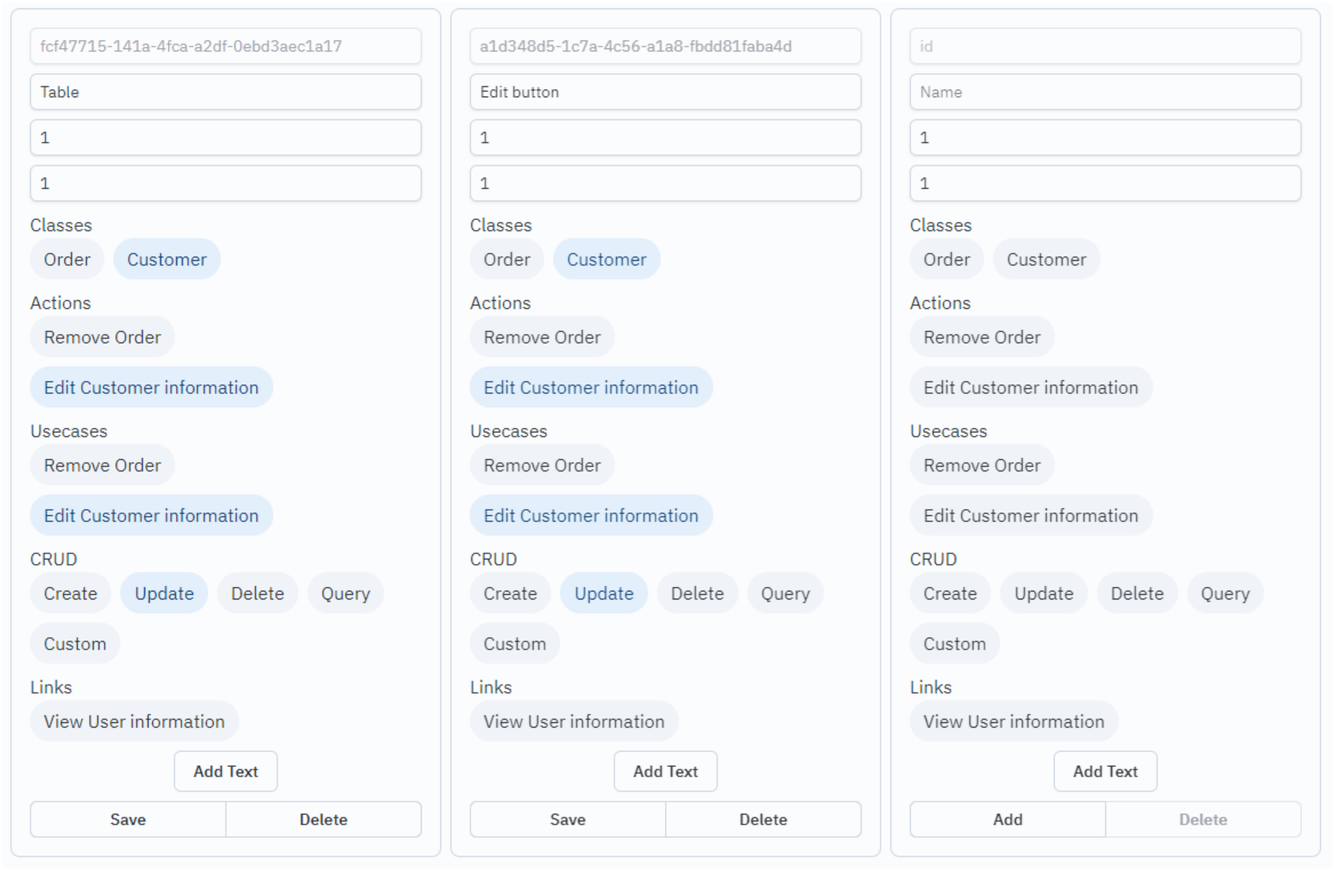
id
...
Edit Order information
1
1
Section Components
Table | Edit button
Add | Delete

Figure 13: Create or remove Pages

4.2.4 Section Components

As mentioned before, a Section Component is a very powerful component with abundant information for the generation process. Next to the information from the diagrams, the Section Component must exist of CRUD operations or a query. Users also have the option to choose from a custom method which can be added in the class diagram. A Section Component can be placed on a row and column on a page.

There are two more useful functions in the Section Component. First, users can add text. When they want to display a message, title, or description on a page, they can enter the text into an input field, add an ordering, and choose the font size. Secondly, users can add links to the Section Component. For example, when they want to make a button that links to another page, they can add a link to that page.



fcf47715-141a-4fca-a2df-0ebd3aec1a17
Table
1
1
Classes: Order [✓ Customer]
Actions: Remove Order [✓ Edit Customer information]
Usecases: Remove Order [✓ Edit Customer information]
CRUD: Create [✓ Update] Delete Query Custom
Links: View User information

a1d348d5-1c7a-4c56-a1a8-fbdd81faba4d
Edit button
1
1
Classes: Order [✓ Customer]
Actions: Remove Order [✓ Edit Customer information]
Usecases: Remove Order [✓ Edit Customer information]
CRUD: Create [✓ Update] Delete Query Custom
Links: View User information

