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

St John Grimlby
UCT, Dept of Computer Science
GRMSTJ001@myuct.ac.za

Gabriel Stein
UCT, Dept of Computer Science
STNGAB004@myuct.ac.za

Jeremy du Plessis
UCT, Dept of Computer Science
DPLJER001@myuct.ac.za

10 September 2019

This paper explores the investigation into methods of transforming between EER and ARM by implementing the theoretical transformation rules outlined in the KnowID paper [1]. The visual interface developed for use by experts and non-expert users is discussed. The requirements captured are discussed and a detailed discussion on the design and implementation of our software is presented. Finally, the validation and verification methods are discussed. Conclusions and recommendations are then presented. It was found that our software is extremely intuitive in comparison to traditional modelling procedures. Non-expert users found the UI and UX intuitive and easy to learn. Our core requirements were thus met.

Contents

1. Introduction

Traditional data management involves satisfying a sequence of analysis and design requirements ultimately resulting in a database schema and an implementation of a physical database, in order that the data may be queried and analysed. A key requirement in the process of data management and database design is the building of conceptual models, such as Entity Relational Models (ERM), which are used for describing the entities in the problem domain and how they are related to one another. An ERM may subsequently be transformed into an Abstract Relational Model (ARM) - which is an extension of the well known Relational Model (RM) - which essentially defines the database schema. The database is then created using the schema and put into production, where after the conceptual model is typically discarded (or kept on file, but never used for any further practical purpose). However, it has been suggested that the conceptual, visual representation of the database captured in a conceptual model such an ERM may hold value for tasks beyond database design, particularly with respect to user interaction. Specifically, Keet and Fillottrani [1] have recently explored using conceptual models in the form of a graphical interface to facilitate processes of interaction with, and querying of, the underlying database structure. Such techniques, which are based on what is called the open world assumption in the study of Ontology (more on this in the Theory section below), would make understanding and interfacing with the database much easier for non-expert users.

The purpose of the project described in this report was to design and build the foundational software tools to allow non-expert database users to easily view and transform between Abstract Relational Models (ARMs) and Extended Entity Relational (EER) models - the ARM is an extension of the RM and will be explained further on in the report. More specifically, this project investigates methods of transforming between EER and ARM by implementing the theoretical transformation rules outlined in the KnowID paper [1]. Focus was placed on implementing clean code for future extension, as well as the design and implementation of a user-friendly interface for easily displaying and interacting with the implemented transformations.

The systematic approach to solving the problem was as follows. We began with foundational research on ontology, EER models, ARMs, and the rules for transforming between them. We then conducted a traditional analysis of the project, beginning with the design of use cases and test cases, and ending with a high-level architectural design of the system. During the implementation phase, we focused initially on designing and creating the formats for the JSON data structures which we would use to represent the ER and AR models in textual form, followed by the OOP representation classes, methods and attributes of the models. We then created the JSON files for all our test cases and began the process of developing the rest of the system according to test-driven development philosophy. Our goal was to develop a basic evolutionary prototype which would handle the easier test cases, and iteratively develop adding new functionality for each progressively more complicated test case, thereafter, refactoring in real time. The graphical user interface, written in JavaScript, was developed in parallel to the back-end portion of the system, written in Python, and the two were connected towards the end of the implementation phase; communicating via a flask server using the JSON data structures to pass information between them.

What follows in the report is a high-level overview of the theory behind the development of the software, followed by a summary of the client requirements and how they have been captured, thereafter an overview of the design and implementation is outlined along with a summary of the system functionality, ending with a description of the different testing mechanisms that were used to validate and verify the systems performance.

2. Theory

The KnowID [1] paper proposes a new transformation procedure for converting between abstract relational models (ARM) and enhanced entity relational models (EER) models, and back. This transformation procedure is intended as an extension to an already-existing pipeline of transformations which essentially utilise the structure the ARM to allow for the interpretation and execution of Sequel Path Queries (SQLP). The physical implementation of the procedure would further extend the pipeline by adding a conceptual layer on top. This should theoretically allow for the development of software which will enable users to interact with and query an underlying database structure through a graphical interface by masking use of the conceptual model initially used to model the schema.

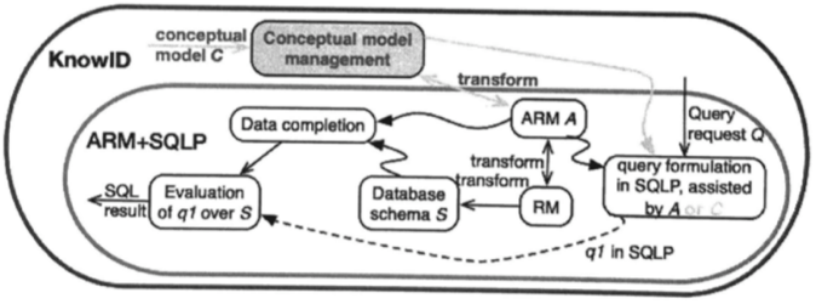


Figure 1: Diagram showing how KnowID extends traditional modelling methods.

In the study of Ontology, this approach to database access and design is rooted in the open-world assumption is that the assumption that a statement cannot be known to be false unless it is explicitly stated as such. It is the exact opposite of the closed-world assumption. The closed world assumption is traditionally assumed for database. The extension of traditional models to ARM is significant as it can operate fully within a closed-world assumption, similar to the underlying database. Keeping in mind the overall objective described above, this project is concerned with the software implementation of the EER ¡-¿ ARM transformation. Below is a description of each of the models, along with examples.

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 [1]. An example of an EER is shown in figure 2.

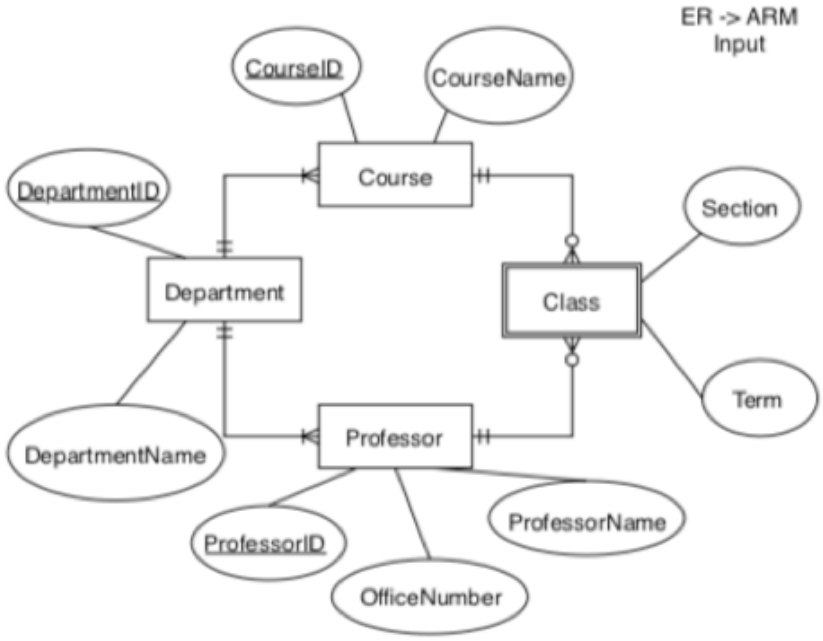


Figure 2: EER model example as used in this project.

