Technical University of Denmark

Matthias Larsen, s103437
Joachim Jensen, s103430

UML

A Graphical User Interface (GUI) for creating UML class diagrams

02350 Windows Programming using C# and .Net, December 2012

DTU Informatics
Department of Informatics and Mathematical Modelling

Matthias Larsen, s103437
Joachim Jensen, s103430

UML

A Graphical User Interface (GUI) for creating UML class diagrams

02350 Windows Programming using C# and .Net, December 2012

UML,A Graphical User Interface (GUI) for creating UML class diagrams

This report was prepared by
Matthias Larsen, s103437
Joachim Jensen, s103430

Supervisors
Bjarne Poulsen

Release date:	December, 2012
Category:	1 (public)
Edition:	First

Department of Informatics and Mathematical Modelling
Technical University of Denmark
Asmussens Alle building 305
DK-2800 Kgs. Lyngby
Denmark

www.imm.dtu.dk
Tel: (+45) 45 25 33 51
Fax: (+45) 45 88 26 73
E-mail: reception@imm.dtu.dk

Abstract

Planning and analyzing specifications when developing software is crucial.

Drawing sequence and class diagrams, writing use cases and tests is all very important parts when working on software products to ensure that the application contains as few bugs as possible and is behaving as intended.

This paper shows the development and considerations throughout the project of an application that has been used to get a better understanding of C# and the .NET platform.

Contents

List of Figures

List of Tables

Listings

1 Introduction

The problem in this project is to analyse and develop a C# .NET application to manage UML class diagrams in a way so that it can be extended with other types of UML diagrams in the future.

This is interesting because different design patterns and methods can be used, and it is important to design a proper architecture meeting the goals of the analysis.

This project is going to focus on usability and extensibility as key components.

Since the prototype demonstration on December 12, the following have been done:

- Save uses a singleton pattern and its own thread
- Text added on relations between classes
- Relations are removed when their respective classes are removed

2
Analysis

A lot of considerations have gone into the development process of the application. The following chapter explains the considerations with the most impact on the application.

TÆLLER 5

2.1 Choice of diagram

While state and sequence diagrams show how (part of) an application works in a detailed way, class diagrams give an overview over all or some classes in the application, what properties and methods they contain and how they are related to each other. In this way, class diagrams can be used to easily see how an application is modelled and thus how it can be altered or extended. Therefore, class diagrams are used more often than state and sequence diagrams when it comes to software architecture, and therefore an application to manage class diagrams has been chosen.

2.1.1 Solution strategy

Due to limited time, not all specifications for a UML class diagram can be implemented in the application, and thus we have made a prioritized specification list, an iteration plan, where the first element is of highest priority and will be designed and implemented first.

1. Different types of classes
2. Undo and redo functionality
3. CRUD for classes
4. Movable classes
5. Relations between classes
6. Save and load functionality
7. Automatic save functionality

One should think that *CRUD for classes* and *Movable classes* is more important than *Undo and redo functionality*, but as they are highly dependent on *Undo and redo functionality*, this has higher priority. E.g. if *CRUD for classes* was made before *Undo and redo functionality*, one could risk having to modify it all again in order to get undo and redo to work properly in the end.

2.1.2 Risks

Because of the solution strategy where a functionality has been given a priority corresponding to its importance, the risks of not implementing one of the specifications due to limited time have been minimized. If *Automatic save functionality* was left out, it would not cause the application to work improperly as it has no impact on the actual purpose - that is, creating an application that can manage UML class diagrams.
This is an advantage of using an iteration plan in the process of software development.

3

Design & Implementation

3.1 GUI

TÆLLER 35

The GUI consists of a menu, a two-column grid and a status bar. The features that can be interacted with in the GUI have been outlined below. An overview of the available class types can be seen on figure 3.1.

3.1.1 Editable classes

There are different gestures for creating, updating and deleting classes, each chosen to get the best user experience. In the left grid column, one can click on a button to create a specific type of class in the application. At the moment there are four types to choose among:

- Class
- Abstract Class
- Enum
- Struct

Clicking one of the buttons will also insert a box in the right grid column representing the class in the application. The color of the boxes are determined by the type of the classes.

Deleting a class is done by right clicking on the specific box representing that class.

