Opleiding Informatica

Utilising Use Case Models

For Multi-actor Prototype Generation

Lucas Willemsens

Supervisors:
Guus Ramackers & Joost Visser

BACHELOR THESIS

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

04/10/2023

Acknowledgements

How this thesis has come to a successful close is in great part thanks to the support of my supervisors Dr. G. J. Ramackers and Prof.dr.ir. J.M.W. It was Joost Visser who initially referred me to Guus Ramackers when I contacted him in the summer of last year. Both of their sharp academic feedback during the project is greatly appreciated. The insights of Guus Ramackers in project management, UML and software development, especially, have been a great inspiration for me.

Our meetings with Guus Ramackers and Max Boone, whom I would also like to thank for sharing his vast knowledge as Technical Architect on the platform, and the other Bachelor students Pieter de Hoogd and Sven van Dam were great learning opportunities and motivated me to work harder and remain focused on the most important tasks at hand.

I would also like to thank Bram van Aggelen for his previous work on the platform and the crucial tips he shared with us on displaying prototype software.

Lastly, I thank my love Annabel for her patience and support during this important conclusion to my studies.

Abstract

This study explores the application of use case diagrams to Model Driven Engineering. A method of template-based generation is proposed and implemented as an extension to the AI4MDE platform. The implementation is validated with a survey of participants following a set of instructions where they edit UML diagrams and examine the resulting prototypes.

By utilising metadata of use case diagrams and class diagrams, smart default values are calculated. Classes correspond to models in the prototype database using Object Relational Mapping. Similarly, for each actor in the use case diagram, a corresponding app will be generated in the prototype, with each use case that the actor interacts with represented as a unique page on its app. Operations on the database are made available through the page depending on the name of the use case. Use cases with *extends* relations result in compound pages, fusing sections with operations on identical models.

Test user responses indicate the additional support for use case diagrams offer an increase in ease-of-use, clarity and accessibility to Model Driven Engineering. Users are aided in generating prototype software that supports creating, reading, updating, deleting and calling custom methods on models represented in UML diagrams.

Contents

Acknowledgements 1

Abstract 2

1 Introduction 4
- 1.1 Research Questions . . . 5

2 Background 6
- 2.1 Definitions . . . 6
- 2.2 AI4MDE . . . 7

3 Related Work 8
- 3.1 Unique contribution . . . 10

4 Research Methodology 10

5 Conceptual Architecture 11
- 5.1 Generator Design . . . 12
- 5.2 Smart defaults . . . 12
- 5.3 Platform metadata model . . . 13

6 Technical Architecture 14
- 6.1 Code Implementation . . . 14
- 6.2 Custom Methods . . . 15
- 6.3 Prototype Database . . . 16

7 Case Studies 17
- Diagrams Webshop . . . 18
- 7.1 Results . . . 20

8 Validation 21
- 8.1 Test Summary . . . 21
- 8.2 Results . . . 21
- 8.3 Qualitative analysis . . . 22
- 8.4 Reflection . . . 23

9 Conclusion 24

10 Discussion 25
- 10.1 Extending the AI4MDE platform . . . 25
- 10.2 Future work . . . 26

References 28

Appendix 29

1 Introduction

Software systems can be difficult to understand. Today, technology advancements from the past 50 years have culminated in endless options being available at the time of their design. Efficiency and power is lost in abstraction because complexities of incomprehensibly tiny parts composing computer hardware systems pre-suppose extensive inquiry. In software, frameworks built upon frameworks result in similar confusion. The supposed age of Artificial Intelligence (AI) does not bring immediate solutions either. In practise, AI is often seen as a black box. No one knows its inner workings.

Decisions on how specific computer systems should work are kept from people without an extensive knowledge of computers in general. Software should be designed in a way that utilises the wasted potential of powerful technological options of recent times. Model Driven Engineering (MDE) offers a solution to these issues by enabling the design of complex systems through comprehensible models.

This bachelor thesis investigates the design and implementation of a prototype generator utilizing metadata of diagrams in the Unified Modeling Language (UML). This research is conducted within the broader Prose-to-Prototype (P2P) project at the Leiden Institute of Advanced Computer Science (LIACS). Designs and code were developed for the AI4MDE platform prior to this research, when the platform was known as Next Generation UML (ngUML). This research is a collaborative effort involving two other bachelor theses extending the platform in parallel.

The focus of this research is on use case diagrams and how their metadata can be utilized to develop functional web application prototypes for systems described on the AI4MDE platform. Class diagrams are also examined, for which preliminary support existed on the platform. Both diagram types play a crucial role in the proposed MDE methodology. Finally, their combination presents a unique research challenge, which is addressed through the introduction of a novel data structure. The generation process from a platform user perspective is visualized in Figure 1.

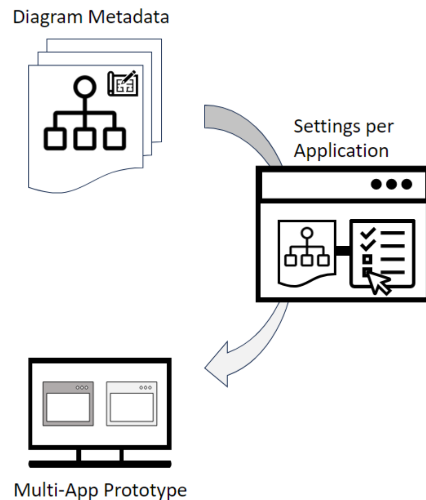

Figure 1: AI4MDE enables users to create prototypes using UML diagrams. The workflow includes selecting relevant parts of diagrams for output apps

1.1 Research Questions

This research explores the usage of use case diagram metadata during prototype software generation, combined with class diagram metadata and supplementary user settings. For this, the following central questions are addressed:

How can a UML diagram-based prototype generator be designed?
How can this design be implemented on the LIACS AI4MDE project?

Sub-questions

For a comprehensive examination of our central research questions, the following sub-questions are formulated:

- How do specific aspects of use case diagrams translate to prototype requirements?

  Generated prototypes should adhere to the intention of use case diagram designers in addition to the already supported class diagram. For this reason, use case diagram usage is researched. Relevant features of use case diagrams in MDE context should be prioritized in order the answer our central research question and potentially simplifying implementation.

- How is the combination of other model specifications, such as requirements specified in class diagrams, effectively incorporated in the generation process?

  It is crucial to harness the collective information provided by the diagrams by extracting the most pertinent data from each type in order to answer our research questions. This approach streamlines implementation and ensures that the generator can be used intuitively.

- How can a generic prototype generator fulfill requirements dictated by UML models?

  Ensuring that the final implementation remains faithful to the system described by the model is crucial. However, achieving this alignment can be challenging due to inherent limitations. It is imperative to identify and address these limitations comprehensively to optimize the functionality of the generator. By carefully determining the limitations and exploring alternative solutions, the design can adapt to effectively address the research questions while maintaining alignment with the model.

Research Overview

The introductory chapter provides background information regarding the research questions. In the subsequent background chapter definitions are provided and the AI4MDE project is explained. A thorough review of related work offers insights into existing literature and the relevance of this research. Research Methodologies are then elucidated in detail. The conceptual and technical architectures are expounded upon in separate chapters, shedding light on the design and implementation strategies employed. Case studies and validation results are presented which test and demonstrate the proposed solutions. The thesis concludes with a discussion on extending the AI4MDE platform and outlines avenues for future research.

2 Background

2.1 Definitions

- Model Driven Engineering (MDE): A software development approach that focuses on creating and using models to design and implement software systems.
- Model-View-Controller (MVC): A design pattern which separates user interface into the three interconnected parts in its name. Only the view part is open to a user and presents it with input options. These inputs are handled by the controller, which communicates with the static model part representing a database.
- Software Development Life Cycle (SDLC): A term used mostly in the context of entrepreneurship referring to the whole process of designing, creating and maintaining software.
- Unified Modeling Language (UML): A standard for diagrams of different types, among which use case Diagrams, Activity Diagrams and Class diagrams. It was adopted and managed by Object Management Group (OMG) in 1997 and since then has been adopted by many software developers and scientists.
- Leiden Institute of Advanced Computer Science (LIACS): The faculty of AI and Computer science of Leiden University originally stemming from the Maths department.
- Artificial Intelligence for Model Driven Engineering (AI4MDE) platform: A project directed by Guus Ramackers with the goal of improving and actualizing the potential of UML in software development for both software engineers and entrepreneurs.
- Next Generation UML (ngUML) platform: The former name of the AI4MDE platform.
- Prose-to-Prototype (P2P): Referring to the AI4MDE platform with a highlight to the functionalities about the input of natural language and the output of a working prototype.
- Natural Language Processor (NLP): A computer model that can extract meaning from human language. Usually a model trained on a large database of text found on the internet or in books using machine learning. An example is ChatGPT.
- Object-Relational-Mapping (ORM): A database management technique whereby models are mapped to tables in a database.
- Django: A Python Framework with features that deal with issues relating to the creation of prototype apps. For example, Django implements ORM using Python objects representing database entries [d*Django*].
- SQLite: A public domain database system written in the C programming language.
- Migrations: Migrations in Django are Python files generated by the Django framework to manage database schema changes. These migrations encapsulate the modifications made to the models defined in Django applications, such as creating, altering, or deleting database tables and fields.

- CRUD: Operations on database entries which are often used: Create, Read, Update and Delete.

2.2 AI4MDE

AI4MDE is a software research project at LIACS that aims to automate software development by generating UML diagrams from natural language descriptions. By leveraging metadata from these UML diagrams, web application prototypes are then created automatically, assisting developers and entrepreneurs from the start of the SDLC.

Previous research on-platform

Prior to this thesis, the NLP component was successfully implemented on AI4MDE. UML use case, activity and class diagrams could be generated from a textual input. These diagrams could be altered using a web editor within the environment. Diagram metadata was stored in JSON format. A prototype app could be generated on-platform using class diagram metadata and user input, as demonstrated by Van Aggelen et al. (2022) [Agg22], Figure 2. Significant inspiration was drawn from the generator design. However, excessive user input required by the What You See Is What You Get (WYSIWYG) editor and a lack of maintainability limited this approach.