The ARM structure extends the traditional the RM structure 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. Identifier attributes belonging to the entity are thus not mapped to the relation as primary keys, but rather the combination of a set of identifier attributes A1,,An belonging to a relation form a path functional dependency for that relation, which points to the abstract self (pathfd(A1,,An)-¿self) which essentially serves as the primary key for the relation. This mechanism allows for the inference of several constraints pertaining to the disjoint-ness or the covering between relations in the database, as well as supporting the SQLP functionality. An example of an ARM is shown below in figure 3.

```
table department(
    (self OID, DepartmentID String, DeparmentName String),
    primary key (self),
    pathfd (DepartmentID) -> self
    disjoint with (course, class, professor)
    )

table professor (
    (self OID, ProfessorID int, ProfessorName String, OfficeNumber int, DepartmentID OID),
    primary key (self),
    constraint dept foreign key (DepartmentID) references department
    pathfd (ProfessorID) -> self
    disjoint with (course, class, department)
    )

table course (
    (self OID, CourseID int, CourseName String, DepartmentID OID),
    primary key (self),
    constraint dept foreign key (DepartmentID) references department
    pathfd (CourseID) -> self
    disjoint with (professor, class, department)
    )

table class (
    (self OID, Selection String, Term String, CourseID OID, ProfessorID OID),
    primary key (self),
    constraint prof foreign key (ProfessorID) references professor
    constraint course foreign key (CourseID) references course
    pathfd (CourseID, ProfessorID) -> self
    disjoint with (professor, course, department)
    )
```

Figure 3: ARM textual representation example, as used in this project.

For this report we have designed a graphical representation of the ARM, that is intended to be used as a summary of the underlying model to be used in the graphical interface. The same textual example as above may be represented in this form as in figure 4.

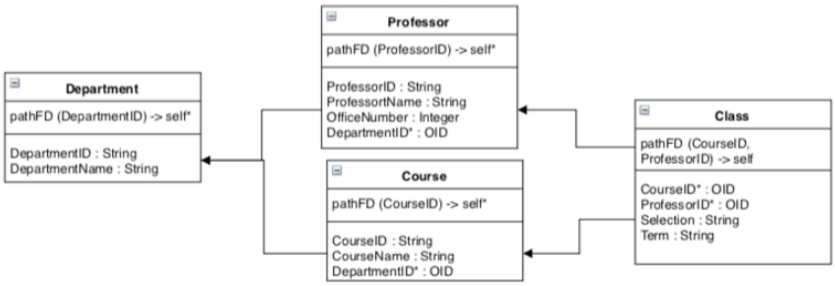


Figure 4: ARM visual representation example, as used in this project.

3. Requirements Captured

The high-level scope of the project was dictated by the client as a list of requirements for the system. These formed the basis for the creation of the artefacts during the analysis stage of the project as well as the functional, non-functional and usability requirements.

3.1. Functional Requirements

1. Implement the rules as in the KnowID paper, both the EER to ARM and ARM to EER.
2. Report on success/failure of a transformation.
3. Build a user interface to manage and control the transformations.
4. Open/save models.
5. Report on those things that could not be transformed.
6. Report on what happened with each element.

3.2. Non-functional Requirements

1. Permissible to extend a current open source EER tool.
2. Textual representation (perhaps) with XML.
3. Suitable performance (quick transformations).

3.3. Usability Requirements

1. Intuitive front-end design.

Ultimately, we feel all the requirements specified by the client have been satisfied. Reporting on what happens with each element in a transformation was not done. We did not implement this as we feel the graphical representation of the models (as seen in section 2) will provide visual information to the end user which should give them an intuition about the transformation of specific elements from one model to another. Additionally, creating a text-based format for describing the transformation of individual entities would clutter the user interface and we feel that it would degrade the user experience, as the interaction with both the models are meant to be as graphical as possible.

3.4. Analysis Artefacts

Using the requirements dictated by the client, the following artefacts were produced as part of the analysis and ultimately used to develop the software:

1. Use case narratives
2. Use case diagram (from use case narratives)
3. System sequence diagram

3.4.1. Use Case Narratives

At a user level, the system is fairly simple, and the functional requirements can be distilled into 5 use cases as described below.

1. **Convert ER model to AR model**
   The user (Primary Actor) will create a JSON representation of an EER (such as the many examples provided in our user manual) and then upload the file to our program by clicking the Load Model button. They will then click the Transform button and the system will send the EER JSON to the server and get back an ARM JSON representation, the front-end will then render an abstract

relationship (AR) model. The one alternative flow to a direct transformation, is if the user created an ER model that cannot be transformed into an AR model due to not fitting the transformation rules. In that case, the error will be reported in the error log below the action area. Another alternative flow is if the user created an ER model that can only be partially transformed into an AR model in which case whatever transformation cannot be performed, will be outputted in the error log below.

2. **Convert AR model to ER model**
   The user (Primary Actor) will create a JSON representation of an ARM (such as the many examples provided in our user manual) and then upload the file to our program by clicking the Load Model button. They will then click the Transform button and the system will send the ARM JSON to the server and get back an EER JSON representation, the front-end will then render an EER. The one alternative flow to a direct transformation, is if the user created an AR model that cannot be transformed into an ER model due to not fitting the transformation rules. In that case, the error will be reported in the error log below the action area. Another alternative flow is if the user created an AR model that can only be partially transformed into an ER model in which case whatever transformation cannot be performed, will be outputted in the error log below.

3. **Save transformation report (transformed model and error log)**
   Once the user (Primary Actor) performs the relevant transformation, they click the Save Transformation Report button and the system will download the file to the users computer. This allows the user to load the model when they come back onto the system at a later stage, as well as view the error log. The only alternative flow is if the transformation cannot be performed, as the system will not allow the user to save a model that cannot be rendered.