If one wants to edit a class, this is done by double (left) clicking on the specific box. This will cause the box to switch to an editable field where one can type in the name, properties and methods for the class in accordance with the syntax of UML class diagrams. Clicking on the same box again or double clicking on another box will save the changes, and the boxes will automatically resize to fit its content. Thus, only one class can be edited at a time.

Choosing inline text edit instead of a pop-up window or likewise is because the UML class diagram content syntax is so simple and understandable and because it

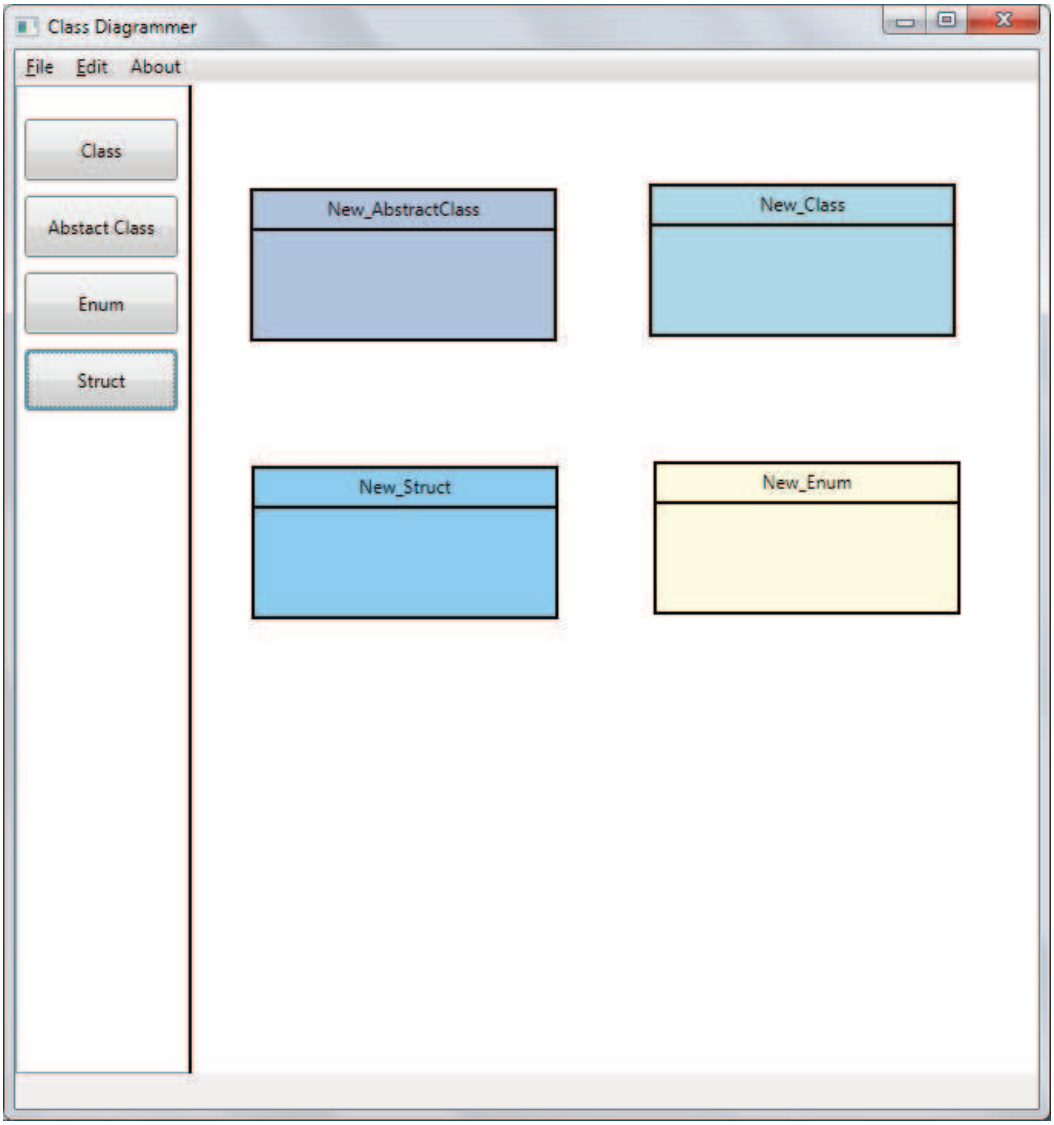

Figure 3.1: Overview of classes

(for a software architect's point of view) is much faster to use this text based UI than pick the different properties and methods etc. in a GUI.