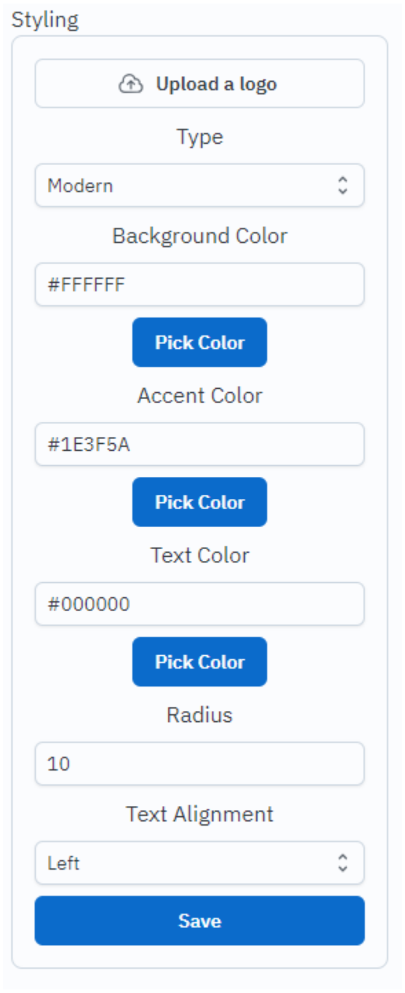
id
Name
1
1
Classes: Order Customer
Actions: Remove Order Edit Customer information
Usecases: Remove Order Edit Customer information
CRUD: Create Update Delete Query Custom
Links: View User information

Figure 14: Create or remove Section Components

4.2.5 Styling

Styling of pages is a part of Pages in the metadata model but an independent component in the Application Component. Users have the option to choose from some basic styles (basic, modern, abstract) which define most styles. Next to that, users have the option to style individual elements such as radius or accent color (for buttons).

The logo here is stored in a Base64 string which can be rendered to an image in HTML. With that way of working, there is no need to store a whole file [bas].



Type: Modern
Background Color: #FFFFFF
Accent Color: #1E3F5A
Text Color: #000000
Radius: 10
Text Alignment: Left

Figure 15: Edit style of pages

4.3 Smart Defaulting

To reduce the amount of actions a user needs to do in the interface, this project contains a smart defaulting technique. With information from the diagrams, a default Application Component can be built. As a base for this technique, the use case diagram can be used. From that diagram, classes and actions can be connected based on string matching. Then, all components such as Pages and Section Components can be defaulted. As default, one Application Component will be created for each actor from the use case diagram. Then, all use cases with information about their extends and their actors will be fetched. It is important to know which use cases are extended, which use cases are directly linked to an actor, and which use cases have extends, because the smart defaulting technique relies on an algorithm that uses this information. A thesis by L. Willemsens [Wil24] provides a deeper insight into the underlying thoughts and choices made for the smart defaulting technique. The pseudocode for the main algorithm to fetch all use cases with full information corresponds to the following:

```
function findAssociatedUseCases():
    associated_usecases = empty list
    for each usecase in all_usecases:
        if actor is the primary_actor of the usecase:
            add usecase to associated_usecases
    return associated_usecases

function determineRelationships(usecase, all_usecases):
    directly_linked = empty list
    extends = empty list
    extend = empty list
    for each other_usecase in all_usecases:
        if usecase is directly linked with other_usecase:
            add other_usecase to directly_linked
        else if usecase extends other_usecase:
            add other_usecase to extends
        else if usecase is extended by other_usecase:
            add other_usecase to extend
    return directly_linked, extends, extend

all_usecases = getAllUseCases()

result = []

for usecase in all_usecases:
    associated_usecases = findAssociatedUseCases(usecase.primary_actor, all_usecases)
    directly_linked, extends, extend = determineRelationships(usecase, all_usecases)
    result.append((usecase, associated_usecases, directly_linked, extends, extend))

return result
```

With this algorithm, use cases for the default Fragment can be fetched. The actions and classes can be linked based on string matching: for example, when there is a use case *Create Order*, it can be linked with the action *Create Order* and the class *Order*. All use cases, actions, and classes that are connected to the actor are placed in the default Fragment. It is important to first load the default Fragment, because default Categories, Pages, and Section Components use information from the Fragment.

Categories are defaulted based on classes from the Fragment. In most cases, for each class, there are actions to perform. These actions are on pages. For example for class *Order* there can be a page *Edit Order*, *View Order*, and *View Orders with negative Status*.

```
for each class in fragment:
    categories += class.name
```

Pages are defaulted based on use cases from the Fragment. Each use case that is not an extension and is directly linked to the actor is a page. Each use case which is an extension is on the page that is linked to the actor of which the use case is an extension.

```
for each usecase in fragment:
    if usecase.direct and !usecase.is_extension:
        page.name = usecase.name
        page.category = stringmatching(usecase.name, category)
        for section in sections:
            page.sections += stringmatching(usecase.name, section.name)
            for each usecase_extension in fragment:
                if usecase_extension.is_extension.to == usecase:
                    page.sections += stringmatching(
                    usecase_extension.name, section.name)
    pages += page
```

A Section Component is defaulted based on each use case from the Fragment. The linked use case, action, and class are connected to the Section Component.

```
for each usecase in fragment:
    section.name = usecase.name
    section.usecases = usecase
    for action in actions:
        section.actions = stringmatching(action.name, usecase.name)
    for class in classes:
        section.classes = stringmatching(class.name, usecase.name)
    section.crud = stringmatching(['Create', 'Read', 'Update', 'Delete'],
    usecase.name)
    sections += section
```

The interface is extended with a button 'reload Fragment' and a button 'reload defaults' to load this smart defaulting technique.

4.4 Communication with Backend

The editor application frontend from AI4MDE was developed using ReactJS: a JavaScript library known for flexibility and efficiency. The backend from AI4MDE was written in Python with a Django REST API. With ReactJS it is possible to create dynamic frontend applications which can update the rendered information at any time. The most used components like buttons, input fields, and selectors were sourced from Material UI because this library has a good performance and a wide community.

To enable communication between the frontend and backend, the Axios library is used. This library can make GET, POST, and PUT requests over HTTP and exhibits straightforward usability in ReactJS [PR20]. To request data from the backend, there are queries in the frontend that make a GET request. This is used for getting all classes from the diagram for example. To place something in the database, the frontend can mutate data which makes a PUT or POST request. Linking this to CRUD operations, a Create operation can match with a POST request, a Read operation can match with a GET request, an Update operation can match with a PUT request, and a Delete operation can match with a DELETE request.

