ViKER: A Visual Interface for Transformations Between EER and AR Conceptual Models

|  |  |
| --- | --- |
| St John Grimbly  UCT  Department of Computer Science  GRMSTJ001@myuct.ac.za | Jeremy du Plessis  UCT  Department of Computer Science  DPLJER001@myuct.ac.za |
| Gabriel Stein  UCT  Department of Computer Science  STNGAB004@myuct.ac.za | |

This paper investigated the… We found that…This is useful because…

# Introduction

Traditional data management modelling procedures involves designing a sequence of requirements in the form of a conceptual model such as an entity relational model (ERM). This conceptual model is then transformed into a relational model (RM), which is more representative of the actual implementation of the database. This database is then created and put into production.

Interesting ideas for reusing the valuable information presented in the conceptual models have arisen in the past, including using conceptual models to ‘query-by-design’ using UML-like notation.

Open world…closed world…

This project investigates methods of transforming between EER and ARM by implementing the theoretical transformation rules outlined in the KnowID paper (incl. references). Focus is placed on implementing clean code for future extension by other teams, properly documenting code and implemting a user friendly interface for easily performing transformations implemented in this paper.

# Theory

The KnowID (insert reference) paper introduces a new transformation procedure for converting between abstract relational models (ARM) and enhanced entity relational models (EER) models. ARM is an extension of RM presented in reference.

ARM extends traditional RM diagrams by including abstract datatypes - OID - which act as memory references to other relations in the model. Each relation is assigned a ‘self’ reference which uniquely identifies it. Attributes of the relation are thus not primary keys and we can be assured that every relation has a primary key – self. Attributes can be identifiers, forming part of a path functional dependency. This pfd->self.

EER extends traditional ER diagrams by including notation for specialisation, partitioning, generalisation and aggregation relationships. It is thus a more representative conceptual model for the underlying database structure and makes ideas such as ‘query-by-design’ easier to implement (reference).

# Requirements Captured

**Back-end**

**Front-end – for Gabriel**

The next section deals with the analysis of your system. Cover the functional, non-functional and usability requirements. This is where you present your use case narratives and diagrams.

Discuss the major analysis artefacts that you produced. We will expect you to produce at least one overall description of the architecture used in your system as a diagram, either here or below (see Section 2.3). You may also want to include an analysis class hierarchy diagram.

# Design Overview

**Data Flow**

A screenshot of a social media post

Description automatically generated

**Back-end**

**Front-end**

The next section is an overview of your design. The system design has to be justified in terms of the expected behaviour of the final product.

If you produced a design class diagram put it here.

You must present the overall architecture of the system together with an architecture diagram. You may choose what kind of diagram best suits your project, but we would expect a layered architecture diagram (see Figure 1) unless there is a good reason for some other kind of diagram. It need not be a formal UML diagram as long as it conveys all the necessary information clearly.

You should then (in subsections) cover the algorithms and the data organisation used and why they were considered the best.

# Implementation

Jeremy+St John Back-end Implementation

Gabriel Front-end Implementation

A screenshot of a cell phone

Description automatically generated

* Describe your data structures and be sure to illustrate them with a diagram.
* If your user interface was a key feature describe how that was implemented.

**Figure 1:** An architecture diagram. Caption to go below figure.   
Note that LibreOffice handles this better than MS word

* Discuss the function of the most significant methods in each class. This may well require flowcharts, or sequence diagrams, in some cases.
* Any special relationship between the classes (e.g. friends) and why they exist.
* A description of any special programming techniques or libraries used.

# Program Validation and Verification

Validation and verification were taken seriously throughout the development plan. During the design phase of the software, we decided to take a test-driven development approach. That is, we would create our critical and functional tests for the software and test our progress against these test cases throughout the development cycles. Test cases were added as needed when a unplanned or unintended task needed to be tested.

Our software consists of two major branches – the front-end and the back-end. Different approaches were taken in the testing of these. User friendliness and user interaction was the top priority in the design of the the interface – discussed in the section. Accuracy of transformation and correct error reporting was the priority for the back-end.

Table 1: Summary Testing Plan.

|  |  |
| --- | --- |
| Process | Technique |
| 1. Class Testing: test methods and state behaviour of classes | Random, Partition and White-Box Tests |
| 1. Integration Testing: test the interaction of sets of classes | Random and Behavioural Testing |
| 1. Validation Testing: test whether customer requirements are satisfied | Use-case based black box and Acceptance tests |
| 1. System Testing: test the behaviour of the system as part of a larger environment | Recovery, security, stress and performance tests |

Table 2: Summary of tests carried out. A table caption goes above the table.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Set and reason for its choice | Test Cases | | |
| Normal Functioning | Extreme boundary cases | Invalid Data (program should not crash) |
| Preliminary test (see Appendix 3) | Passed | n/a | Fell over |
| Development Unit Testing |  |  |  |
| Final Testing |  |  |  |
| UX/UI User Testing |  |  |  |

Tell us how you tested the system and why you believe it works. Describe the Quality Management Plan for your project, that is, software testing plan. The plan should indicate the types of testing that was performed and detail how they were done. This must include the reasons on why the chosen testing protocol was considered effective.

Create a table that summarizes the testing plan (see Table 1).

Table 1: Summary Testing Plan. A table caption goes above the table.

|  |  |
| --- | --- |
| Process | Technique |
| 1. Class Testing: test methods and state behaviour of classes | Random, Partition and White-Box Tests |
| 1. Integration Testing: test the interaction of sets of classes | Random and Behavioural Testing |
| 1. Validation Testing: test whether customer requirements are satisfied | Use-case based black box and Acceptance tests |
| 1. System Testing: test the behaviour of the system as part of a larger environment | Recovery, security, stress and performance tests |

Describe all the steps taken to validate the correctness of the program.

If you had user tests then say what you did and what the results were. Describe why these test data were chosen (what test conditions the data was testing). Table 2 provides an example of the sorts of results we are looking for. The full detail of the test runs should be appended to the report.

Table 2: Summary of tests carried out. A table caption goes above the table.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Set and reason for its choice | Test Cases | | |
| Normal Functioning | Extreme boundary cases | Invalid Data (program should not crash) |
| Preliminary test (see Appendix 3) | Passed | n/a | Fell over |
|  |  |  |  |
|  |  |  |  |

Follow your table of results with a discussions of them highlighting how useful and usable your system is for its intended purpose.

# Conclusions

Your report must have a clear conclusion where you revisit the aims set out in the beginning and discuss how well you met them. Did you achieve the objective of creating a well-structured, modular, and robust system? Please summarize the design features and test results that show this.

# Appendix A — Code Legibility and Output

Include user manual

This is not strictly part of the report but is a requirement for the final hand-in.

* Each method should start wide a brief description of its function.
* Use indentation to display the structure within a method.
* Comments should be used extensively. They are best used to describe logical blocks of code rather than individual statements. Line-by-line comments have the drawbacks of not providing any overview and of decreasing readability.
* Meaningful identifiers should be chosen.
* Output should be pleasingly formatted and easy to read.

# References