4. **Load model**
   If the user (Primary Actor) wants to load a model (can a new model or previously transformed model), they click the Load Model button and select a model (represented as a JSON file) from the file browser that they want to load. There are no alternative flows.

5. **View error log**
   When the user (Primary Actor) performs a transformation, the success or failure of the transformation will be outputted in the error log that is located at the bottom of the interface. If the transformation is completed successfully and all transformation can be performed, the log will output Transformation Successful. If only some of the transformations can be performed, the system will output which transformation cannot be performed and why not. If none of the transformations can be performed, the log will output all the transformations that cannot be performed and why not. There is also detailed information on what information is lost during transformations.

3.4.2. Use Case Diagram

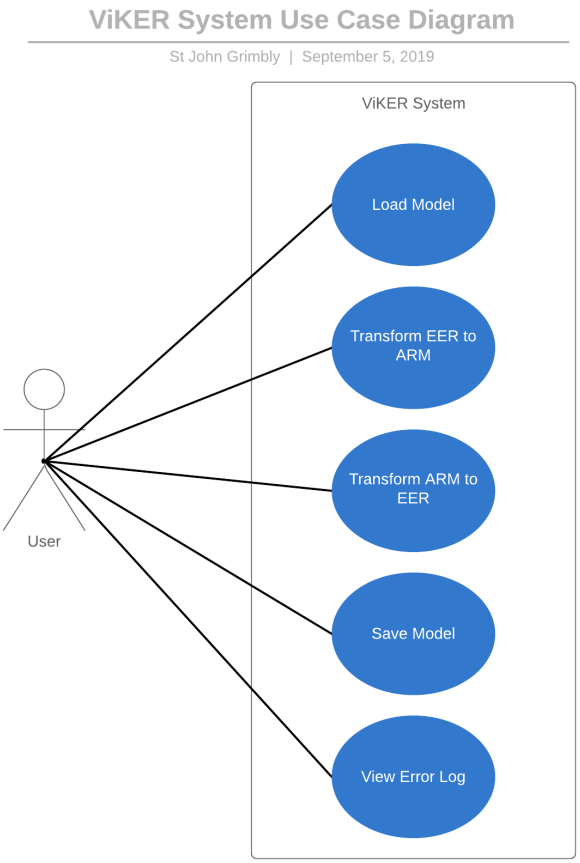


Figure 5: UML use case diagram depicting the main use cases developed for in this project.

3.4.3. System Sequence Diagram

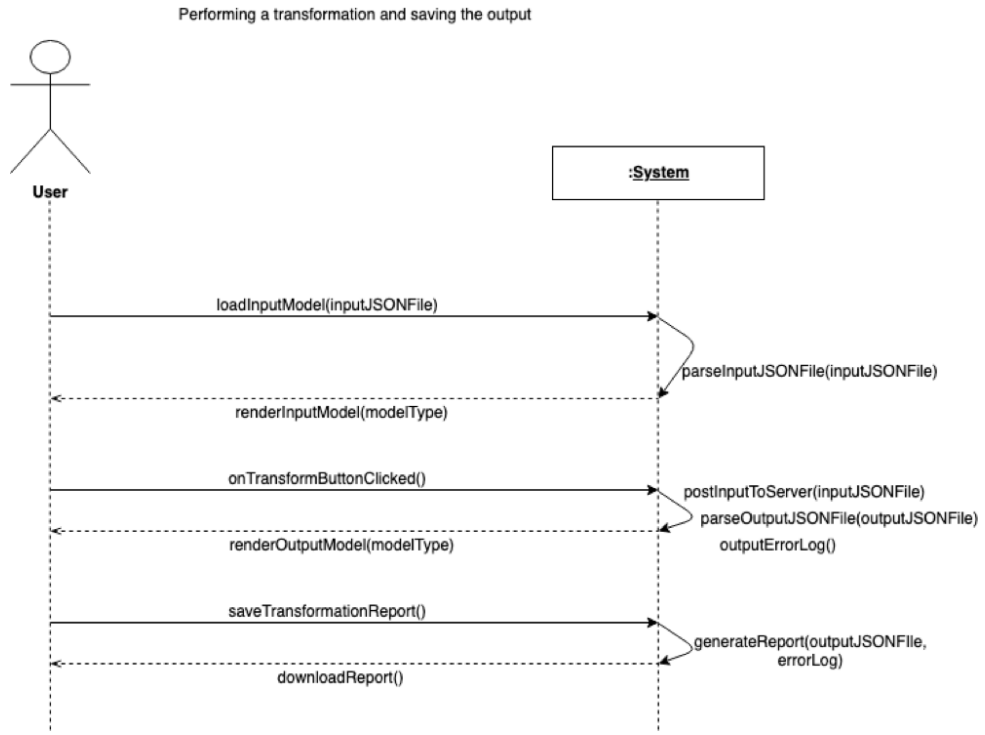


Figure 6: Diagram showing the interaction with the user and the system. This is shown as a UML system sequence diagram.

The user loads their JSON representation of either an ARM or EER using the LoadInputModel() method which is called by pressing the Load Model button and selecting a JSON file. The system then parses the JSON into objects that are then rendered as a model in the input section of the user interface. The user will then transform the model by clicking the Transform Model button which will send a server request to perform the transformation logic and parse the output JSON representation of the transformed model. The system then renders this outputted JSON as a model in the output section of the user interface. The system will also print out the error report in the error log on the user interface, this details whether the transformation was successful or not as well as reports on the data that was lost between transformations. The user can then decide to download the full transformation report, which includes a JSON representation of the output model, as well as the entire error report. The report will be downloaded onto the users computer.

3.5. Requirements Captured

3.5.1. Requirements Captured in Back-end

Special focus was placed on capturing the key requirements of the project. Careful reading of the theoretical discussions of both ARM and EER was done before any design was started. This led to a smooth development process in which all the core requirements set by the client were captured. The captured requirements of the backend are as in the following form:

1. The rules of the transformations have been captured in the EERToARM and ARMToEER classes in which OOP representations of the respective models are transformed. The OOP representations are then transformed into JSON representations in the ReadWriteARM and ReadWriteEER classes.

2. The status of the transformations (success/failure) and information about the transformations are captured during the transformation process in the EERToARM and ARMToEER classes. This is

captured as a python dictionary and appended to the JSON representation of the models in the WebServer.

3. The user interface requirement is discussed in detail in the next section. This project created an intuitive UI far beyond the scope of the hard requirements.