The parsing of inline text edit is done with regular expressions, RegEx, and is described in section 3.1.1.1

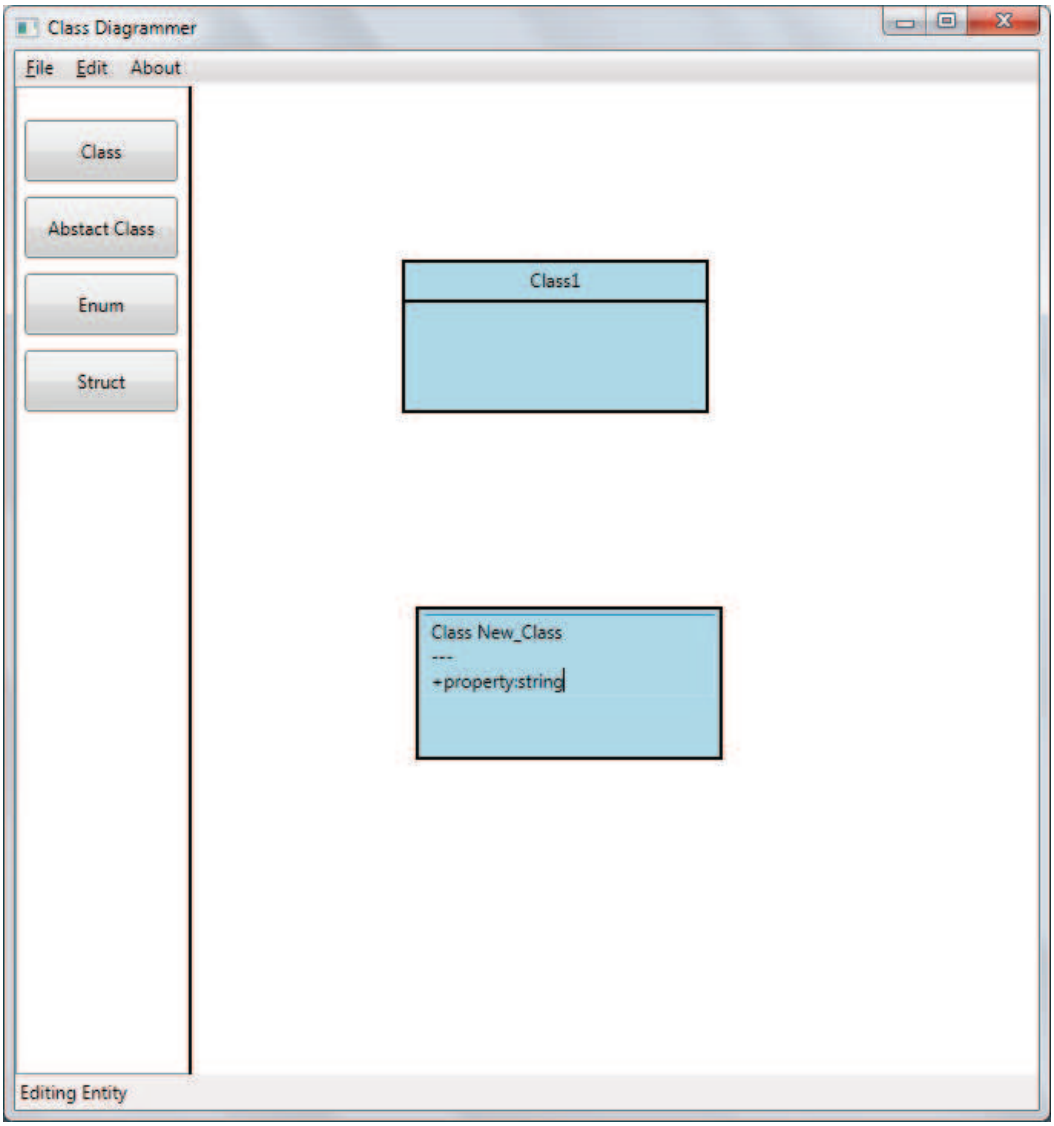

Figure 3.2: Edition of a class

3.1.1.1 RegEx

To validate and parse the text from inline edit regular expressions, or RegEx, is used. RegEx is extremely useful to parse text using defined patterns and extract parts of text.