4.5 Validation

In the generation process data from the output JSON is used. Therefore, it is required that the JSON contains valid data according to the right format used in the generation. To guarantee this, the interface contains validation in the different components.

The interface makes use of Formik, a third-party library to create and handle forms in ReactJS. All information entered by the user, such as input fields or checkboxes are handled by a Formik form. To handle validation in these forms, Yup can be used. Yup is a library that can handle form validation and schemes. For example, Yup can validate whether an input is a number, string, or date. Furthermore, a string can have a minimum or maximum length [SOM21] [Yup] [For].

By using Formik and Yup across all components—be it Category, Page, or Section Component—the interface assures a high degree of accuracy in the generated output. Furthermore, Yup serves as a potent defense against SQL injection attacks. By mandating that input strings adhere strictly to alphanumeric characters, Yup effectively reduces the risk of SQL injection vulnerabilities. This is important to ensure a high level of security in the generation process.

5 Generation

In the generation process, Django is used in combination with Jinja2. The main idea is that the data required for generation is combined in one JSON file. This JSON file facilitates the creation of Python objects used for templates. This thesis focuses on the user interface specification of the generation process. This section gives an overview of:

- How template-driven generation works with Django
- An intermediary layer in metadata parsing (interface to generation)
- What will be generated from the Application Component metadata
- A deeper insight into the generation of HTML pages with sections

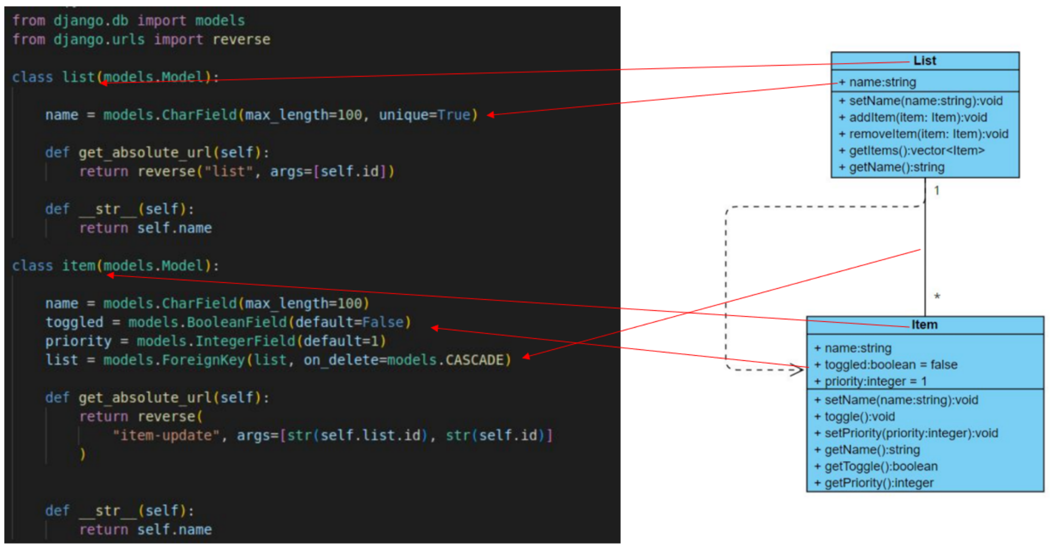
5.1 Template driven generation

When a user sets up Django, it creates some base files for the application:

```
application/
├── manage.py
├── project_name/
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── application_name/
    ├── migrations/
    │   └── __init__.py
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── models.py
    ├── tests.py
    ├── urls.py
    └── views.py
```

Normally, a user who will use Django has to set up these files by hand: for example, urls.py has to contain the whole path through the web application. Users can enter all kinds of URLs there and connect them to views.

The metadata can be used to generate these files in advance. For example, a class diagram can be mapped to the models.py (which contains the database structure). This mapping can be seen in figure 16



```
from django.db import models
from django.urls import reverse

class list(models.Model):

    name = models.CharField(max_length=100, unique=True)

    def get_absolute_url(self):
        return reverse("list", args=[self.id])

    def __str__(self):
        return self.name

class item(models.Model):

    name = models.CharField(max_length=100)
    toggled = models.BooleanField(default=False)
    priority = models.IntegerField(default=1)
    list = models.ForeignKey(list, on_delete=models.CASCADE)

    def get_absolute_url(self):
        return reverse(
            "item-update", args=[str(self.list.id), str(self.id)]
        )


    def __str__(self):
        return self.name
```

List
+ name:string
+ setName(name:string):void
+ addItem(item: Item):void
+ removeItem(item: Item):void
+ getItems():vector<Item>
+ getName():string
1
*
Item
+ name:string
+ toggled:boolean = false
+ priority:integer = 1
+ setName(name:string):void
+ toggle():void
+ setPriority(priority:integer):void
+ getName():string
+ getToggle():boolean
+ getPriority():integer

Figure 16: Mapping the class model to models.py (Django)

Each base file from Django can be manipulated with Jinja2 to already create these files. In Jinja2, the code would look like this:

```
{% for model in models %}
class {{ model.data.name }}(models.Model):
{% for attribute in model.data.attributes -%}
    {%- if attribute.type == "bool" -%}
    {{ attribute.name }} = models.BooleanField(default=False)
    {% endif -%}
{% endfor -%}
{% endfor -%}
```

This makes it also possible to generate custom stylesheets and HTML pages. With data from Python objects, retrieved from the user, variables can be passed to a Jinja2 CSS or HTML file.

Upon the completion of configuring all of the user's preferences, data from all components are initialized: the class diagram, the activity diagram, the use case diagram, and the application component. All these components are stored in a large JSON file and can be mapped to Django files. This mapping can be seen in figure 17.