4. The open/save functionality was implemented on the front-end. See next section.

3.5.2. Requirements Captured in Front-end

The front-end was, though not a hard requirement of the client, a key focus in our design process. The client specified that a major reason for this transformation procedure was to allow conceptual models of the database to be useful beyond the design phase. It was thus decided that a user interface would be the most user-friendly experience for an end user, as having to write text based, complex models would be difficult and cumbersome. The features of the project captured on the front-end are as follows:

1. A user can load a JSON model by clicking the Load Model button on the web application GUI. This logic is performed in the app class that creates an ERModel or ARModel object to parse the JSON and manipulate it in order to render a graph.

2. The model graph is then rendered in the input section via the EntityGraphModel class.

3. The model can then be transformed from to EER/ARM by clicking the Transform Model. Which makes a POST request to our python webserver and gets the transformed model back as JSON.

4. The success or failure of the transformation is reported in a dedicated error log section of the GUI which can later be downloaded by clicking the Save Transformation Report button.

5. Detailed information about information lost in the transformation procedure is displayed in the error log. The newly created model can be saved to a JSON representation by clicking the Save Transformation Report button.

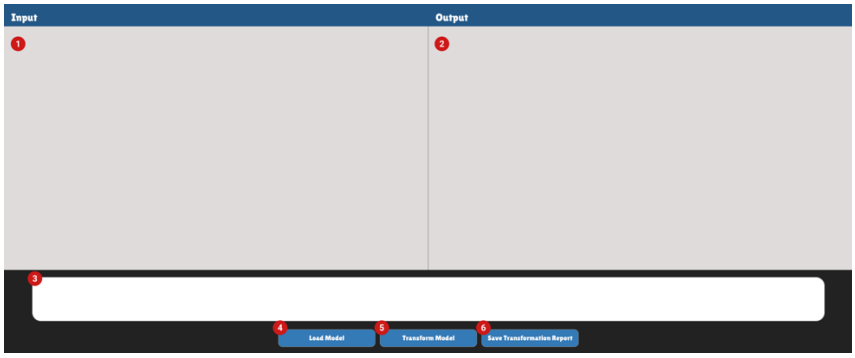


Figure 7: Front-end user interface for interaction with the system developed in this project. Discussion on each of the requirements satisfied and features present is presented below.

1. Where the input model is rendered.

2. Where the output model is rendered.

3. Where the error log is printed.

4. Button to load model as a JSON file.

5. Button to send input model to server and transform to output model.

6. Saves the output model as a JSON file along with the error log.

4. Design Overview

4.1. User Interaction and Data Flow

The diagrams below show a high-level overview of the system for both the class interaction and data flow. These diagrams capture the requirements and user functionality as in the systems sequence diagram in figure 6.

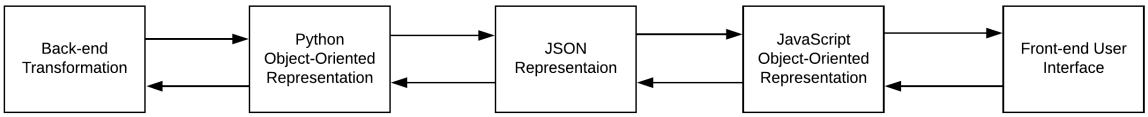


Figure 8: Diagram showing how data flows in the architecture of this project.

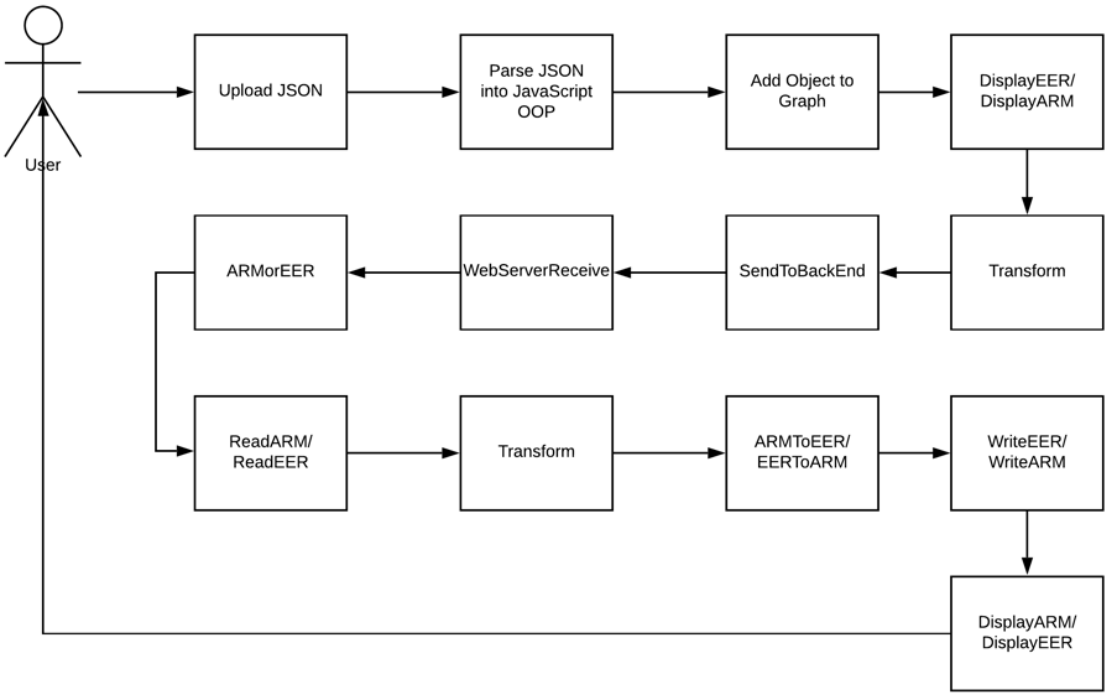


Figure 9: Diagram showing how the program flows with user interaction in the architecture of this project.

4.2. Architecture Design Overview

4.2.1. Back-end Architecture Design

In designing the overall architecture of the back end we aimed for modularity (high cohesion) and low coupling. We started with the broad functional requirements and broke the large tasks up into smaller ones, each handled by a different class in the system, as pictured in Figure 10. A web server module will receive calls via an API from the front end system along with a JSON file containing the JSON representation of either the AR or EER model. The JSON file is the parsed by a separate JSON Parsing module which is in charge of reading the file and converting the model into an object representation in memory, returning either an array of entities or relations to a module which contains all the transformation logic. Inside the logic module, there are sub modules which will perform the EER to ARM and ARM to EER transformations, and keep a log of the errors and auxiliary information produced during the transformations. The output of the logic module is also an array of objects, which will be returned to

the Parsing module, which will in turn turn the objects into JSON files and deliver them back to the web server module and finally to the front end through the API. Consideration was also made with regards to an integrated unit testing module, which was built into the back end too. All the back end code was written in Python, the Numpy and Enum libraries were used in the logic module, the JSON library was used in the parsing module and the Flask library was used in the web server module.