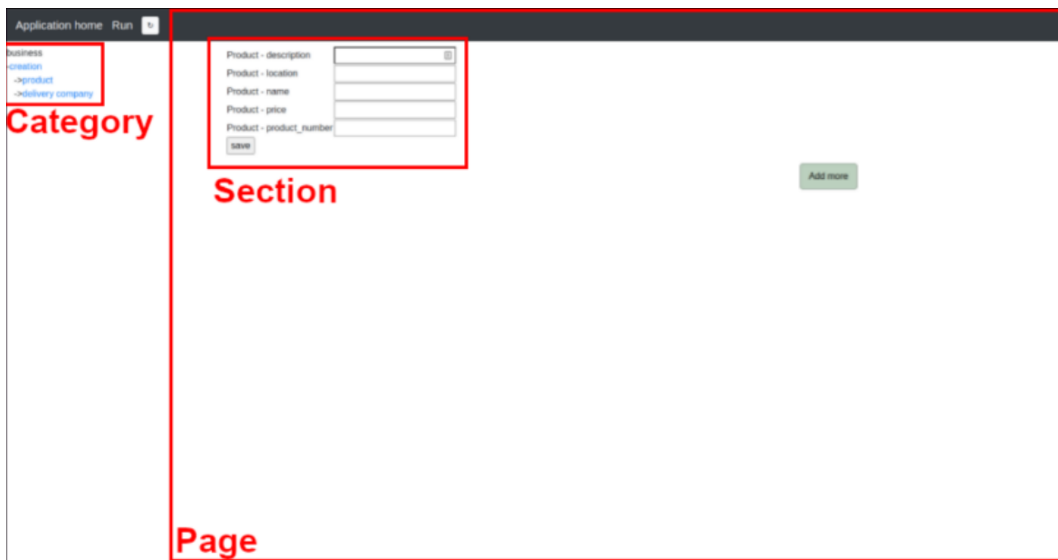

Figure 2: A prototype app displaying categories, sections and pages could be generated using the WYSIWYG editor available on AI4MDE prior to this research [Agg22]

Aims to improve on previous work include additional features such as generating software by utilizing more diagrams without relying on the WYSIWYG editor and having more user-friendly tools available on the platform. Methods were developed to integrate both use case and class diagrams into a single functional multi-application prototype by consolidating them within specified data structures. These include *page, section* and *category* structures, as displayed in Figure 2.

Van Aggelen et al. (2022) also advocate for wireframes as a rapid prototyping tool in software development. Test user response indicates a clear demand for pre-views of pages based on current

settings during editing. Wireframes were a suitable option. However, the currently implemented Django apps generated using SQLite databases are sufficiently lightweight and can be generated easily, therefore removing the need for wireframes. Instead, a duplicate prototype using a demo database could be used for a dynamic and more realistic page pre-view in the editor.

Application model

Foundational work for the AI4MDE platform, established by Driessen et al. (2020) [R20], introduced the concept of the "application model." This model specifies relevant subsets of one or more "domain models," enabling various views to interact with the same underlying model following the MVC design pattern.

3 Related Work

Template-based approaches

Expanding the application model into the application component introduces the specification of user interactivity with shared models. This step presents an opportunity to consider the user interface, aligning with end-user requirements described in other models such as the use case diagram.

In the realm of web development frameworks, Model-View-Template (MVT) architectures, exemplified by Django, offer a robust approach to software development. Streamlining the MVC paradigm adopted by frameworks like Phoenix, Laravel, and Ruby on Rails, MVT enhances the separation of concerns like requirements and presentation using template-based generation mechanisms. These mechanisms are utilized during generation, leveraging the application component.

Jinja2, a powerful templating engine commonly used in web development frameworks, facilitates the creation of dynamic HTML templates [d*Jinja2*]. One notable feature of Jinja2 is its support for macros, which allow developers to define reusable chunks of code within templates. Macros streamline the development process by promoting code re-usability and maintainability. In the context of styling in HTML templates, Jinja2 macros prove invaluable, enabling developers to encapsulate CSS style sheets or inline styles within reusable macros. This approach ensures consistency across the application's visual presentation while minimizing redundancy in code.

Relevance of UML diagrams and MDE

In a study by Fernández Sáez et al. (2018), the impact of utilizing UML diagrams in software maintenance was examined [Sáe18]. The research highlighted the efficiency gains attributed to the integration of diagrams, particularly beneficial for non-senior developers unfamiliar with the codebase. Noteworthy improvements were observed in comprehension, communication, debugging, and overall software quality, as reported by a majority of surveyed software maintenance professionals.

Utilization of MDE techniques, as advocated by Mohagheghi et al. (2013), facilitates automation of repetitive tasks and promotes higher levels of abstraction, enhancing productivity and reducing development time [MGSF13].

Additionally, Nugroho (2009) delved into the intricacies of UML diagram granularity and its consequences [Nug09]. The study analyzed the varying levels of detail in UML diagrams and their corresponding enhancements of software development processes.

The success of other platforms, such as Mendix, further underscores the value of MDE [HM]. While these platforms provide rapid development capabilities, they often come with drawbacks, including high operational costs and vendor lock-in. For example, Mendix imposes limitations on database integration, requiring complex workarounds for feature testing. In contrast, open architectures like AI4MDE offer virtually limitless possibilities for prototyping and system verification.

Diagram types

Koc et al. (2021) contributed to the discourse on the significance of UML in software engineering through a comprehensive literature overview, shedding light on its multifaceted role in modern software development practices [KEBP21]. Use case diagrams and activity diagram appear in literature in similar frequency. Class diagrams appear more often, while sequence and state machine diagrams appear the least often.

However, state machine diagram usage and sequence diagram usage remain in close proximity to the three most frequently used UML diagram types. Their inclusion could potentially enhance the capabilities of AI4MDE. In combination with activity diagrams, for which a definitive role during prototype generation was less obvious, the support for complex prototype behavior could be strengthened. Relying on use case diagrams and activity diagrams to manage intricate sequences resulted in cluttered diagrams, prompting the adoption of custom methods as a preferable alternative. This approach may yield better results as more diagram types are included, provided each is assigned a clear purpose in describing parts of complex behaviours. Distinct requirements in the generated prototypes can be intuitively distributed among them to avoid cluttering.

Model-centric software development trade offs

Lethbrigde, Forward, et al. (2008) Surveyed software professionals on MDE practices [FL08]. While the field of Software Engineering has changed significantly since 2008, their results shed light on perception of "modeling" during software development, and its usefulness compared to code-centric development styles. Notably, a prominent downside to the model-centric approach was identified when code diverged from the models, rendering models inaccurate.

Different solutions to model inaccuracy have been proposed in literature. Rasha et al. (2021) utilise reverse translation from source code to UML diagrams [GAMAF21]. Their work supports Python source code. Their strategies could potentially be implemented allowing for prototypes to be used as input on AI4MDE. Meticulous testing is another option. Arora et al. (2018) address automatic test case generation where functions are defined by UML diagrams [AB18]. Test case generation was also employed efficiently by Lasalle et al. (2011) [LPF11].

The perceived risk of model inaccuracy is reflected in the feedback from test users. Encountering inconsistencies while editing the application component, a user observed that previously defined restrictions could be compromised during the interface step. Fewer changes could be allowed for users during the interface step to efficiently address model inaccuracy. This further clarifies the distinction between tabs in the editor where requirements design is separated from styling and minor adjustments proposed by this research.

3.1 Unique contribution

In a literature review by Ozkaya et al. (2019), 58 UML tools were analyzed across various dimensions [Ozk19]. Key findings include: 32 tools use XML/XMI as their exporting format and 29 tools support code-generation, with 18 supporting round-trip engineering. Java emerges as the most popular programming language for this, as seen in Figure 3. The support for Python apps is notably low. Most tools provide user manuals, but interactive guidance is rarely supported.

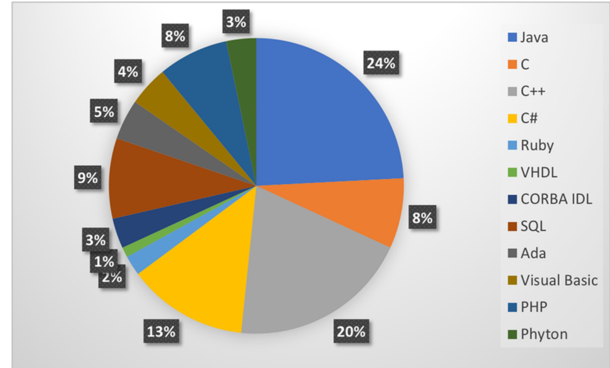

Figure 3: Programming languages supported by the UML tools for code generation [Ozk19]

There is significant potential for improvements in enhancing user experience for MDE platforms. Utilizing Python as the programming language for prototypes, coupled with frameworks like Django, enables comprehensive management of prototype applications. A reduced barrier to entry is achieved in line with the primary objectives of MDE. Moreover, the JSON metadata format offers fewer restrictions on content while maintaining comprehensiveness compared to the standard XMI format. The effectiveness of custom Python methods is highlighted as a cost-efficient alternative to modeling non-CRUD behavior, which otherwise clutters the use case diagram. It was noted during testing that a user intuitively implemented a unique custom method. This emphasizes the value of this alternative approach. Especially when complemented by clear error codes and generation times under 5 seconds, a streamlined process facilitates easy debugging of the Python methods added to the class diagram. It does, however, falls short of providing interactive guidance in the editor.

4 Research Methodology

The principles of Example-Driven-Development were followed during this research. The project was divided into three main parts; theoretical, implementation and testing. The research iteratively looped over these three phases, depicted in Figure 4, each time increasing in complexity.

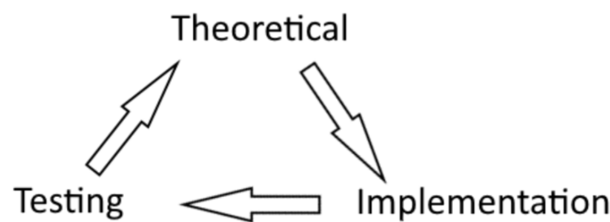

Figure 4: phases in example-driven development

This iterative method facilitated the continuous enhancement of AI4MDE with progressively complex features, enabling end-to-end improvements each cycle. Additionally, it allowed individual

students to concentrate on specific aspects within the broader project despite the platform's extensive scope.