The RegEx implemented parses the text according to the following pattern:

1. A visibility token[1]
2. A variable name
3. Optional space
4. Optional default value noted by =, optional space followed by the value
5. Optional space if default value specified
6. A colon (:)
7. Optional space
8. The variables type
9. The carnality of the relation

The RegEx parsing is done in multiple steps. The first line is the first one to be parsed against the pattern shown in listings 3.1. When the name and type has been parsed it checks if the second line consists of exactly three dashes "−−−" denoting that the rest of the lines will be either properties of functions of the class.

Listing 3.1: RegEx for class type and name matching

```
(abstract class|class|enum|struct) (.+)
```

Each line is then matched against two patterns to see if its a function or property. If the line matches the function pattern shown in listings 3.2 the parser takes the matched groups and populates a function entry for the current class.

Listing 3.2: RegEx for function matching

```
([#+-])(.+?)\((.+)\)(?: ?: ?(.+))
```

If the line does not match the function pattern, but instead matches the property pattern as shown in listings 3.3 further processing is done otherwise the line is skipped.

If a line has been successfully matched to the property pattern the type group of the pattern is then again matched against the pattern given in listings 3.4 to see if it has been defined as an array.

Listing 3.3: RegEx for property matching

```
([#+-])(.+?)(?: ?= ?(.+?))?(?: ?: ?(.+))
```

Listing 3.4: RegEx for array matching

```
(.+?)\[([0-9]+)\]
```

The property entry is then populated with the matched groups, and if the matched type group is a name of a class already on the canvas a line is drawn between them, and the carnality is written on the line. The carnality of the relationship is taken from the array size if specified otherwise 1 is used.

[1] + for public, - for private and # for protected.

3.1.2 Movable classes

It is possible to freely move the boxes around in the right grid column, but only there. That means that it is not possible to e.g. move the boxes out of the window or into the left grid column.

When editing a box, one cannot move it, but it is still possible to move other boxes.

3.1.3 Undo and redo

The undo-redo functionality has been implemented with the command pattern. Every time there is a change to the item canvas; add, removal or moving of classes the action is put on the undo stack.

When the undo command is then called, the stack is popped, the command is executed thus reversing the action and putting it on the redo stack. The redo command then acts in the same way as the undo command, if there is anything on the stack.

3.1.4 Relations

Relations will be created, updated and deleted automatically based on the classes (boxes) present. Also, a text will say what kind of relation it is, e.g. one-to-one, one-to-many

The relationship text is depending on the information provided for the carnality as described in section 3.1.1.1.

3.1.5 Menu and shortcuts

In the top menu, one can find various options and functions. Each of them has been given a shortcut key combination that correspond to the default shortcut for its type (if it exists). The options and their shortcuts can be seen in table 3.1.

Option	**Shortcut**
New file	Ctrl+N
Open file	Ctrl+O
Save file	Ctrl+S
Save as	Ctrl+A
Undo	Ctrl+Z
Redo	Ctrl+Y

Table 3.1: Shortcuts of menu options

3.1.6 Status bar

On the bottom of the GUI a status bar can be seen. Here, the user gets messages about what the status is, i.e. what is going on in the application. E.g. if a box is