4.2.2. Front-end Architecture Design

The front-end was developed in JavaScript with React. The user interface has the form of a modern React website. The function of the front-end is to provide an easy to use UI for expert or non-expert users to interact with our back-end system developed for this project. As discussed in section 3, the front-end's main requirements are to allow loading of models, transformation of models, saving of models as well as error reporting on the success and failure of transformation. These requirements are all well captured in our system, as shown in figure 10. A user can click on the 'Load Model' button to load their EER or ARM model. This model is rendered and displayed. The user can interact with this model by moving objects around. The user can then transform the model by clicking the 'Transform Model' button. The output model is automatically rendered. An error log is also displayed. By clicking the 'Save Transformation Report' button, the user can download the output, transformed, model with the transformation log in the form of a JSON file.

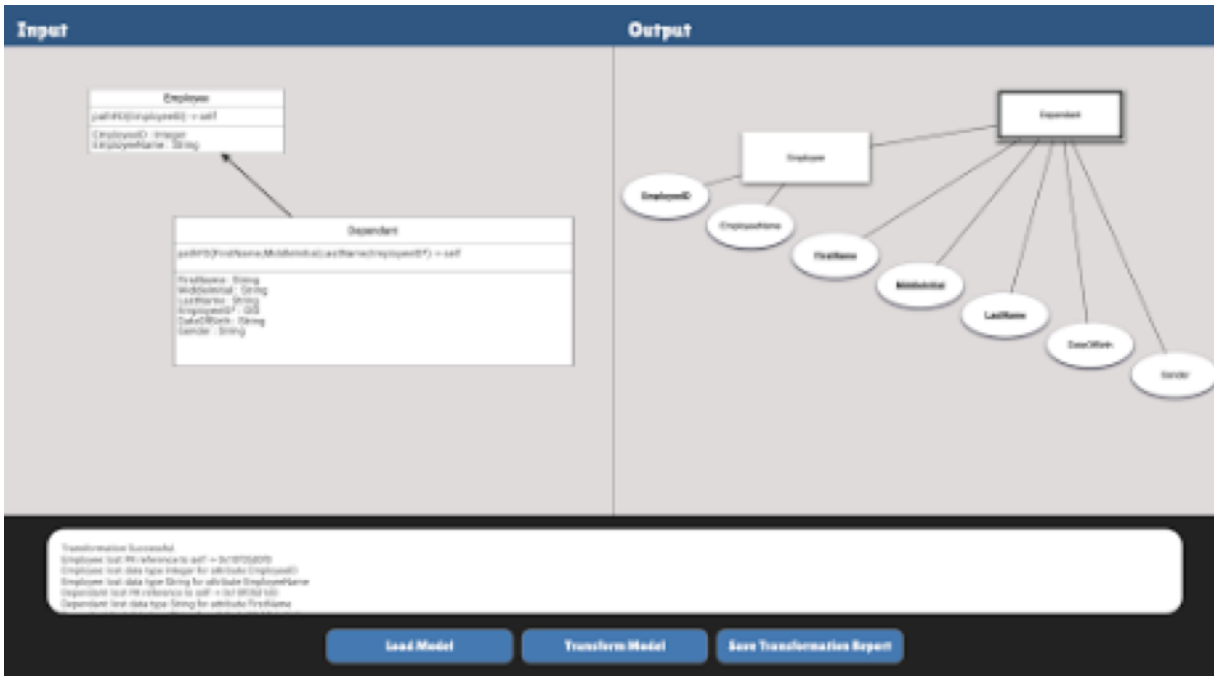


Figure 10: User interface with a model rendered and transformed.