However, a lack of consistency across designs also emerged as a result. Throughout iterations, major overhauls rendered previous work obsolete. Moreover, designs evolved during implementation, resulting in discrepancies and bugs.

The outcome nevertheless comprises innovative contributions of every participant, serving as a testament to the power of collaboration.

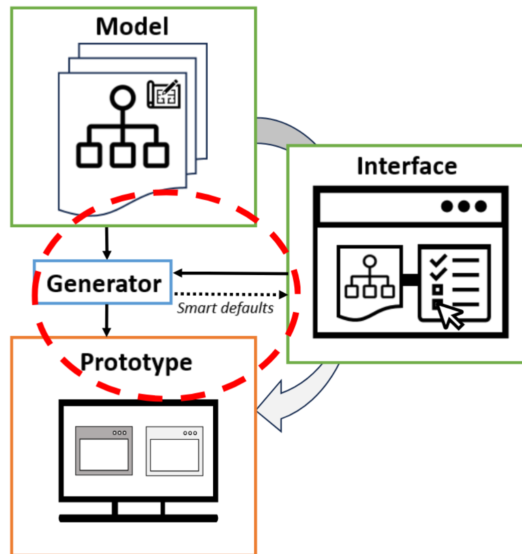

Figure 5: This research specifically focused on developing the prototype generation architecture capable of utilising use case models in conjunction with other metadata

5 Conceptual Architecture

In this chapter, a detailed prototype generation architecture is proposed alongside three pivotal concepts: application components, section components, and smart defaults. Their interconnections with models described in UML diagrams are explored to underscore potential applications. Figure 6 illustrates three distinct blocks in different colors, representing steps in the generation process. Each block highlights essential coding artefacts pertinent to the corresponding step.

The separation of concerns mirrors the sequential steps taken by the user as depicted in Figure 5, beginning with model creation, followed by interface design, and concluding with prototype generation. However, the emphasis on coding artefacts provides a distinctive layer of detail. Combining user input for requirements and UI design offers a more realistic representation of the generation pipeline. The generator operates in the background and combines user input from the model and interface steps to generate the prototype output. Therefore, the model and interface steps are grouped into the same block responsible for generator input, both are featured in the JSON file used in the implementation of the proposed architecture.

5.1 Generator Design

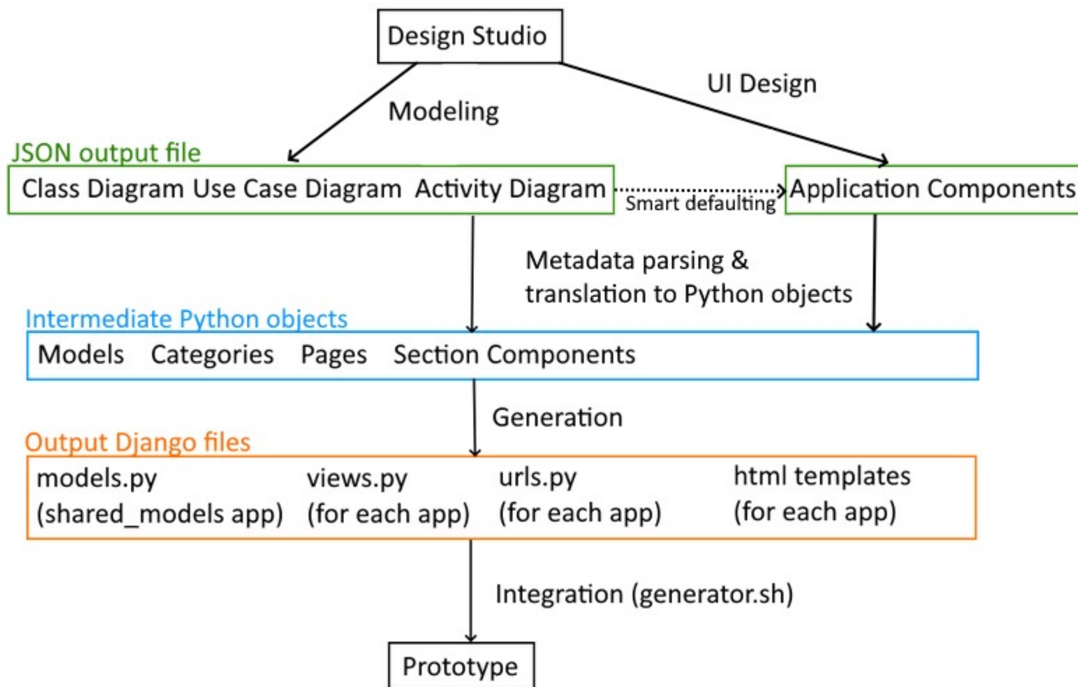

Figure 6: A detailed visualization of the proposed architecture, including coding artefacts at each step

Diagrams

First, users specify the prototype system by creating models in UML diagrams, depicted as the initial design studio block in Figure 6. This process delivers the artefacts in green. For instance, classes represent database expansion through their addition in the JSON file. Prototype applications connect to the expanded database and utilize the new objects if they are relevant to the application.

Application component

This relevance is specified by the selection of diagram subsets called fragments in the design studio. These fragments form the core of the application components which contain all information gathered during the interface step. As Figure 6 illustrates, some user input is also requested about UI design to complement the technical requirements. Settings about pages, categories and styling for example, are changed here for each app individually. Other values, however, are inferred through previously created models and presented to the user as smart defaults values.

5.2 Smart defaults

A dotted line in Figure 6 represents smart defaults and signifies preliminary calculations assisting users during the interface step. For example, additional actors in the use case diagram correspond to new application components where use cases that the actor interacts with are defined in more context. AI4MDE does not support the detailed textual aspect of use case models represented by

use case specification. However, the application component effectively addresses this and could be further defaulted with their addition in the future.

Section Component

Intermediary objects represent these interactions in full context. Specific operations on class diagram models are defined in section components. These are placed on pages to fill sections, defining what a user can do on that page. A direct mapping of use cases to section components is employed as a smart default. Similarly, pages are created for each section component. Furthermore, extends relations in the use case diagram translate to a fusion of pages. Finally, section components acting on identical models are grouped into compound sections with various options that allow for a multitude of operations to be defined compactly.

Every section component reads a primary model, determined from the use case name by default. Other CRUD behaviour can also be determined with use cases of the form "create model" or "add model". If this is the case, the compound section is extended with necessary data. Similarly, some users of prototype might be allowed to edit only specific parts of models or delete them, whilst others have a custom method available.

Section table

Objects depicted in the blue block in Figure 6 compose the coding artefacts during generation. Metadata is parsed and translated to facilitate their creation. The generator temporarily stores them in a section table, which acts as an overview of the entire prototype that will be generated and includes UI information for each app, extracted from the application component.

Apps

Lastly, in the orange block in Figure 6, artefacts composing the generated apps are displayed. Our open architecture offers users full access to the generated app, ran locally for immediate inspection. Users maneuver through the different apps and access the database that corresponds to the class diagram. In each app, different subsets of models are presented as dictated by the fragments of the respective application component where permitted operations on these models are further specified.

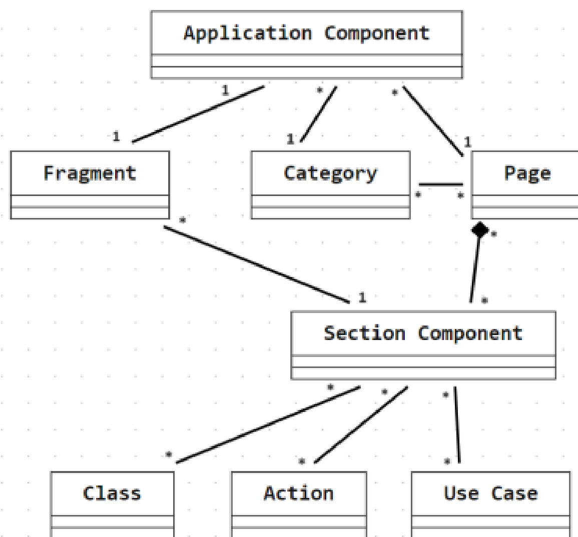

Figure 7: Application components connect diagrams to pages utilizing the section component intermediary

5.3 Platform metadata model

Dependencies in the proposed objects are modelled in Figure 7. The application component connects diagrams to pages. Between apps in the prototype, unique fragments of models described in diagrams can be used. This is an extension of previous work applied to multiple diagrams, where previously domain models could be represented by application models [R20] in class diagrams only.

6 Technical Architecture

This chapter describes implementation of the prototype generation architecture. The generation process commences with the provision of metadata from the AI4MDE platform in a structured format, as illustrated in Figure 6. Subsequent steps in the proposed conceptual architecture is outlined and coding decisions are elucidated, alongside discussions on custom methods and prototype databases. In total, 39 files with 3048 lines of code were developed to implement the generator.

6.1 Code Implementation

A shell script called generator.sh can be called to start generation, requiring two inputs: The JSON file generated on AI4MDE containing diagram and UI design metadata, and a name for the output project. Using Django, a Python framework for app development [d*Django*], the command `Django-admin startproject $PROJECT_NAME` is executed to create files used by Django in order for a generic app to run in the context of a project.