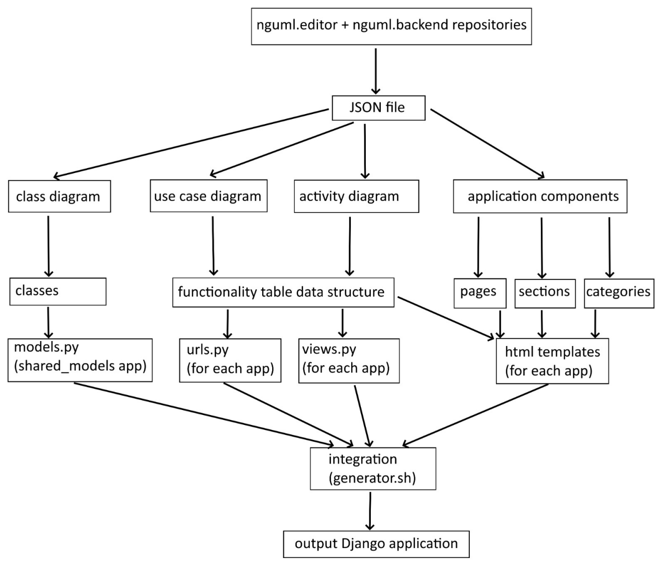

nguml.editor + nguml.backend repositories
JSON file
class diagram
use case diagram
activity diagram
application components
classes
functionality table data structure
pages
sections
categories
models.py (shared_models app)
urls.py (for each app)
views.py (for each app)
html templates (for each app)
integration (generator.sh)
output Django application

Figure 17: Mapping of components to Django files

5.2 Metadata parsing

To parse all metadata into Python objects, an intermediary layer was implemented (functionality table data structure in Figure 17). This layer is used for validation and mappings between elements and components. A thesis by L. Willemsens [Wil24] provides a deeper insight into this intermediary layer and how it can be generated from the diagrams and Application Components.

5.3 Application component generation

All data from the application component can be retrieved from the JSON file. An overview of this metadata is given in section 4.

First, there is a need for the creation of stylesheets. These stylesheets can vary per application (which is linked to actors) in a prototype based on its type (Basic, Modern, Abstract). From the JSON, a dictionary is created for the whole styling part, which looks like this:

```
style = {
    "type":"modern",
    "background-color":"#FFFFFF",
    "text-color":"#000000",
    "accent-color":"#EF9AC",
    "radius":"10"
    "text-alignment":"left"
}
```

The combination of keys and variables can be passed in a Jinja2 file, allowing it to create custom stylesheets.

Secondly, to easily access pages through the application, there have to be Categories stored. These Categories consist of a unique id and a name. Each application has its own Categories. Categories are stored in a list:

```
categories = [
    {
        "id":"24b9ab06-9a19-4872-b1f5-23b4c4ab3cfa"
        "name":"Forms"
    }
    {
        "id":"7e82ada1-05b9-454b-ac6e-d125deb0cad4"
        "name":"Questions"
    }
]
```

This data can be passed into Django's base.html file which is the base for each page. Section 5.4 goes into more detail on this. In the base.html file, each Category from the list is rendered through the Jinja2 base.html file to be placed in the menu so users can navigate through the pages.

The most complex part is the Section Component. To fully understand the structure of the metadata for Section Components, section 4 gives a structured overview.

All metadata from the diagram and all metadata from the application component are combined into a large JSON file:

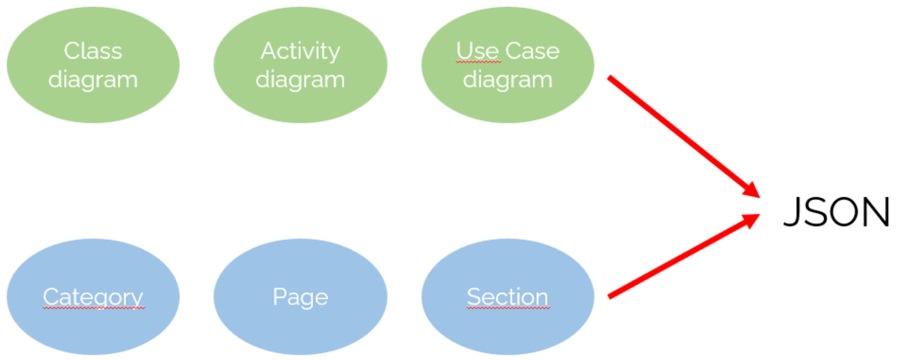


Figure 18: 'Diagram world' and 'Application Component world' combined into a large JSON file

5.4 HTML page generation

Django uses templates for webpage generation. Normally there exists a base.html file which is the base for each webpage. To generate different content on different pages, there are HTML files that exclude the base.html file. Jinja2 provides support for this approach with the following architecture:

```
<body>
    {% block content %}{% endblock content %}
</body>
```

Inside the body of a file, there is 'block content' generated. This block content is the content from the other files, which will render the right content into the base.html file. The other HTML files' architecture will look something like this:

```
{% extends "base.html" %}

{% block content %}
    <h1>Hello World</h1>
{% endblock content %}
```

In the end, the total mapping from the diagrams and Application Component to Django files and templates looks like this:

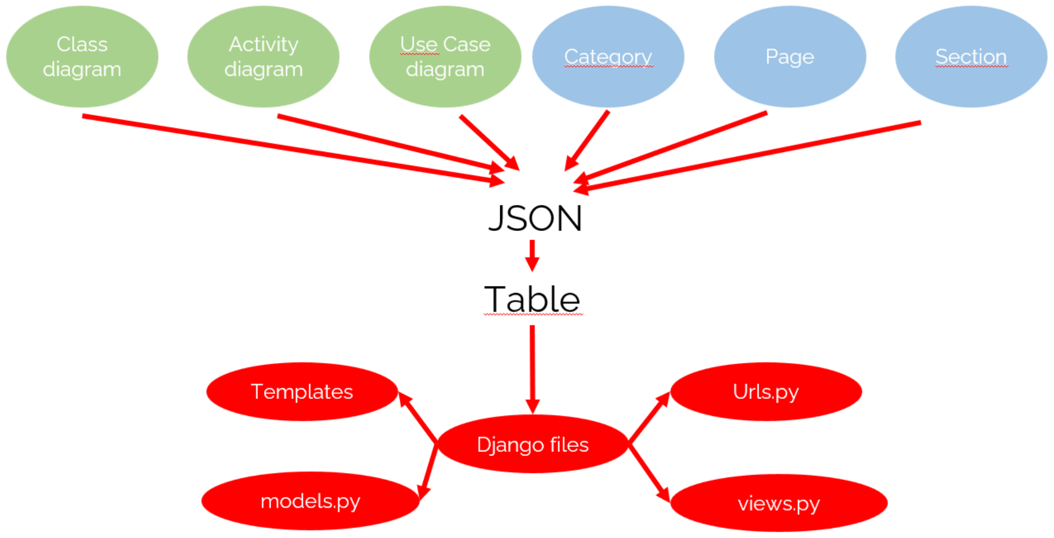


Figure 19: Total generation mapping

5.4.1 Style

For an attractive web application, some basic styles are implemented. Values like background color and radius are set by the user in the application component and are passed into a CSS Jinja2 file. This file is linked to the base.html file so each webpage has the same format of style.

5.4.2 Grid

To solve the problem with the old WYSIWYG editor from B. van Aggelen [vA22], each Section from the Application Component will be placed on a grid (see an overview of how this works in section 4). With this grid, the same functionality as the WYSIWYG editor is kept but in a much more user-friendly way.

For each CRUD element, there are one or more HTML items placed on the page. For example, a *Delete* element is a button to delete something. These HTML items together form a section on the page and this section is placed on a row and column. For example, in figure 20 there is a page with 3 columns and 4 rows.

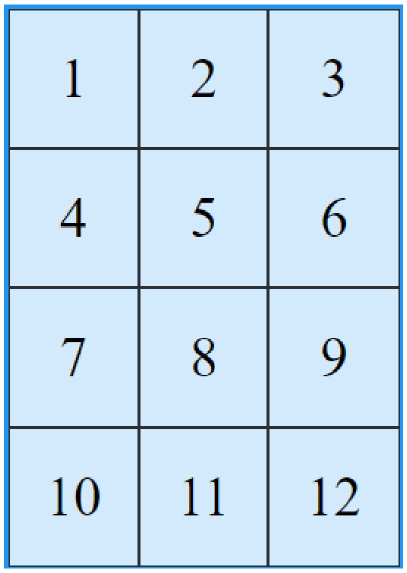


Figure 20: Example grid

5.4.3 Output

To execute the generated prototype, the location of the JSON file from the interface could be called in a shell script. This shell script has two arguments: the JSON file and a name for the project:

```
./generator.sh runtime.json test_project
```

This script calls all generator functions such as HTML page generation scripts and immediately runs the generated prototype.

The following figure is an example output:

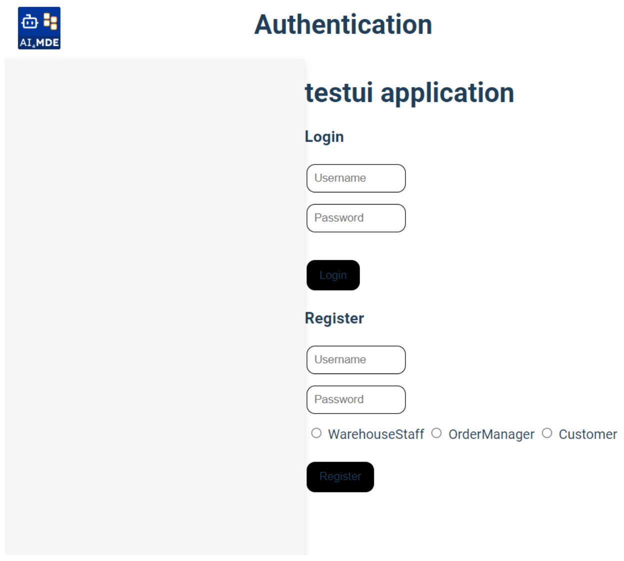


Figure 21: Example output

5.4.4 Reusable Application Components

The generation process only supports CRUD operations. To generate complex advanced functionality like authentication or registration, a component library was implemented. This library consists currently of one component: authentication. The component is added as a non-editable Application Component in the interface. The generation process recognizes this component with a hardcoded id and generates default code for this component. This solution provides support for common complex non-CRUD operations.

6 User Testing

To test the solution provided, test users were invited to work with the extended AI4MDE framework. These users were selected by their ability to provide relevant feedback: for example, business owners or software architects are people who will work with the framework when it is open-source in the end.

6.1 Manual & Questions

The test users were handed a manual that consisted of two parts. First, they could take a look at the extended AI4MDE application: they could inspect diagrams and application components (set up on forehand) but not edit them. Then, the test users could change or add elements in the diagrams and application components to show the impact on the generated prototype. After each part, they were handed a few questions on which they could answer from 1 to 5. In addition, they could make suggestions about improvements to the platform. The questions, answers, and manual can be found in Appendix A and B.

6.2 Analysis

The results of the testing were clear. The platform has proven to be useful to our test users, providing them with the tools and resources needed to generate a prototype. However, there were still some bugs that made it hard to use for some functionality. Furthermore, it was not always clear what actions the test user was actually performing. For instance, when the user made a change in the diagram, he/she did not have a precise understanding of the implications it would have on the final prototype. The main point of improvement was calling the shell script: the test users found this hard to execute and follow, so a button would have been better. An extensive qualitative analysis was done by L. Willemsens [Wil24].