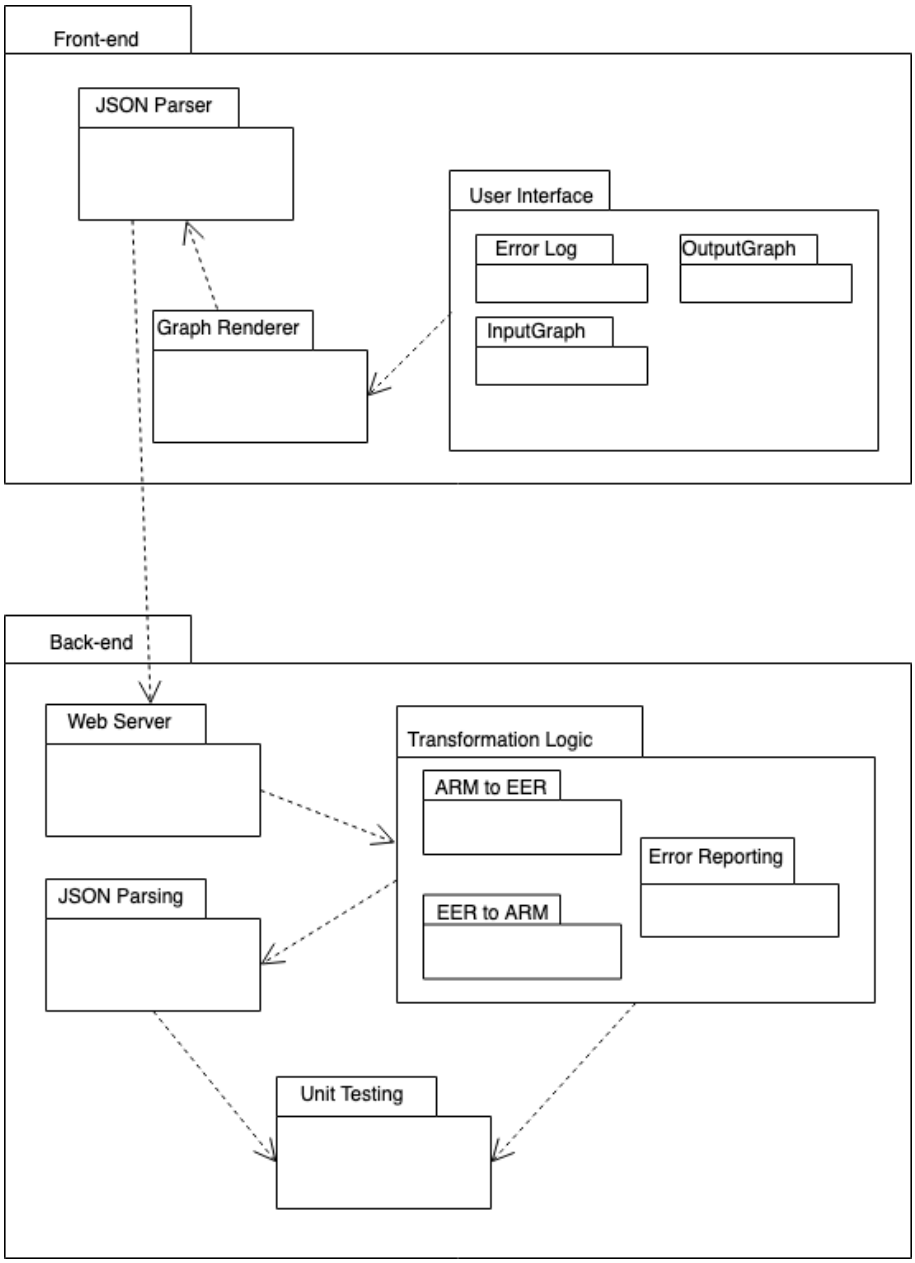


Figure 11: Diagram showing the architecture of the system in the form of a package diagram.

5. Implementation

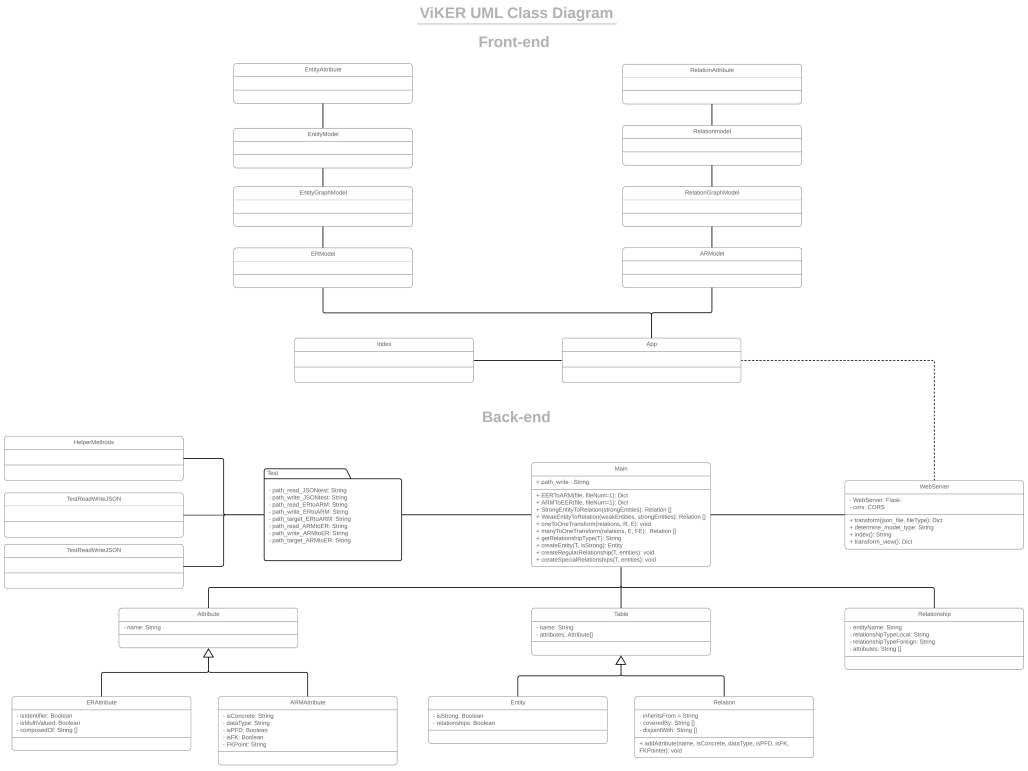


Figure 12: UML diagram showing both the back-end and front-end class structure.

5.1. Architecture Implementation

The key vision in the design of the system architecture as a whole was to ensure separation of concerns. That is, logic is separated from communication and storage of data. We designed our classes such that the principles of low coupling and high cohesion were followed. To elaborate, we ensured that the elements within a specific class belong together in that they are closely correlated in their function or dependency. Further, we ensured that classes themselves are not dependent on each other. They are independent entities in their design.
The system was developed in a layered architecture. The fundamental logic of all the transformations were developed in the back-end. The UI was layered above this, interacting through web sockets in a flask and node server.
The UML class diagram in figure 12 shows the class structure of both the front-end and back-end. Inheritance was a key in the choice of structure as both EER and ARM naturally share many fundamental concepts such as attributes.
Classes were designed for attributes, relationships and tables. Specific attributes ER and ARM derive from the attribute class. ER entities and ARM relations derive from the table class.

The Main class was responsible for executing transformation logic, calling methods from classes and creating objects as needed. Python Lists, numpy arrays and python dictionaries were the main data structures used throughout the back-end transformation process.

5.2. Classes

5.2.1. Back-end Classes

1. **Main**
   The main class is responsible for coordinating the transformation logic by calling the appropriate

transformation procedures and storing the entities/relations and error log.

2. **EERToARM**
   The EERToARM class is responsible for performing the actual transformation from EER to ARM by manipulating the OOP representation of the EER model.

3. **ARMToEER**
   As in the EERToARM case, the ARMToEER class is responsible for the transformation of ARM OOP model to EER OOP model.

4. **Constants**
   The Constants class is the class in which the enumerated types are defined for our OOP representations of ARM and EER.

5. **Table**
   The Table class is the parent class of Relation and Entity. These form the fundamental structures of both ARM and EER models respectively. Relation and Entity inherit from parent as they share concepts of attributes and specific attribute properties as well as Table names.

6. **Attribute**
   The Attribute class is the parent class of the ERAttribute and ARMAttribute. As mentioned in the discussion about the Table class, EER and ARM models share the concept of attributes. The attributes themselves do differ in their properties.

7. **Relationship**
   The Relationship class is responsible for storing all the appropriate information about a relationship between entities in an EER diagram. Relationships between Relations in ARM can be dynamically determined based on the self* and foreign keys specified in the relation. In EER, however, one cannot dynamically determine the nature of the relationship between entities. For example, an ExactlyOne to OneToMany relationship provides specific information about the nature of the relationship between the local and foreign entities.