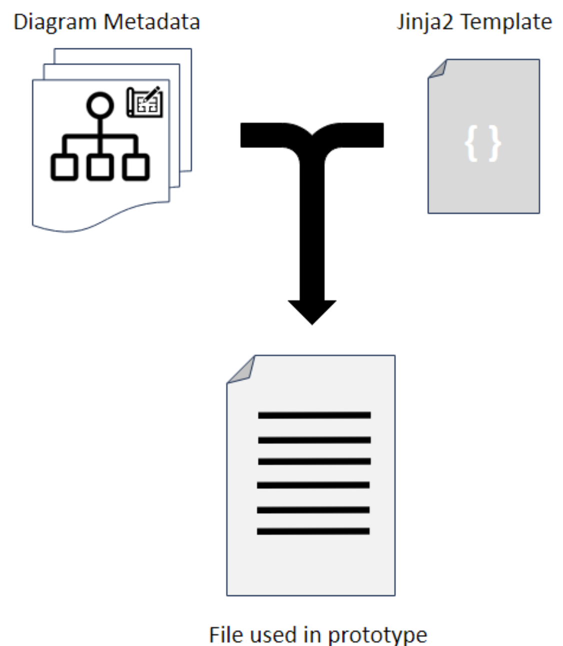

Figure 8: Generation by combining metadata, including UML diagram metadata and application components, with templates using Jinja2 [d*Jinja2*]

Subsequently, the shell script uses Python scripts to create a section table after parsing the application component and diagrams, translating them into Python objects and temporarily storing them. Other scripts are called for each specific file that needs to be generated. The scripts and templates used to generate the files are divided in different source code files. The process of using templates is illustrated in Figure 8 and the resulting Python files are presented as coding artefacts in Figure 6

Shared models

Each app is generated individually, however, models are shared between users. In the context of Django, which utilizes ORM, models are Python objects representing database entries. A models.py file is generated within a special application called "shared_models". Here, the models for all apps are generated including their custom methods. The resulting models.py file is ran inside a Python shell to check for syntax errors. If the custom code in model methods written during design results in an error at this step, they are instead replaced with commented lines. This way, the prototype can be executed nonetheless, apart from non-CRUD operations.

Django apps

The styling and accessible parts of models in each app varies based on section table content. Leveraging a list of application components in the metadata, the script proceeds to execute

Django commands, constructing individual Django applications corresponding with each application component. Additional Python scripts are called to generate essential components like views.py, urls.py, base.html, styling.css, and multiple HTML templates for each application component in the JSON file. Objects stored on the section table fill in templates to facilitate their creation.

Jinja2

The generated Python files operate on Python object representing database entries at run-time. The Python files are then use the database instances to fill templates that result in rendered pages accessed by the user of a prototype app, in a similar fashion to how the templates were used which created the Python files themselves. A comprehensive template engine is used to develop these *templated templates* required for flexible generation of Django apps [d*Jinja2*]. Jinja2 templates tailored to each specific part are utilised. For models.py, the data structure involves a list of classes, attributes, and relations generated based on the class diagram. On the other hand, urls.py and views.py use a list of section components that specify operation on models, instead. Non-CRUD behaviour is not directly supported using the models on AI4MDE. However, custom methods are implemented to address this.

6.2 Custom Methods

Custom code is written by the user on the AI4MDE platform to facilitate methods outside CRUD. By incorporating their class in the fragment, methods are be presented for section components during the interface step. If the user enables custom methods on section components, a button next to an instance of the model on the prototype page is rendered. When the button is pressed, the method of the class in models.py is called, alongside request information and a pointer to the object instance as parameters. Django uses ORM, therefore a Python model object represents the database entry. This allows behaviour at run-time to be programmed intuitively at design-time.

Clear error handling during run-time allows users to check for pre- and postconditions in custom methods. For example, as shown in 9, the custom method "pack" was called when the conditions specified in the class diagram were not met. The button was pressed causing the error to be thrown.

Warehousestaff

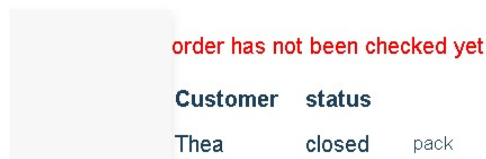

Figure 9: Errors caught at runtime are displayed in red on top of the page

Another feature is implemented using custom methods, addressing derived attributes. A method is called in response to a derived attribute being rendered if the model contains a custom method of which the name includes the corresponding attribute name, alongside a "get_" prefix. These steps can be easily automated if the custom method approach is extended in the future. For example, when an attribute is changed to derived, the custom method could be created. Further extension

of custom methods allows for user-friendly access of database entries, as an efficient alternative approach to ensure expected behaviour that is hard to catch in other models.

6.3 Prototype Database

SQLite was chosen as the database for the generated prototypes for several reasons. Firstly, it aligns with the goals of AI4MDE as it is in the public domain. Secondly, SQLite is server-less and lightweight, making it well-suited for the template-based approach of this research, where new projects including these databases and their corresponding migrations could be generated in just a few seconds. Additionally, SQLite is the default database system used by Django apps. Therefore, extensive documentation exists focused on comparable use cases for prototype apps using Django. This aids development of the generator in addition to supporting developers that wish to extend the generated prototype.

Consistent Database

In the evolution of a software system, modifications to the class diagram necessitate corresponding adjustments in the database. When a new class is added in the class diagram that was missing in the previous prototype, for example, it typically translates to the creation of a new table in the database of the new prototype. In general, our template-based approach builds completely new prototypes from the ground up, meaning that a new empty database is created each time. Any changes made on platform will only be reflected in the prototype after a new prototype is generated. This results in previous data on the old prototype being lost.

To address this issue, the generator.sh script was extended to ensure database consistency. When invoked with the main parameter $ PROJECT_ NAME == "latest", the script initializes the generated project with the database file from the preceding project labeled "latest." Additionally, this extension offers the option to create a backup of the database, although this feature is disabled by default.

In order to streamline the process of invoking the generator script following any modifications to the diagrams or the application, two PowerShell scripts were developed. The first script automatically transfers the JSON files from the download directory to the working directory of the generator.sh script and runs the generator. Furthermore, the second PowerShell script orchestrates the execution of this first script as a new job each time it detects the download of the "runtime.json" file in a continuous while loop. As an alternative approach, a custom input path for the runtime.json file was implemented, empowering users to select the execution location of the generator.sh script.

Limitations

SQLite, being a self-contained, server-less database engine, has certain limitations compared to more robust database systems like PostgreSQL or MySQL. One such limitation is the initial migration stored in the SQLite database resulting in the inability to update the schema for an existing database.

For instance, the implemented generator fails to account for data consistency when a class is renamed. Consequently, existing data associated with a renamed class persists in the copied database, despite

the absence of corresponding models. Also, a new table should be generated for the renamed class, but this does not happen either due to the initial migration on the copied database. The data from the table of the old class is copied over without any changes made and the prototype crashes when it attempts to load the renamed data.

7 Case Studies

In this chapter, the implemented architecture is demonstrated through a concrete worked example concerning a webshop. During this research, three case studies were examined in total. The scope of the first two was significantly smaller and will be described briefly. The final case study is presented in detail and the resulting prototype generated using models describing it is discussed. The requirements of a webshop are first determined, after which a conclusion is drawn about the capacity of the generator to fulfill them.

Figure 10: A diagram with use cases of a Webshop application visualized per Actor.

Use cases Per Actor Webshop

One of the central user types of the webshop app is Customer. Three pages were identified, enabling users of this type to order products. First, it should be able to view all the available products sold by the webshop. On that page, customers should be able to also select products and add them to their shopping cart. customers should subsequently be able to see their shopping cart on another page and update the quantity of each selected product whilst viewing the total price. Also, user account information should be updated on a third account page to facilitate delivery.

The other two user types (Warehouse Staff and Order Manager) mostly act on the orders that should be created when the user confirms its shopping cart. Aside from interactions with order models, which order managers notably have to check before order are send, Order Managers also create and manage products on a webshop and can view all customer accounts to verify transactions made on the webshop. For each actor, a simplified use case diagram can be seen in Figure 10

The Warehouse Staff application is the simplest among the three actors. It requires two pages: The first is used to re-order products to manage inventory and the second is used to view checked orders that need to be packed and send to customers.

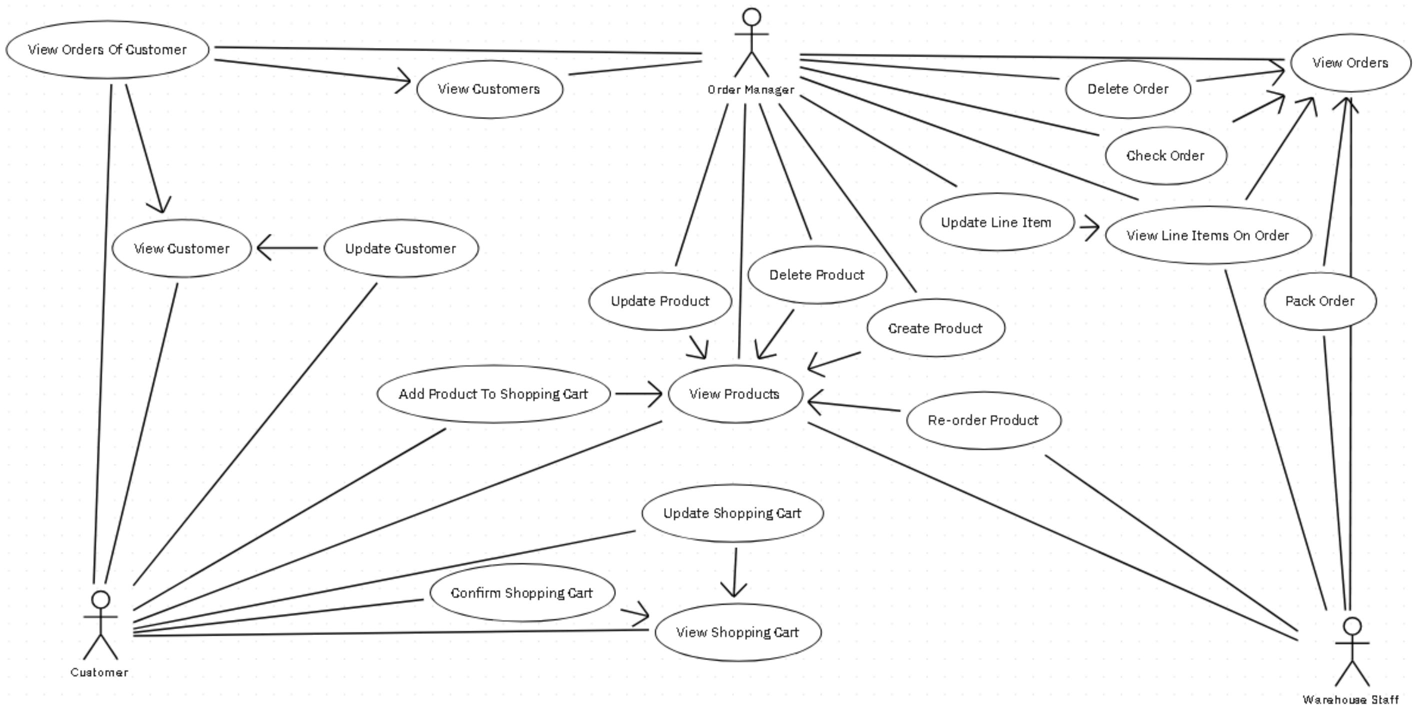

Figure 11: A detailed use case diagram of a Webshop application that displays shared use cases. A simplified version can be seen in Figure 10

Main use case diagram webshop

All three actors of the described system are displayed in Figure 11, which includes shared use cases connected to multiple actors denoted as interactions. These interactions represent actor privileges. Other types of relations between nodes are available on AI4MDE as well. Among these, a special type of dependency called `<<extends>>`, is utilized during this research to connect two use cases. This commonly indicates alternative flow [ZV09]. In the implemented prototype generator, this relationship is mostly used to extend the section component that will be generated represented by the read / view use case. Use cases of this variety can also be implicated when other use cases lack this type of dependency. Reading / viewing instances of a class and adding options for the user is modelled intuitively using `<<extends>>`. Additionally, `<<extends>>` indicate multiple sections required on a prototype page, when the classes in the use case names are not identical.