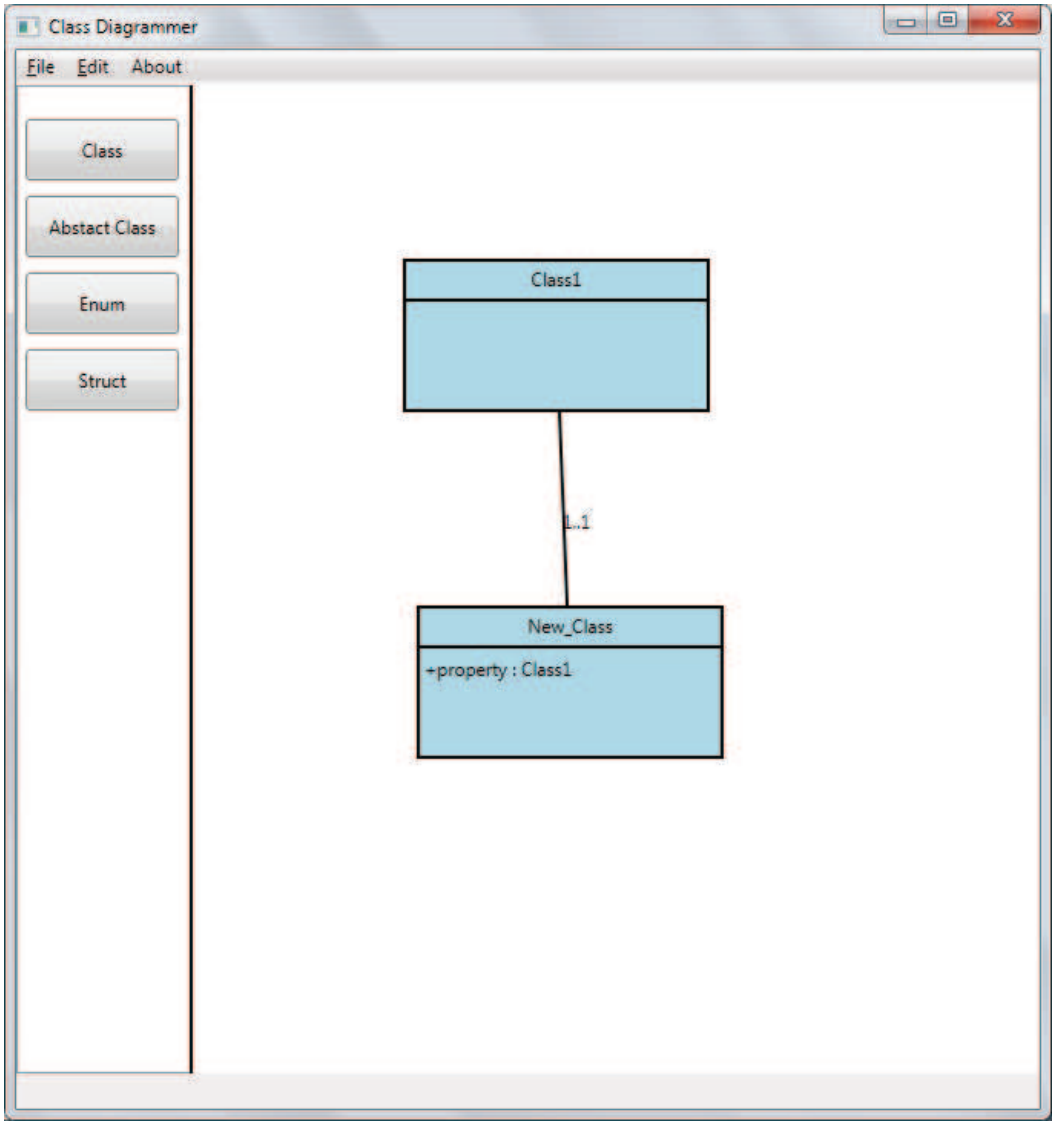

Figure 3.3: One to one relation

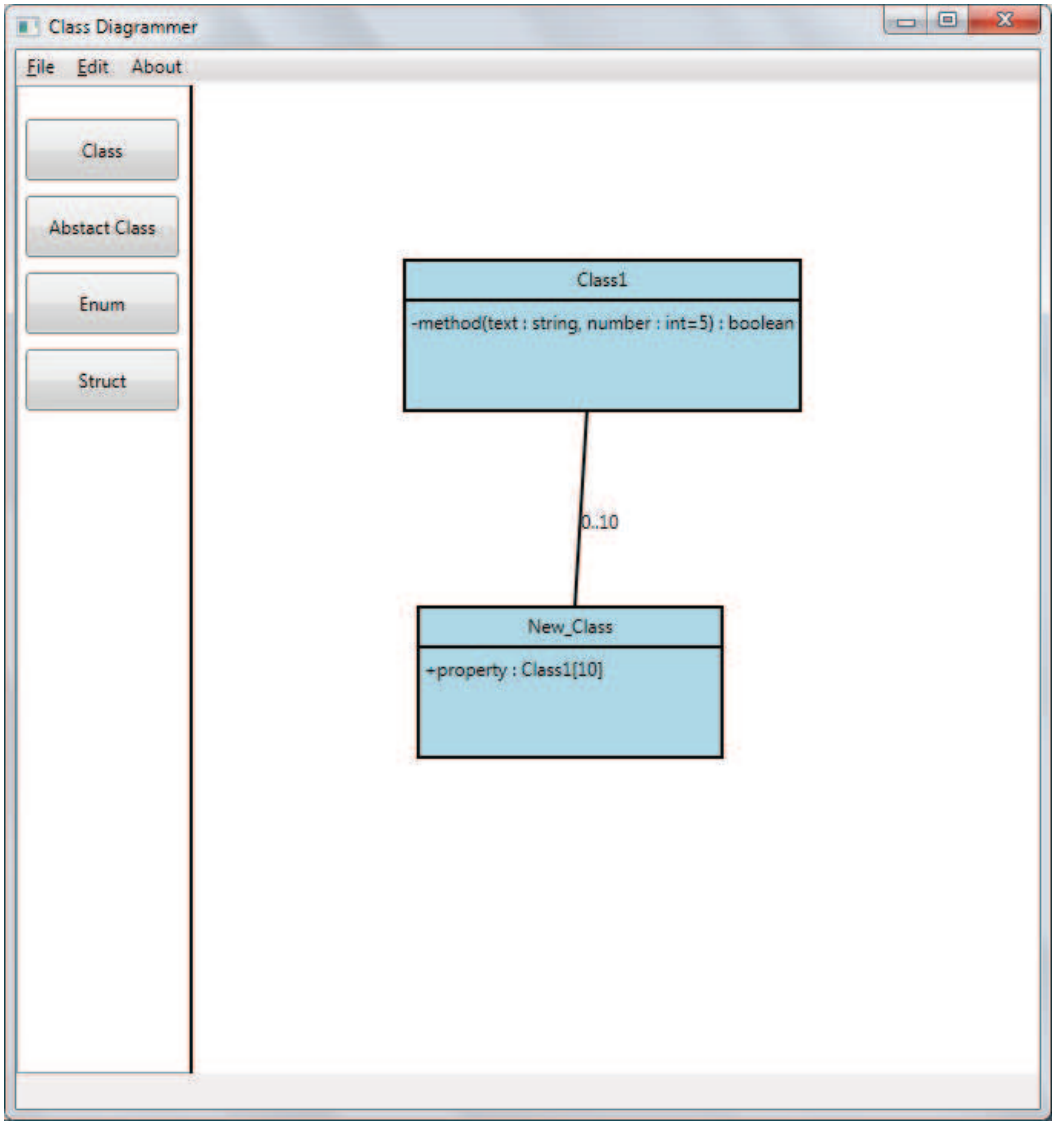

Figure 3.4: One to many relation

being edited, removed, moved or if a file is being (automatically) saved or loaded.

3.2 Application layer

TÆLLER 20

3.2.1 DLL

To ensure the application is extensible all models have been put in a separate DLL file.

This also allows for development of other tools using the same models, for instance a web based UML diagrammer.

3.2.2 Save and load

The save functionality has been implemented in its own thread, because the application could otherwise be halted while this operation were executed. The load functionality remains in the same thread as the GUI, because one should not be able to modify the canvas while loading a file.

A prettier solution would be to use background workers to load the file meanwhile the GUI was disabled or perhaps showing a progress bar. When the background workers had completed loading the GUI was to be enabled again.

After a file has been saved the first time, the application will automatically save it continuously in a specific time interval. This way, the user does not have to worry about losing hours of work if the computer crashes or shuts down.

The application saves the applications state by taking the list of classes on the canvas (List<Base> in the source code) and serializing the entire list with a binary formatter whose output is then saved to a file on the computer.

3.3 Design patterns

3.3.1 Singleton

The singleton design pattern restricts the inistantation of a class to one object. The pattern is useful when exactly one object is needed to coordinate actions or use of data throughout the system, such as a config class or data model.

The singleton pattern is implemented in the Serializer class, that is used when saving and loading files.

The singleton pattern is also used by the MVVM framework GallaSoft for interacting with the MainViewModel to ensure, that it is the same instance of the class that is being used throughout the application, thus the same data.

3.3.2 Command

The command pattern is a behavioural design pattern where an object is used to encapsulate all information needed to call a method at a later time. The command pattern is useful for implementing GUI button, undo/redo, wizards and transactional behavior.

The command pattern has been used to implement the undo-redo feature.

3.3.3 MVVM

The Model View ViewModel is an architechtural design pattern used in software development. It originated from Microsoft and was introduced by Martin Fowler. MVVM is targeted at modern UI development platforms which supports event-driven programming such as Windows Presentation Foundation, Silverlight and HTML5. MVVM is largely based on the Model View Controller, MVC, pattern which strives to separate presentation logic from business logic and data.