6.3 Population Analysis

In conducting a population analysis for this study, careful consideration was given to the selection of test users for evaluating the prototype application generator. The chosen test users were selected based on their relevance to provide valuable insights into various aspects of the prototype. The diverse pool of test users encompassed individuals with distinct backgrounds, including a business owner, IT specialists, WordPress developer, a junior database designer, a full-stack developer, and a visual basic expert. The total number of test users involved in the evaluation process was seven, reflecting a focused and targeted approach to user testing. While the sample size may appear relatively small, this analysis emphasizes qualitative insights rather than quantitative metrics.

7 Conclusion & Discussion

In conclusion, this thesis explored the design-time and run-time shortcomings in prototype application generation and tried to enhance these shortcomings in the AI4MDE platform. First, the already existing metadata model in the AI4MDE platform was extended. Categories, Pages, and Section Components have a new purpose and more attributes. Then, a 'Fragment' was added to specify which parts of the diagram the different applications could use. All components together formed the new Application Component.

By extending the metadata model, this thesis facilitated an interface for the generation of multiple CRUD (Create, Read, Update, Delete) operations on custom HTML pages, thereby affording users greater flexibility and control over their application designs. Furthermore, the incorporation of multiple Application Components empowered users to create multiple applications within a single project. In combination with the extended generation process amplifies this solution the platform's versatility and utility.

The solution was tested by a diverse selection of test users, ranging from a business owner to IT specialists. These test users got acquainted with the extended AI4MDE platform and provided feedback about the interface and generation process. Most test users indicated they would use the platform in the end but there is additional work needed to provide an effective, bug-free solution for prototype generation. Furthermore, documentation and descriptions are essential for the platform in the end.

The extended AI4MDE platform is effective for basic prototype application generation. UML diagrams specify all functionality needed in the application and with additional information from the end user, it is possible to generate a multi-application prototype with custom HTML pages. This generation process is fast and efficient. However, for end users the platform requires improved documentation and guidance. Furthermore, the platform needs some improvements in functionality. For example, only CRUD operations are fully supported. Additionally, it lacks the capability for queries, limiting its utility in scenarios requiring more complex data retrieval and manipulation functionalities. In conclusion, while the extended platform demonstrates effectiveness in prototype application generation, it necessitates significant additional work and refinement before it can be confidently presented to end users, particularly in terms of improved documentation, guidance, and additional functionality such as query capabilities.

This study has demonstrated that a well-functioning interface is crucial for prototype generation. The selected user input plays a pivotal role in the final functionality of the prototype, with the actual functionality within the components section being of utmost importance. Additionally, it is imperative that such an interface is highly user-friendly and accompanied by comprehensive documentation.

8 Future Work

For this thesis, a few features were out of scope. First, the idea was to implement two versions of functional components. One abstract component was implemented: authentication. In future work that component library can be extended with more functional components. The second version of functional components are advanced Section Components like a calendar or diagram based on data. It would be a great feature if there could be advanced Section Components added to the Interface.

Another beneficial feature would be a real-time wireframe for the user. This means when a user for example sets all styles for pages, they can immediately see a small wireframe that displays all choices the user made.

Queries can be improved. Now, the user has to type in a string of a query but it would be nice to make this more dynamic. For example, the user only has to select options from dropdown menus with classes and attributes in which the query operates. Also, the query variables can be added to a dropdown (if, else, greater than, and, or). The abstract idea behind this can be found in a thesis by S. van Dam [vD24].

References

[bas] Base64. https://developer.mozilla.org/en-US/docs/Glossary/Base64. Accessed: 2024-04-15.

[cru] Crud operations explained: Create, read, update, and delete. https://www.educative.io/blog/crud-operations. Accessed: 2024-04-03.

[Dja] Django - the web framework for perfectionists with deadlines. https://www.djangoproject.com/. Accessed: 2024-02-19.

[Dri20] R. Driessen. *UML class models as first-class citizen: Metadata at design-time and run-time*. Bachelor thesis, Leiden University, 2020. Available at https://theses.liacs.nl/pdf/2019-2020-DriessenR.pdf.

[For] Formik - get started. https://formik.org/docs/overview. Accessed: 2024-04-03.

[Jin] Jinja. https://palletsprojects.com/p/jinja/. Accessed: 2024-02-19.

[KEBP21] Hatice Koc, Ali Erdoğan, Yousef Barjakly, and Serhat Peker. Uml diagrams in software engineering research: A systematic literature review. *Proceedings Volume 74*, 2021.

[PEH23] Benjamin Prautsch, Uwe Eichler, and Uwe Hatnik. Generating the generator: A user-driven and template-based approach towards analog layout automation. *Electronics*, 12(4), 2023.

[PR20] Archana N. Mahajan Prateek Rawat. Reactjs: A modern web development framework. *International Journal of Innovative Science and Research Technology Volume 5, Issue 11, November – 2020*, 2020.

[Pra20] Cahyadi A. Pratama, M. A. Effect of user interface and user experience on application sales. *IOP Conference Series: Materials Science and Engineering*, 2020.

[Ram21] Griffioen P. P. Schouten M. B. J. Chaudron M. R. V. Ramackers, G. J. From prose to prototype: Synthesising executable uml models from natural language. *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021.

[SJM03] Takao Futagami Stephen J. Mellor, Anthony N. Clark. Model-driven development. *IEEE SOFTWARE*, 2003.

[SO08] Saeko Matsuura Shinpei Ogata. Automatic generation of uml-based web application prototypes. *Proceedings of the Tenth International Conference on Enterprise Information Systems - Volume 3*, 2008.

[SOM21] Marzieh SOMI. *User Interface Development of a Modern Web Application*. Master thesis, POLITECNICO DI TORINO, 2021. Available at https://webthesis.biblio.polito.it/secure/30076/1/tesi.pdf.

[vA22] B. van Aggelen. *Enabling data-driven wireframe prototyping using Model Driven Development*. Bachelor thesis, Leiden University, 2022. Available at https://theses.liacs.nl/pdf/2021-2022-AggelenBvan.pdf.

[vD24] Sven van Dam. *A flexible, template-driven generation framework for Model Driven Engineering*. Bachelor thesis, Universiteit Leiden, 2024.