Source for smart defaults

In order to load smart defaults, an algorithm walks through the use case diagram starting from the actors selected for the application component. The names of the use case are matched to classes in the class diagram, seen in Figure 12. When a use case name contains a class name as a substring, the class is added to the fragment of the application component. The use cases themselves are also added, as well as actions that share their name with any of the use cases. Similarly, default values for section components, pages, categories and styling are determined using the fragment, as implemented and discussed in detail by de Hoogd et al. [dH24].

Class Diagram Webshop

In figure 12, the class diagram is displayed. It includes models required for a webshop app to function. Three user types are features, which correspond to the three actors and applications of this case study. Notably, they all connect with generic User class through a generalisation, which indicates inheritance. The User class is implemented by inheriting the AbstractUser model from the Django Framework. The Customer class is a special case of user as a result, which facilitates authentication while it also contains relations to other classes in the diagram.

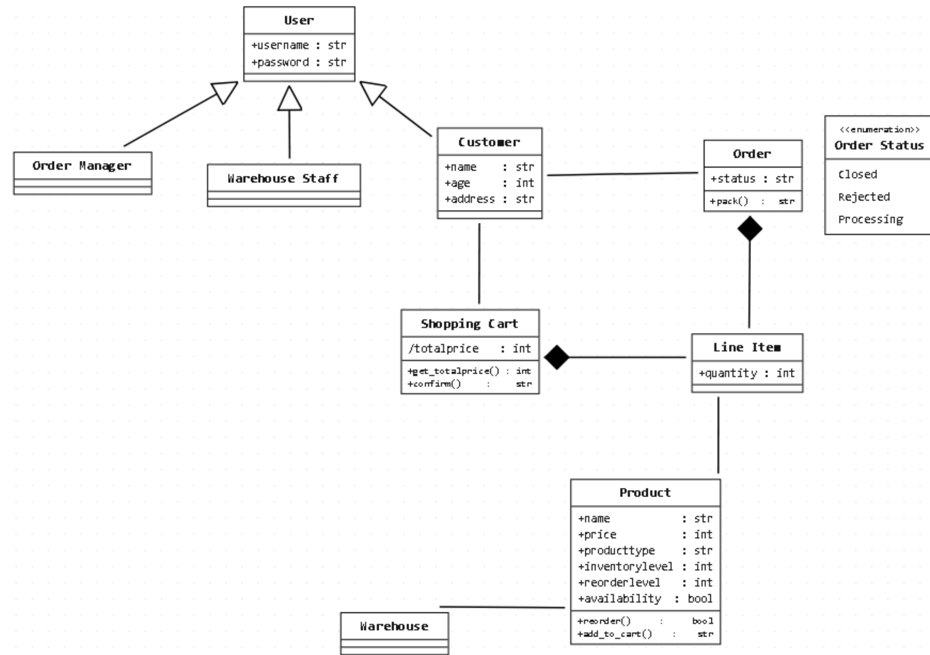

Figure 12: A class diagram of a Webshop application

Activity Diagram Webshop

The activity diagram shown below in Figure 13 displays the swimming lanes of different users of a webshop. Each swim lane contains actions that are only accessible to users of the specific user type and which are necessary to complete in order to start the next action in the activty flow.

In this example, only after a customer has added products to its cart will it be able to confirm the shopping cart to create an order. After an order is created by pressing the confirm shopping cart button, the ordermanager is able to view and check the created order. The status of the order instance is updated, which allows the warehouse staff to pack it and send it to the customer.

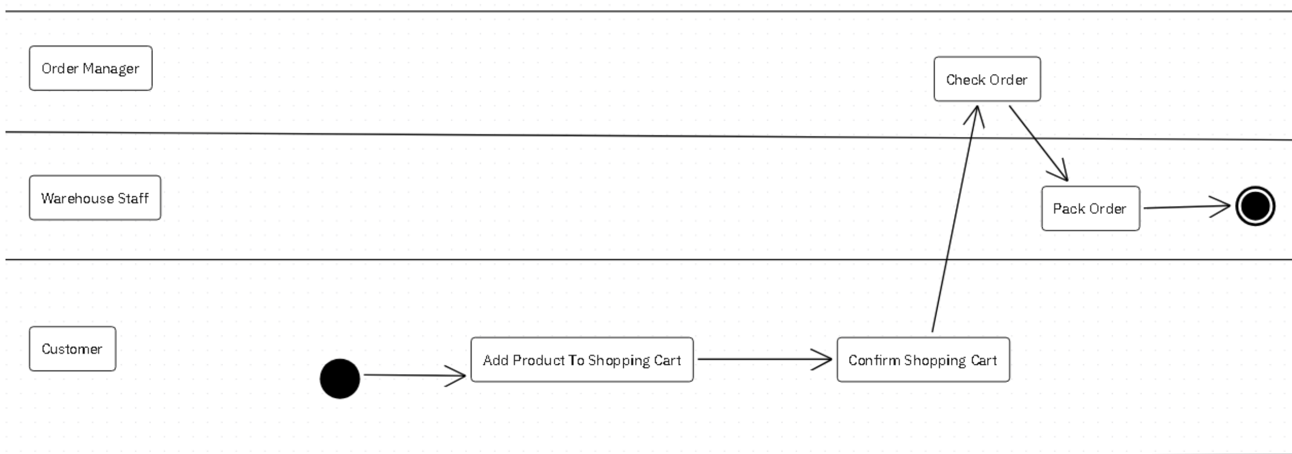

Figure 13: An activity diagram with swim lanes of a Webshop application

7.1 Results

This section summarises the capacity in which prototype apps could be generated during case studies. Lessons learned during case studies are presented, in addition to the final prototype generation capabilities of the implementation of the proposed architecture.

Prototype capabilities

Because of time constraints, the implemented generator lacked the capacity to generate a prototype fully supporting all requirements. Unimplemented queries meant customers were unable to specifically view products of a selected type and unavailable product were never hidden. Shopping carts were shared with every other customer as well. However, products could be displayed to customers, and functional buttons to call custom methods like "Add to Cart" were rendered successfully.

Queries

Fundamental structures for queries were created [vD24], aiding its implementation in the future. A potential solution could utilise the activity diagram to determine default queries based on the action preceding the current action connected with a section component, causing only checked orders to be presented to warehouse staff, for example

Activity diagram strategies

The platform currently supports generation utilising a single diagram of each type. For behavioural diagrams, more can be created for documentation purposes but bugs caused the JSON file to exclude all except the oldest, leading to invisible data for the generator when their content is accessed in the application component. Bugs in the editor tool also hindered development because of the many edges between nodes in activity diagrams, which could not be updated and did not display guards.

However, the activity diagram was utilised, despite difficulties, during earlier case studies to determine buttons on pages corresponding to actions in the activity diagram. Their intended impact on prototype flow was no longer supported by the generation architecture in the end. The introduction of pages in the activity diagram muddled its distinctive requirements engineering purpose and was delayed until later in the generation process, where users could create them during the interface step. Similar to categories depicted in Figure 2, utilising links on the sidebar and on prototype pages, users are provided with a means to navigate through the prototype app. Finally, activity diagram utilisation was dropped when pages no longer relied on them and time constraints prevented further development,

Custom method support further provided alternatives to activity diagrams for describing behaviour. Their flexibility allowed for most of the requirements during the final case study to be fulfilled. To achieve a more robust adherence to the principles of MDE, the integration of activity diagrams would substantially enhance the comprehensiveness and methodological rigor of this approach

8 Validation

In this chapter, quality of the implemented architecture is determined by collecting impartial user feedback. Tests and results are provided, followed by a qualitative analysis (n=7) that focuses on ease-of-use ratings compared to previous software development experience.

User testing

Users with various backgrounds tested the proposed prototype generator architecture which was implemented on the AI4MDE platform, following a set of instructions referred to as the user script. Test users would often go off-script, when features of the platform inspired them to be creative.

While the impartial users acted completely on their own accord during testing, they were informed about some of the details of MDE beforehand. A conceptual introduction dependant on the familiarity of the user with similar concepts commenced each validation session. Subsequently, the abstract introduction was reinforced with recognizable examples on the platform. Diagrams of case study 3: Webshop revealed how certain ideas were displayed on the platform and some of the possibilities that MDE could offer.

8.1 Test Summary

First, users visually inspected the present content on the AI4MDE platform and pressed the prototype button, sufficiently aware of how the prototype being generated should work. Afterwards, the test user interacted with the prototype to confirm or deny if generation had been successful. To collect this data, they were asked to fill in a questionnaire, included in the appendix of this research 10.2.

Afterwards, users added classes, attributes and an association to another class in the existing class diagram. Subsequently, the use case diagram was updated with an additional actor and two use cases. Interactions were drawn to the newly added actor and another actor. A connection was drawn from one of the new use cases, which was of the create crud type, to the other new use case, which was of the view crud type, as an extends relation. The additional actor was also constructed in the class diagram as a class of the same name as the actor, that generalized to the abstract user class.

After the test user added these changes to the existing diagrams, the application components were updated. A new application component corresponding to the inserted actor enabled test users to interact with smart defaults at this stage. For each interface step, the test user either confirmed or updated values. Afterwards, the generated prototype was inspected, this time focusing on the specific changes made during design time and their effects in the prototype. Finally, the user answered the remaining questions.