MVVM has the same data separation as MVC, as it uses a ViewModel to expose the data from the data model to the view for presentation. The pattern was designed to make use of the data binding functions of WPF, which would facilitate the separation of the view layer development from the rest of the pattern thus removing almost all GUI code from the view layer.

Instead of the user interface developers to write GUI code, it is possible for them to use markup language instead (such as XAML) and create data bindings to the ViewModel, which is made by the application developers.

This seperation allows for higher productivity because application and UX developers are working on different parts of the program, without them having to worry about the other party's code. Even if the application is developed by a single developer, said developer would still benefit from this separation of logic in the late development cycles where the UI often changes frequently based on end-user feedback.

TÆLLER 10

3.4 Tests

Throughout the development of the application several tests has been conducted to ensure that the application is performing as expected.

The various use cases used for testing the application can be found in appendix A.

4 Conclusion

An application to manage UML class diagrams was created successfully. It has been made so that other UML diagrams can be implemented in the future and because all models are saved in a dll, this layer can easily be reused in other applications.

All specifications from the iteration plan REF? have been designed and implemented

The final product ended up being a satisfactorily solution for a UML class diagram application that can manage the content in an intuitive way.

4.1 Future work

Because of the limited time for the project, some specifications were left out. Some features that would have been useful to have in the application are stated below.

4.1.1 Copy and paste classes

Sometimes it would be convenient to copy and paste created classes. As of now it is not possible to copy class boxes, but it is however possible to copy all the content when editing a class and then paste this text into another class. Thus the outcome will indirectly be the same as copying class boxes.

4.1.2 Code scaffold

When creating a class diagram, it would be neat to generate the represented code automatically in a specified programming language such as C#, Java or PHP.

Although this quickly can become quite complex due to the many different syntaxes combined with the application having to cope with each language restrictions such as PHP not having structs or enums while C# and Java does. This was the main reason why this feature got a very low priority thus not making it into the final application.

4.1.3 Export

Having an opportunity to export the class diagram to different formats such as XML, PDF or PNG is definitely a good idea so that a diagram can be used on websites, in presentations or imported by other applications.

A

Use cases

Actors:

- User: Person using the application.

A.1 Create a class

Actor: User
Scenario:

- User left clicks once on Class in the left grid column
- Class *New_Class* is created and shown in the right grid column

A.2 Remove a class

Actor: User
Scenario:

- User right clicks once on a specific class in the right grid column
- The specific class is deleted and removed from the right grid column

A.3 Edit a class

Actor: User
Scenario:

- User left clicks twice on a specific class in the right grid column
- The specific class becomes editable
- User types in some valid content for the class
- User left clicks once on the specific class

- The specific class returns to normal state and changes are saved

Alternative scenario:

- User types in some invalid content for the class
- Invalid content gets ignored on save

A.4 Move a class

Actor: User
Scenario:

- User left clicks once and hold on a specific class in the right grid column
- User drags class around in the right grid column
- User releases mouse button and the new class position is saved

Alternative scenario:

- User tries to move the class out of right grid column
- The class is stopped by the edges

A.5 Save a file

Actor: User
Scenario:

- User goes to File > Save (or clicks Ctrl+S)
- A Save File dialogue pops up
- User specifies a name and clicks Save
- File is saved

Alternative scenario:

- User cancels Save File dialogue
- File is not saved

A.6 Open a file

Actor: User
Scenario:

- User goes to File > Open (or clicks Ctrl+O)
- A Open File dialogue pops up
- User picks a relevant file and clicks Open
- File is opened and content shown in the application

Alternative scenario:

- User cancels Open File dialogue
- File is not opened and current content remains in the application

A.7 New class diagram

Actor: User
Scenario:

- User creates a class
- User moves created class
- User goes to File > New (or clicks Ctrl+N)
- All classes (and undo redo stack) are cleared and the right grid column becomes blank

www.imm.dtu.dk

Department of Informatics and Mathematical Modelling
Technical University of Denmark
Asmussens Alle building 325
DK-2800 Kgs. Lyngby
Denmark
Tel: (+45) 45 25 33 51
Fax: (+45) 45 88 26 73
E-mail: reception@imm.dtu.dk