[Wil24] Lucas Willemsens. *Utilising Use Case Models for multi-actor Prototype Generation*. Bachelor thesis, Universiteit Leiden, 2024.

[Yup] Npm - yup package. https://www.npmjs.com/package/yup. Accessed: 2024-04-03.

A Appendix: User Testing Manual

1. Generating a Webshop prototype using the already present Diagrams and User Interface

   1.1 Go to the design studio: http://localhost:5173/

   1.2 On the home page, click on "Get started"

   1.3 Click on the already present "Webshop" project to open the project that is used for testing

   1.4 Select the already present "Testing System" at the systems page of this project

   1.5 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

   1.6 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.

   1.7 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.

   1.8 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.

   1.9 After this generation is done, your prototype can be inspected. Go to: http://127.0.0.1:8001/ to inspect the newly generated Webshop.

   1.10 Now please answer the questions on the first page of the questions sheet.

2. Now, we will edit the modeling diagrams and the user interface and inspect the changes that will occur in the generated prototype

   2.1 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.

   2.2 The Class modeler wil open. You can edit classes by clicking the right mouse button on a class (a rectangular box) and then clicking "Edit".

   2.3 A popup window will open. Try adding a new attribute by pressing "Add Attribute".

   2.4 A new attribute will appear. You can give this attribute a name (no spaces!!!) and a type: String: Text, Integer: Number, Boolean: Yes/No

   2.5 When you are done press "Save"

   2.6 We will now add a new class. Right click on the empty canvas and click "New Node"

   2.7 A popup window will open. Select type "Class" and give the class a name. When you are done press "Add"

   2.8 Give the new class new attributes like you have done in steps 3-5

   2.9 We will now connect the new class to an already existing class. Right click the new class and press "Connect"

2.10 A new line will follow your cursor. Left click a class you want to connect the new class to.

2.11 A new popup window will open. Select type “Association” and press “Add”

2.12 The new class is now connected to the model. We will now leave the Class Diagram editor and start editing the Use Case Diagram

2.13 On the System page, left lick the “Use Case Diagram” tile to start editing the Use Case diagram.

2.14 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”

2.15 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”

2.16 We will add the new Use Case to an existing actor. Right click the Use Case and click on “Connect”

2.17 Connect the Use Case to an actor and select type “Interaction” in the new popup. Click “Add” when you are done.

2.18 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”

2.19 In the popup select “Actor” and give the new Actor a name. When you are done click “Add”

2.20 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 17

2.21 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)

2.22 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”

2.23 The models are done. We will now start editing the user interfaces by clicking the “Interface” tab in the system menu

2.24 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.

2.25 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.

2.26 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

2.27 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”.

2.28 If wanted, you can edit the styling for this user interface under the “Styling” tab.

2.29 When you are satisfied with everything, click the “Prototype” tab to download the new JSON file. Repeat steps 8-9 to generate the new prototype.

2.30 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

2.31 Please fill in the questions on the second page of your question sheet. Thank you for your service!

B Appendix: User Testing Questions and Answers

PART I
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- PROTOTYPE) 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?
Answers: - 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 gebeurd 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 scructured files te gebruiken om je webshop DB als manager eenvoudig te kunnen (bij)vullen?

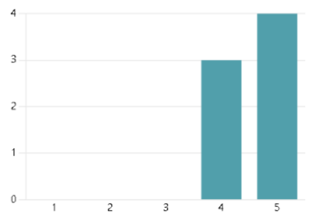


Figure 22: Ease of Use

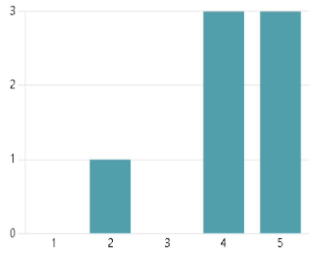


Figure 23: Clarity

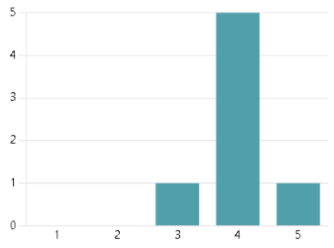


Figure 24: Effectiveness

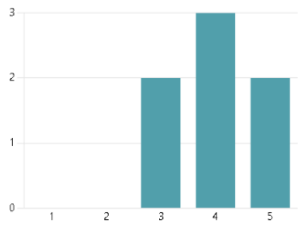


Figure 25: Accessibility

PART II
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?
Answers: - 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.