8.2 Results

A complete overview of user feedback is available in the appendix 10.2. To visualise the results of the survey clearly, diagrams show the distribution of feedback per question rating a specific part of the implementation. Overall, test users responded with high ratings. The lowest grade given was two stars out of five, which was given 9 times, out of a total of 105 datapoints, of which 4 these were given by a single test user.

8.3 Qualitative analysis

Outliers

Apart from one test user, who provided the lowest rating in 10 of the 15 rating questions, data looks generally consistent. A possible explanation for the lower ratings given by the test user is a lack of recent experience with software development tools. The test user was noted having a unique difficulty with the questions, which is also reflected in textual feedback. For example, writing that it lacks a sense for the consequences of working with the platform and feeling inadequate to properly rate some of the technical aspects of the implementation. Our survey required an answer to every rating question, otherwise it would not have rated some of the parts that ended up receiving low grades. For that reason, the more technical questions in which the outlier test user felt least adequate are excluded from the following qualitative analysis.

Variability in Prior Experience Levels

Despite the limited sample size (n=7), the population was segmented into three distinct groups: the first comprising two test users possessing minimal software development experience, the second consisting of three individuals at the junior software developer level, and the final group encompassing two users with advanced professional expertise. These groups were subjected to qualitative analysis.

As depicted in Figure 14, individuals with limited experience (represented by lighter colors) in software system design significantly contributed to lower ratings within the surveyed population.

This trend is pervasive across most responses, as evidenced by Figures 15 and 16. However, there are exceptions, particularly in inquiries regarding accessibility and the reflection of use case model changes. Because of the low sample size, it is unclear

It is noteworthy that the test user discussed as being an outlier, classified within the less experienced population, may have influenced the observed trend in Figures 14 and 15. However, this explanation seems inadequate given the final question response, wherein another less experienced user also provided a notably low rating for the overall prototype quality, as can be seen in Figure 16.

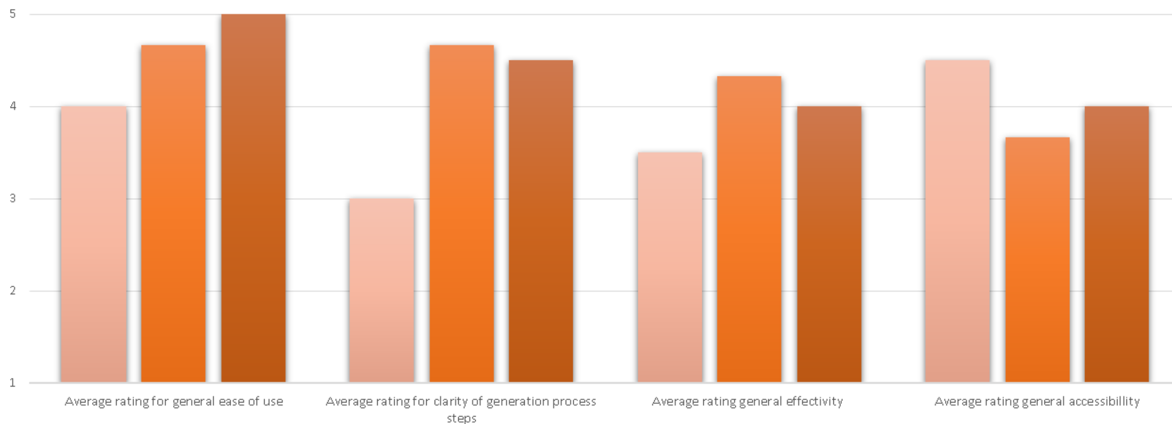

Figure 14: Difference in user response on the first four questions by user level of experience from light to dark colours

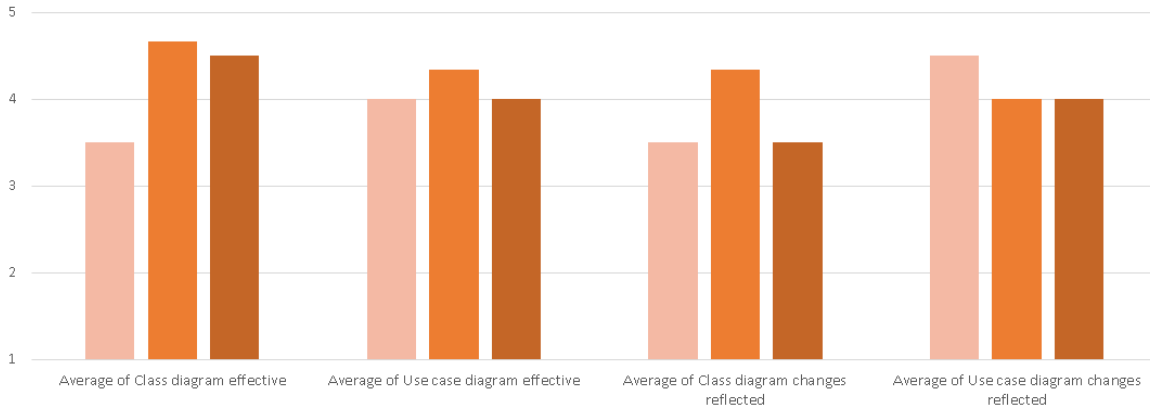

Figure 15: Difference in user response on the first four questions by user level of experience from light to dark colours

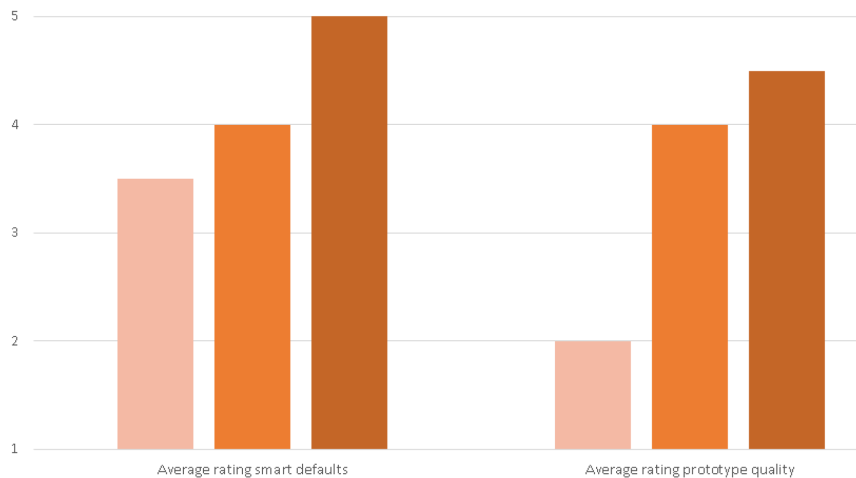

Figure 16: Difference in user response on the first four questions by user level of experience from light to dark colours on final two questions

8.4 Reflection

Survey quality

The extensive survey provided many data points per test user, which resulted in a rich profile that lends itself for analysis and prioritization of future extension of the AI4MDE platform. However, because of the small sample size, trends are difficult to spot conclusively and outside of the AI4MDE context results can not be used to indicate significant correlations. In addition, questions targeted specific parts of the implementation that were often poorly understood by users.

Missed opportunities

The extensive testing of the proposed generator architecture underscores differences in test user responses among populations divided on prior knowledge on software system design. In retrospect, it is evident that employing a larger sample size, while refraining from introducing overly complex concepts, could potentially yield more academically robust results.

Moreover, the overarching objective of this research, aimed at bridging the divide between those with technical expertise in computer systems and those without, could have been advanced further by focusing on broader global perspectives. Extensive technical implementation can provide every option wanted by end-users. However, it can detract focus from options which they need.

Test user response on use case models showed little variance. During this research, features were dropped because they cluttered the diagram. They were not missed, and might have lowered the rating if left included. Further employing this strategy would have increased clarity and likely contributed to higher and more consistent ratings between differences in test user backgrounds.

Encouraging feedback
While the class diagram is of a more technical nature, it received comparable positive responses to the use case diagram. The overall purposes of these diagrams is clear. Every test user was enthusiastic about MDE. Still, the presented implementation presented a clear problem of a high barrier to entry. Nonetheless, test users engaged with it and encouraged further development to realise its recognized potential.

9 Conclusion

Designing an effective prototype generator requires taking many perspectives on the Software development process. In this research, it was examined through a joined effort that extended a platform supporting three UML diagrams. Apart from the `<<extends>>` connection between use cases, experimental features that were implemented throughout the project ended up being mostly dropped because they did not increase clarity for the end-user. However, the combination of data and introduction of the interface step aligned seamlessly with the MDE values.

At the outset of this research endeavor, the AI4MDE platform solely supported class diagrams. While a prototype could be constructed using a WYSIWYG editor, its usability left much to be desired. Presently, while the class diagram continues to serve as the fundamental blueprint, the platform has evolved to encompass use case models, describing permissible user actions within the prototype. This evolution is facilitated by the integration of an application component for each distinct application within the prototype, where smart default values aid the user based on use case models. The introduction of the application component editor allowed for a significant advancement over its predecessor in enhancing overall usability of the platform.

While test users provided pointed criticism of the implementation, their feedback also serves to validate the efficacy of MDE approaches. Inefficient software design persists as a widespread issue. However, leveraging comprehensible models that strictly define a system, requirements can be clarified by individuals lacking intricate technical expertise. Accessible prototype generation thus provides clarity in navigating the often intricate process of computer program design

10 Discussion

10.1 Extending the AI4MDE platform

Previous work on the AI4MDE platform [Zei23], [Hep23] proposed potential avenues for enhancement, which were taken into account. However, due to the limited scope of this thesis, in-depth investigations into these suggested approaches were not feasible. Nonetheless, the results of brief inquiries conducted to assess the potential value of their inclusion are briefly discussed.

LLM for difficult methods

Zeidi et al. (2023) advocated for the utilization of a Large Language Model as an intermediary tool for interpreting diagrams and generating customized code [Zei23]. Their study highlights the promise of NLP techniques in streamlining code generation processes within MDE environments. This approach appeared particularly promising for addressing code generation challenges in scenarios involving Non-CRUD operations.