8. **Test (Package)**
   The Test package is responsible for collating all the unit testing and general testing, as well as helper methods required for testing. ReadWriteJSON, TestEERToARM and TestARMToEER are the classes in this package. The class names are fairly self explanatory.

9. **WebServer**
   The WebServer class is the class that allows communication between the front-end visual interface and the back-end transformation logic and associated error reporting. The WebServer is implemented using the Python Flask package. It also makes use of the Python JSON package for manipulating JSON as needed.
   The WebServer creates a server endpoint on 0.0.0.0:5000/api/transform. This provides both POST and GET procedures. The front-end can send a JSON file to that end-point, at which point the WebServer will determine the JSON type (ARM or EER) and invoke the appropriate transformation procedure. The error log is then fetched and appended to the JSON for front-end logging requirements.

5.2.2. Front-end Classes

1. **Index**
   The index class is responsible for making this a single page application, this means that all components are rendered in one parent component (the index).

2. **App**
   The app class contains all subclasses and subcomponents so that the only component being rendered by the index class is the app class. Majority of application state and logic is performed in the app class.

3. **ERModel**
   This class is used to parse the JSON data into the EntityModel and EntityAttribute classes so that they can be used in the EntityGraphModel class to render the graph.

4. **ARModel**
   This class is used to parse the JSON data into the ARModel and RelationAttribute classes so that they can be used in the RelationGraphModel class to render the graph.

5. **EntityModel**
   This class is used to parse the JSON into a useable object that can be rendered in the EntityGraphModel class.

6. **RelationModel**
   This class is used to parse the JSON into a useable object that can be rendered in the RelationGraphModel class.

7. **EntityAttribute**
   This class is used to parse the JSON into a useable object that can be used to help render the create links between entities and their attributes.

8. **RelationAttribute**
   This class is used to parse the JSON into a useable object that can be used to help render the create links between relations.

9. **EntityGraphModel**
   This is where the rendering logic is performed in order to render the JSON as a EER as a SVG image.

10. **RelationGraphModel**
    This is where the rendering logic is performed in order to render the JSON as a ARM as a SVG image.

On the back-end side, Flask was used to provide a back-end server for front-end communication. The front-end involved a Node server to handle the server communication. JavaScript UI packages were used for rendering components. This is discussed in detail in the user manual.

6. Program Validation and Verification

Validation and verification were taken seriously throughout the development plan. 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 interface. Correctness of transformations and error reporting was the priority for the back-end. Testing for each is outlined below:

6.1. Method Testing

6.1.1. Back-end Testing

During the design phase, we decided that developing the back-end according to a test-driven driven development philosophy would be best. 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. For this particular project we found this approach to be effective since the success of the implementation of transformation rules from EER to ARM and back are completely defined in terms of (reasonable) classes of test cases, of which there are several benchmark examples on online (examples involving RMs were extended to be examples involving ARMs). In the beginning, during a single cycle of development, we began by creating the JSON representations for the EER and ARM models for the simplest test case, created a unit test for the test case and then implemented enough of the transformation rules in order that the test case would succeed. We continued to evolve the software prototype in this manner, refactoring code during each iteration, until we had satisfied all nine of the test cases, which were initially designed to test the software and ensure its correct functionality. We used Pythons unit testing frame work to create a set of automated unit tests for the ARMToEER() and EERToARM() functions in the Main.py file. For these unit tests, the JSON output was compared to a gold standard (correct output) JSON files to assert whether or not the two files were identical (success) or not (failure). Additionally a set of unit tests to test the functions; ReadEER(), WriteEER(), ReadARM(), WriteARM() functions from the ReadWriteEER and ReadWriteARM classes. In these unit tests a JSON file containining a EER or ARM model would be read into python objects using the ReadEER() / ReadARM() functions and then written back to file again using the writeEER() / writeARM() functions. Finally, we asserted whether or not the output was the same as the input, and thus determined the proper behaviour of the functions.

6.1.2. Front-end Testing

The front-end consisting of the GUI shown above is relatively simple and as such did not require extensive testing. Since this was the case, we tesed the functionality of the front-end independently of the back end initially using dummy data in JSON format both EER and ARM examples were used as well as synthetic error log information. The dummy data was used to ensure correct rendering of the model elements on the screen and monkey testing (black-box testing) was conducted in testing all the buttons on the interface to ensure the program will not crash even when the user simply presses buttons randomly.

6.2. Integration Testing

When integrating the front-end and back-end, user testing was conducted using each of the nine test cases. Both the ARM -¿ EER and EER-¿ARM transformations were executed on each test case, and the visual outputs were manually asserted to be correct. We performed these user tests several times with different users who had not ever seen the interface to ensure that the interface was intuitive and that the correct output was computed by the back end, and correctly displayed by the front end.

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

Process	**Technique**
Class Testing: test methods and state behaviour of classes	Black box testing: Unit tests for the reading JSON files, object creation for EER and ARM and writing back to JSON. This tests all the object classes and the I/O for the system.
Front-end Integration Testing: test the interaction of sets of classes for the front-end independently of the back-end.	• Monkey testing: randomly pressing buttons to ensure system does not crash • Behavioural Testing: using dummy data to ensure correct rendering of models and error log data.
Back-end Integration Testing: test the interaction of sets of classes for the back-end independently of the front-end.	Black box testing: unit tests for the 9 individual test cases. This tests all the object classes, as well as the object methods and all of the transformation methods for ARM to EER and EER and ARM
Validation Testing: test whether customer requirements are satisfied using specific test cases approved by the client	Success of test cases confirm that customer requirements have been satisfied.
System Testing: test the behaviour of the system as part of a larger environment. Back-end and front-end integration was thoroughly tested.	Manual testing using test cases (I/O testing): all test cases executed, results manually examined (both visual representation and JSON representation) **Note**: since the back and the front end are connected through a server, if the test cases pass and the back end renders the JSON files correctly there is no need to create automatic unit tests for the system as a whole.
UI/UX Testing: test the user friendliness and intuitiveness of the UI and UX by allowing non-expert users to test the system with minimal guidance.	Anonymous user testing: strangers (including non-expert users) used the system to ensure it the interface was intuitive and behaved as expected.