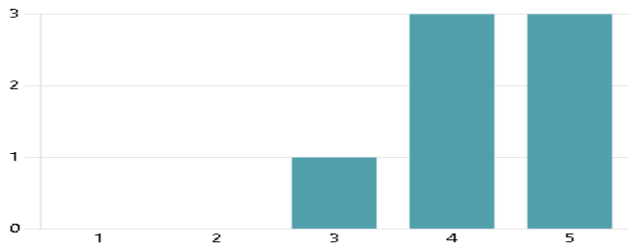


Figure 26: Effectiveness Class Diagram

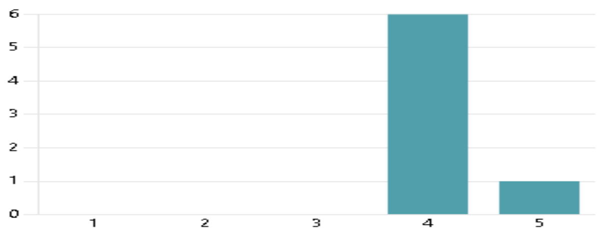


Figure 27: Effectiveness Use Case Diagram

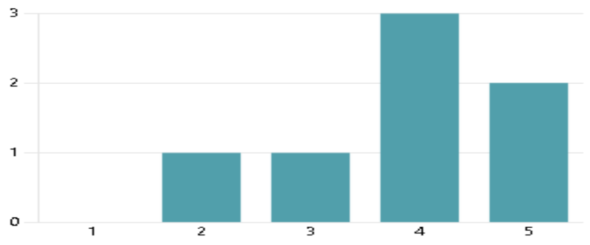


Figure 28: Changes Reflection Class Diagram

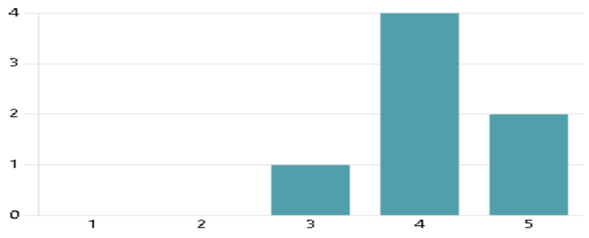


Figure 29: Changes Reflection Use Case Diagram

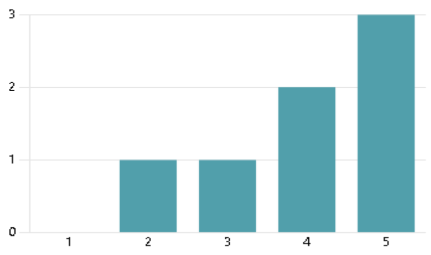


Figure 30: Effectiveness Fragment

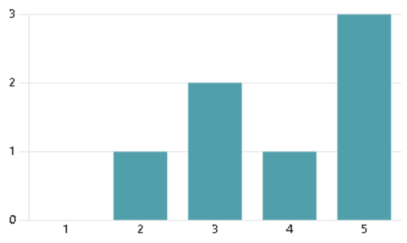


Figure 31: Effectiveness Categories

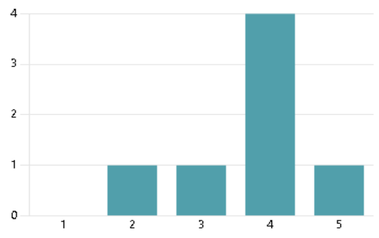


Figure 32: Effectiveness Pages

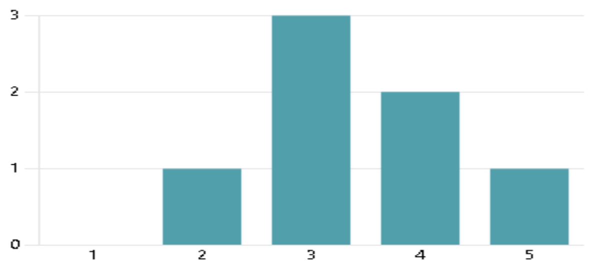


Figure 33: Effectiveness Section Components

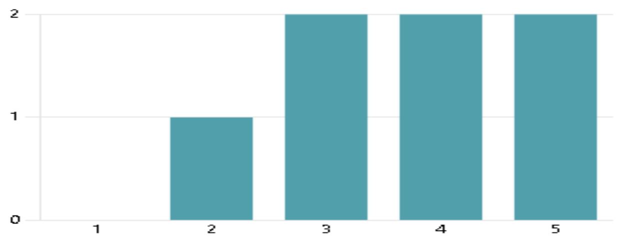


Figure 34: Effectiveness Styling

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?
Answers: - 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 kan 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 uploaden
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?

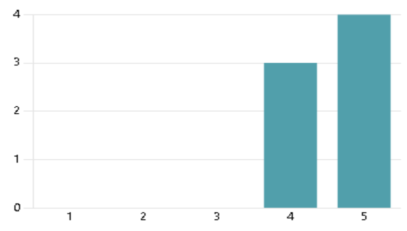


Figure 35: Queries Support

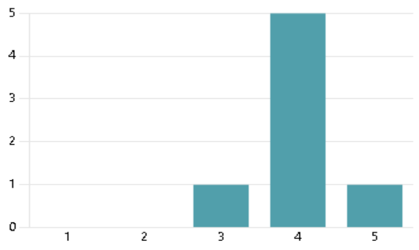


Figure 36: Activity Diagram Support

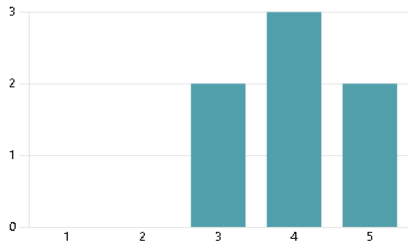


Figure 37: Effectiveness Sidebar

Answers: - Ja, je begint al met de basis opmaak en vanuit daar ga je verder bouwen. Je hoeft dus niet helemaal vanaf het begin to 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 ongeoefende gebruikers.

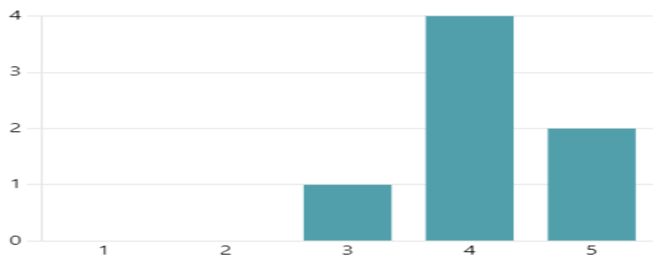


Figure 38: Quality Default Values

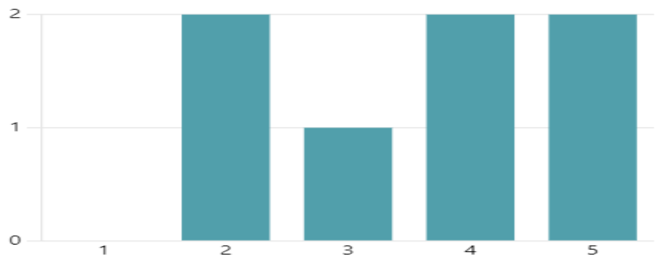


Figure 39: Quality Prototype