However, our template-driven generation process encounters challenges when integrating separately generated code into a not yet existing codebase. Therefore, the most efficient way to address this issue would be to generate the prototype multiple times use automatic testing and debugging tools that also grant the NLP some leeway and could intervene if it hallucinates.

Component library

Collaborative design functionalities, such as locking mechanisms for concurrent editing, have been investigated by Hepkema et al. (2023). Making use of a newly added component library, their research enhanced team collaboration and productivity during design sessions on platform [Hep23]. Saving parts of diagrams to the component library for later re-use inspired potential uses of section components.

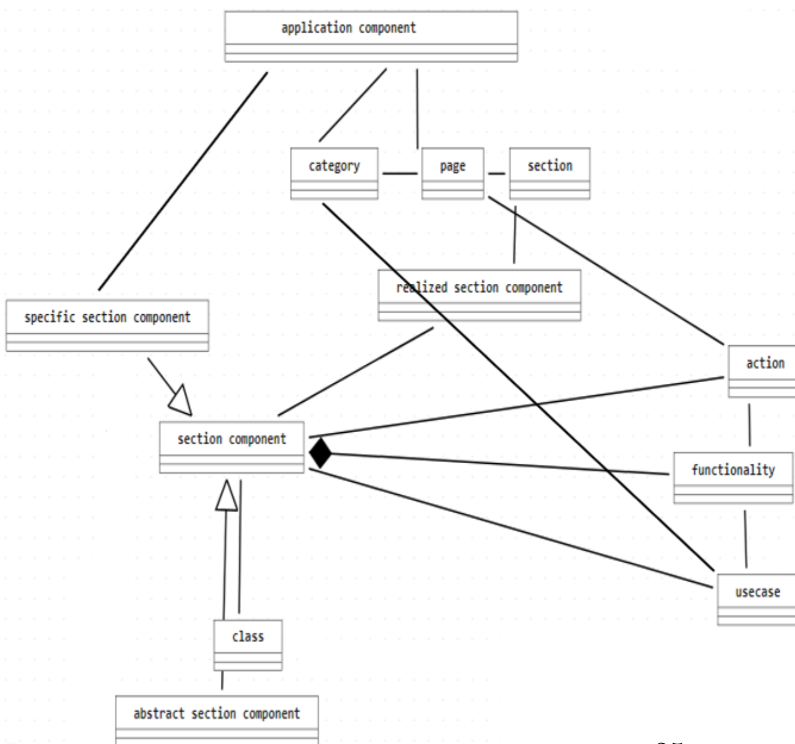

Figure 17: Re-usability of section components in an earlier version of the mapping seen here: 7

The concept of abstract section components was introduced, that could function as interfaces of models in multiple diagrams, specifying their interconnectivity. It displayed in Figure 17. Similar to the final implementation, use cases could indicate new section components or extend existing ones. However, abstract section components would not necessitate the existence of a class in order to be defined. If abstract section components could be saved to a component library, custom methods could be included as a method with an undetermined class, allowing for their realisation when a non-crud use case includes the name of the method in addition to a class name.

In addition, specific section components that match a use case name combined with a class name could receive priority during smart default value determination, over abstract section components. Speedup and ease-of-use during the development of new prototypes would result from the provided re-usability as custom methods would ideally only have to be written once and could be altered from one non-crud use case to the next.

10.2 Future work

LLM

While it has been demonstrated that LLM's are a potential solution for the problem of less straightforward code [Zei23], further efforts are needed to integrate this technology into the code generation process. As LLMs continue to evolve and accommodate larger token sizes for prompts, it becomes feasible to provide the AI with contextual information from the generated code, thereby facilitating the generation of code that is more seamlessly integrable. Moreover, combining this approach with automated testing throughout multiple iterative generations of prototypes could enhance the efficacy of this approach.

Components

The component library as it was introduced in earlier work [Hep23] could be expanded to encompass more of the platform. Application components and section components could be saved in this library. Re-usability during the interface step could be improved and pre-packaged application components like authentication could be defined and updated during the interface step using this approach.

Prototype Database Management

One avenue for improvement in database design for the generated prototypes could involve implementing mechanisms to record all migrations systematically. This would provide a comprehensive history of database schema modifications, aiding in version control and facilitating rollback procedures if necessary.

While that may be ambitious, another potential approach could be to devise a method for loading data from a previous database into a separate JSON file instead of copying the entire database file. Subsequently, a fresh database with the correct initial migration could be created, aligning its tables with the new models. A test user recommended adding a feature to upload structured data to the prototype database. This would allow existing excel sheets, for example, to be used in new prototypes. The problem of data loss could also be addressed in this way, similarly to the JSON import/export option. While this strategy offers the benefit of ensuring data integrity during schema evolution, it introduces room for error in the implementation during loading and does

not solve the underlying problem of conflicting transformations. Therefore, future efforts should focus on addressing these challenges to streamline the migration process and enhance database management.

References

[AB18] Pardeep Kumar Arora and Rajesh Bhatia. Agent-based regression test case generation using class diagram, use cases and activity diagram. *Procedia Computer Science*, 125:747–753, 2018. The 6th International Conference on Smart Computing and Communications.

[Agg22] B. van (Bram) Aggelen. Enabling data-driven wireframe prototyping using model driven development. *Thesis Bachelor Informatica & Economie, LIACS, Leiden University*, 2022.

[d*Django*] developers *Django*. Django documentation. https://docs.djangoproject.com/en/5.0/, 01-04-2024.

[dH24] P. (PIETER) de Hoogd. Synergizing uml class modeling and natural language to code conversion: A gpt-3.5-powered approach for seamless software design and implementation. *Thesis Bachelor Informatica & Economie, LIACS, Leiden University*, 2024.

[d*Jinja2*] developers *Jinja2*. jinja2 documentation. https://jinja.palletsprojects.com/en/3.1.x/, 20-02-2024.

[FL08] Andrew Forward and Timothy Lethbridge. Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals. *Proceedings of the 2008 International Workshop on Models in Software Engineering*, pages 27–32, 05 2008.

[GAMAF21] Rasha Gh, Rasha Alsarraj, Atica Mohammed, and Anfal A. Fadhil. Designing and implementing a tool to transform source code to uml diagrams. *Periodicals of Engineering and Natural Sciences (PEN)*, 9:430, 03 2021.

[Hep23] S. (Sven) Hepkema. Enabling collaboration in a model design environment. *Thesis Bachelor Informatica & Economie, LIACS, Leiden University*, 2023.

[HM] MARC HERMANS and PÉTER MILEF. Introduction to mendix. https://ojs.uni-miskolc.hu/index.php/psaie/article/view/2218/1677, 08-04-2024.

[KEBP21] Hatice Koc, Ali Erdoğan, Yousef Barjakly, and Serhat Peker. Uml diagrams in software engineering research: A systematic literature review. *Proceedings*, 74:13, 03 2021.

[LPF11] Jonathan Lasalle, Fabien Peureux, and Frédéric Fondement. Development of an automated mbt toolchain from uml/sysml models. *Innovations in systems and software engineering*, 7(4):247–256, 2011.

[MGSF13] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel Fernández. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical Software Engineering*, 18:89–116, 02 2013.

[Nug09] Ariadi Nugroho. Level of detail in uml models and its impact on model comprehension: A controlled experiment. *Information and Software Technology*, 51:1670–1685, 12 2009.

[Ozk19] Mert Ozkaya. Are the uml modeling tools powerful enough for practitioners? a literature review. *IET Software*, 13, 10 2019.

[R20] Driessen R. Umlclass models as first-class citizen: Metadata at design-time and run-time. *Thesis Bachelor Informatica & Economie, LIACS, Leiden University*, 2020.

[Sáe18] Ana Sáez. Studying the benefits of using uml on software maintenance: an evidence-based approach. In *Studying the benefits of using uml on software maintenance: an evidence-based approach*, 2018.

[vD24] S.Y. (Sven) van Dam. A flexible, template-driven generation framework for model-driven engineering. *Thesis Bachelor Informatica & Economie, LIACS, Leiden University*, 2024.

[Zei23] S.S. (Seyed) Zeidi. Synergizing uml class modeling and natural language to code conversion: A gpt-3.5-powered approach for seamless software design and implementation. *Thesis Bachelor Informatica & Economie, LIACS, Leiden University*, 2023.

[ZV09] L'uboš Zelinka and Valentino Vranić. A configurable uml based use case modeling metamodel. *Engineering of Computer Based Systems, IEEE Eastern European Conference on the*, 0:1–8, 09 2009.

Appendix

User script questions and results

Q1 Rate the ease of use for generating a prototype using the AI4MDE platform
Q2 How clearly are the different steps in the generation process (MODEL - INTERFACE- PROTOTYPE) presented?
Q3 How effective is the AI4MDE platform in aiding users to generate prototypes?
Q4 How accessible is the prototype generation process using the AI4MDE platform?
Q5 Do you have any suggestions that could help increase clarity during the prototype generation process?

Feedback from test users

User Expertise	Q1	Q2	Q3	Q4	Q5 Do you have any suggestions that could help increase clarity during the prototype generation process?
mean: σ:	4.57 0.49	4.14 0.35	4 1.07	4 1.12	
ICT employee	5	5	4	3	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.
IT Business owner	5	5	4	4	Als je over een item heen gaat met je muis tips weergeven of uitleg wat er precies gebeurd als je dat item invult.
Full stack developer	4	4	4	3	accessibility kan beter, vooral links en tab navigatie is karig. information icons bij actie elementen voor verduidelijking, kleine tutorial voor nieuwe gebruikers
Wordpress developer	4	4	3	5	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.
Junior Database Designer	5	5	5	5	Misschien uitleg over hoe je de diagrammen kunt ontwerpen. Wat mogelijk is en wat niet.
Medior database dashboard designer	5	4	4	4	The application does not update automatically. The generation button action could be integrated in the platform.
visual basic Experrt	4	2	4	4	"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?"

Q1 How effective are the diagrams in displaying what the prototype should do?
A: Class diagram
B: Use case Diagram

Q2 How well were the changes made in the diagrams reflected in the updated prototype?
A: Class diagram
B: Use case Diagram

Q3 How effective is each part of the interface step?
A: Fragment,
B: Category,
C: Pages,
D: Section component,
E: Styling

Q4 Do you have any suggestions that could help increase clarity during this step?

Feedback from test users

User Expertise	Q1 A	Q1 B	Q2 A	Q2 B	Q3 A	Q3 B	Q3 C	Q3 D	Q3 E	Q4: Do you have any suggestions that could help increase clarity during this step?
mean:	4.29	4.14	3.86	4.14	4	3.86	3.71	3.43	3.71	
σ:	0.70	0.35	0.99	0.64	1.07	1.12	0.88	0.90	1.03	
ICT employee	5	4	4	3	5	5	4	3	5	Als je iets aanpast zou het fijn zijn om te zien wat dat voor gevolgen gaat hebben voor andere componentjes
IT business owner	5	4	4	4	5	5	5	3	5	
Full stack developer	4	4	4	4	3	4	3	3	2	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
Wordpress developer	3	4	5	5	2	3	4	4	4	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.
Junior Backend Developer	5	5	5	5	5	5	4	5	4	Het verschil tussen de componenten is duidelijk en het is makkelijk om in een iteratief proces met prototypes het gewenste effect te krijgen.
Medior database dashboard designer	4	4	3	4	4	3	4	4	3	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.
visual basic Expert	4	4	2	4	4	2	2	2	3	Nog te weinig gevoel bij. Meer mee werken om goed te kunnen begrijpen wat de invloeden zijn op het eindresultaat.

Q5 How worthwhile is the addition of support for:
A: Rules or queries in the generated prototype?
B: Activity diagram to dictate the flow of a prototype?
C: Categories in the sidebar of the prototype to group pages based on the data accessed on the page?

Q6 Do you have any other feature in mind that is missing in the current form of AI4MDE, which you would have liked to see?

Q7 Rate the quality of the default values based on changes in the diagram found in the interface step
Q8 Rate the quality of the generated prototype
Q9 Would you use AI4MDE to design and generate prototype software?

Feedback from test users

User Expertise	Q5 A	Q5 B	Q5 C	Q6: Other Feature in mind that is missing?	Q7	Q8	Q9: Would you use AI4MDE?
mean: σ:	4.57 0.49	4 0.53	4 0.76		4.14 0.64	3.57 1.18	
ICT employee	5	4	4	Misschien nog verder automatiseren doormiddel van een script die vraagt wat je wilt aanpassen en dit dan dus ook voor je doet	4	4	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.
IT business owner	4	4	4	Zou mooi zijn als je bij de interface direct vkan zien hoe het er uit komt te zien, ivm juiste kleurkeuze	5	4	Jazeker
Full stack developer	5	5	4	tutorial, documentation/help, user log of actions and events	4	3	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.
Wordpress developer	4	4	5	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	3	2	In the future maybe. Main improvements: more explanation on the choices that could be made and sidebar with generated website
Junior Backend Developer	5	3	3		4	5	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.
Medior Database dashboard designer	5	4	5	Ability to pick a subset of Class Diagrams as it could get quite large and ineligible.Color coding or visual differences in diagrams.	5	5	Definitely if you implement my suggestions.
Visual basic Expert	4	4	3	Structured data uploaden, zoals excelsheets om te databse te vullen.	4	2	Nog niet... Behoeft nog wat werk voor ongeoefende gebruikers.

General platform and generation examination

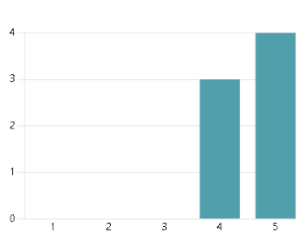

Figure 18:
Rate the ease of use for generating a prototype using the AI4MDE platform (1-5)
mean: 4.57
$\sigma = 0.49$

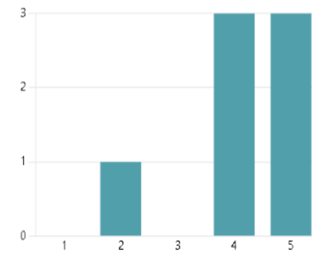

Figure 19:
How clearly are the different steps in the generation process presented? (1-5)
mean: 4.14
$\sigma = 0.99$

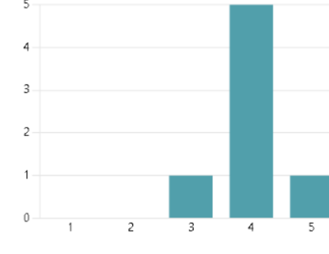

Figure 20:
How effective is the AI4MDE platform in aiding users to generate prototypes? (1-5)
mean: 4.00
$\sigma = 0.53$

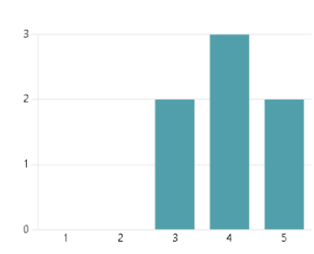

Figure 21:
How accessible is the prototype generation process using the AI4MDE platform? (1-5)
mean: 4.00
$\sigma = 0.76$

Testing using edited models and interface values

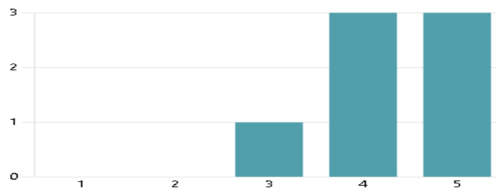

Figure 22: How effective is the class diagram? (1-5)
mean: 4.29
$\sigma = 0.70$

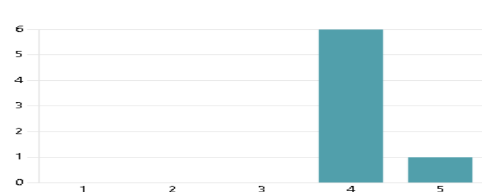

Figure 23: How effective is the use case diagram? (1-5)
mean: 4.14
$\sigma = 0.35$

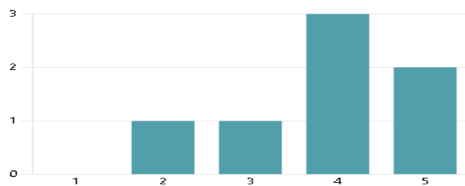

Figure 24: How well were the changes made in the class diagram reflected in the updated prototype? (1-5)
mean: 3.86
$\sigma = 0.99$

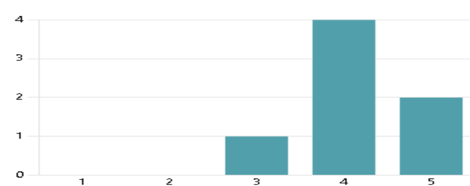

Figure 25: How well were the changes made in the use case diagram reflected in the updated prototype? (1-5)
mean: 4.14
$\sigma = 0.64$

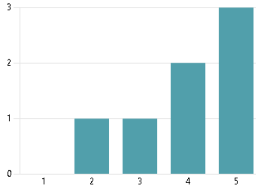

Figure 26: How effective are Fragments?
mean: 4.00
$\sigma = 1.07$

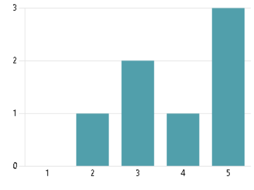

Figure 27: How effective are Categories?
mean: 3.86
$\sigma = 1.12$

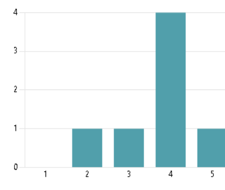

Figure 28: How effective are Pages?
mean: 3.71
$\sigma = 0.88$

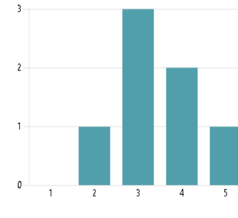

Figure 29: How effective are Section Components?
mean: 3.43
$\sigma = 0.90$

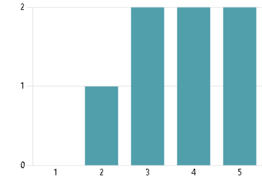

Figure 30: How effective is Styling?
mean: 3.71
$\sigma = 1.03$

.

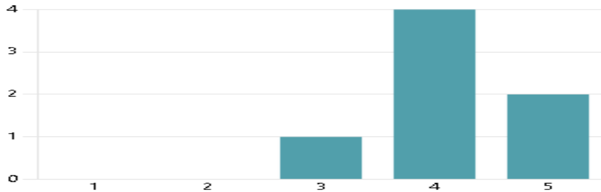

Figure 34: Rate the quality of the default values based on changes in the diagram found in the interface step. (1-5)
mean: 4.14
$\sigma = 0.64$

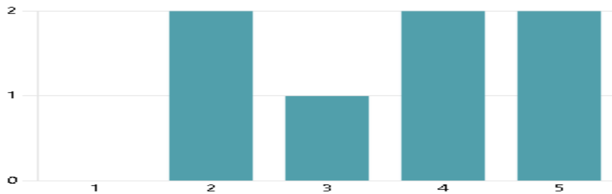

Figure 35: Rate the quality of the generated prototype. (1-5)
mean: 3.57
$\sigma = 1.18$

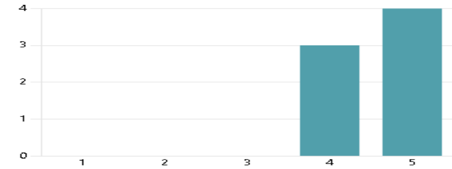

Figure 31: How worthwhile is the addition of support for queries? (1-5)
mean: 4.57
$\sigma = 0.49$

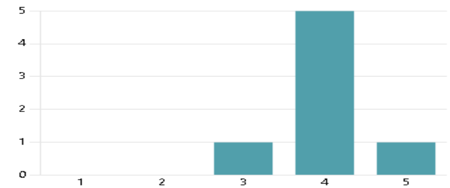

Figure 32: 5: How worthwhile is the addition of support for activity diagrams? (1-5)
mean: 4.00
$\sigma = 0.53$

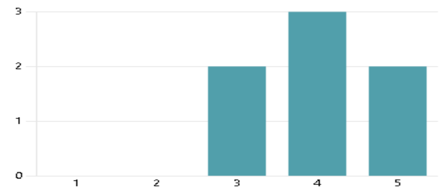

Figure 33: How worthwhile are categories in the sidebar? (1-5)
mean: 4.00
$\sigma = 0.76$