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

Test Cases		
	EER	**ARM**
1	• 1 strong entity • Simple attributes • Identifier attribute	• 1 relation • PK constraint (self) • Pathfd involving regular attributes • Regular attributes
2	• Single, Strong entity • Simple attributes • Composite attribute • Identifier attribute	• 1 relation • PK constraint (self) • Pathfd involving regular attributes • Regular attributes
3	• 1 strong entity • 1 weak entity • Zero to many relationship • Simple attributes • Identifier attribute	• 2 relations • PK constraint (self) • FK constraints • Pathfd involving regular attributes • Pathfd involving regular and OID attributes • Disjointness constraint • Regular attributes • OID attributes
4	• 2 strong entities • One to many relationship • Simple attributes • Identifier attributes	• 2 relations • PK constraint (self) • FK constraints • Pathfd involving regular attributes • Disjointness constraint • Regular attributes • OID attributes

5	• 2 strong entities • Many to many relationship with attributes • Simple attributes • Identifier attributes	• 3 relations • PK constraint (self) • FK constraints • Pathfd involving regular attributes • Pathfd involving only OID attributes (foreign keys) • Disjointness constraint • Regular attributes • OID attributes
6	• 1 strong entity • 3 weak entities • 3 IS-A relationships • Simple attributes • Identifier attributes • Multi-valued attribute	• 4 relations • PK constraint (self) • FK constraints • Pathfd involving regular attributes • Pathfd involving only OID attributes (foreign keys) • Pathfd involving regular and OID attributes • Disjointness constraint • Covering constraint • Regular attributes • OID attributes

7	• 3 strong entities • 1 weak entity • 4 one to many relationships • Simple attributes • Identifier attributes	• 4 relations • PK constraint (self) • FK constraints • Pathfd involving regular attributes • Pathfd involving only OID attributes (foreign keys) • Disjointness constraint • Regular attributes • OID attributes
8	• 3 strong entities with identifier attributes • 1 strong entity without identifier attribute • Simple attributes • Identifier attributes • A ternary relationship involving: – 2 one to many relationships – 1 2...N to many relationship	• 4 relations • PK constraint (self) • FK constraints • Pathfd involving regular attributes • Pathfd involving regular (non-id) attributes and OID attributes • Regular attributes • OID attributes **Note**: ER to ARM test case fails due to strong entity without identifier
9	• 1 strong entity • 1 recursive unary relationship • Regular attributes • Identifier attributes **Note**: ARM to ER test fails due to self-referencing foreign key	• 1 relation • PK constraint (self) • FK constraint, pointing to self • Regular attributes • OID attributes **Note**: ER to ARM test fails due to unary relationship

Table 3: Table explaining testing results based on test cases developed, unit testing of methods, as well as classes as well as class interaction. Results from user testing is also reported.

Type/Stage of Test	**Test Cases**		
	Normal Functioning	*Extreme boundary cases*	*Invalid Data (program should not crash)*
Preliminary test (see Appendix 3)	100% Pass	N/A	Program crash (fell over)
Development Unit Testing	100% Pass	80% Pass	Program crash (fell over)
Final Testing	100% Pass	100% Pass	Relevant user warning for incorrect input
UX/UI User Testing	Client satisfied. UX success.	Relevant warnings as appropriate. Will crash in certain cases.	Relevant warnings as appropriate. Will crash in certain cases.

Development Unit Testing Final Testing UX/UI User Testing

7. Conclusion

This paper explored the investigation of methods of transforming between EER and ARM by implementing the theoretical transformation rules outlined in the KnowID paper [1]. A visual interface was developed to allow non-expert users the ability to load EER/ARM models, transform them to ARM/EER, save the transformed models, display the models, and provide feedback on transformation success/failure and data loss during the transformation process. The project was a success as all the core requirements outlined in the paper have been achieved. User testing revealed non-expert users found the system user friendly and intuitive.
Provided the user adheres to the format outlined in the user manual - i.e. user input is valid and reasonable. See 8 for ideas about how this project can be extended.

8. Future Extension and Investigation

This project has much room for extension and improvement. Here we suggest some paths for future exploration. The major room for improvements and investigation is on testing this software and extending its features to work with an underlying database. We suggest taking an approach such that the software is layered above the database in a manner that the software does not require specific knowledge of the underlying architecture. Query-by-design can be investigated.
On the UI front, we recommend allowing users to create models by drag-and-drop functionality. This would further remove the need for non-expert users to get involved in any seemingly daunting process of modelling. Currently our software requires input of a model in JSON.
Finally, this project could be entirely hosted online for platform independent functionality.

9. Acknowledgments

We would like to thank Dr Maria Keet for expert theoretical guidance throughout the development of this project. We would also like to thank Ryan Lazar for making himself available for help with the implementation of this project.

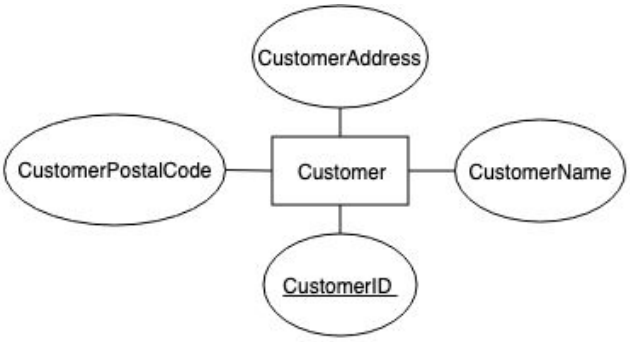
References

[1] C. Maria Keet and Pablo Ruben Fillottrani (2019) *KnowID: An architecture for efficient Knowledge-driven Information and Data access*. Addison-Wesley.

10. Appendix

A. Test Cases

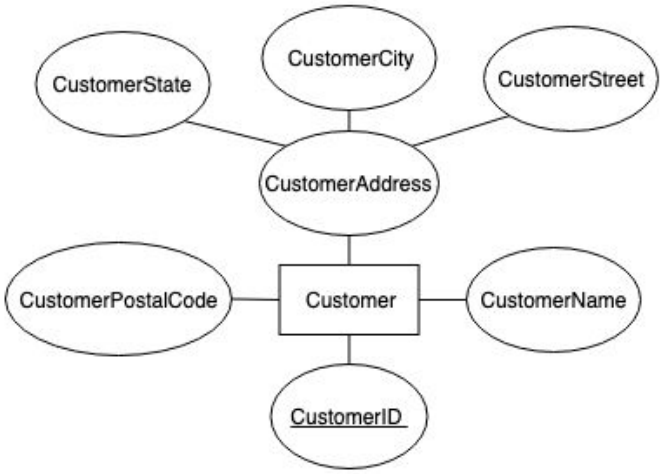
Test Case 1



Customer
pathFD (CustomerID) -> self*
CustomerID : Integer CustomerName : String CustomerAddress : String CustomerPostalCode : Integer

---

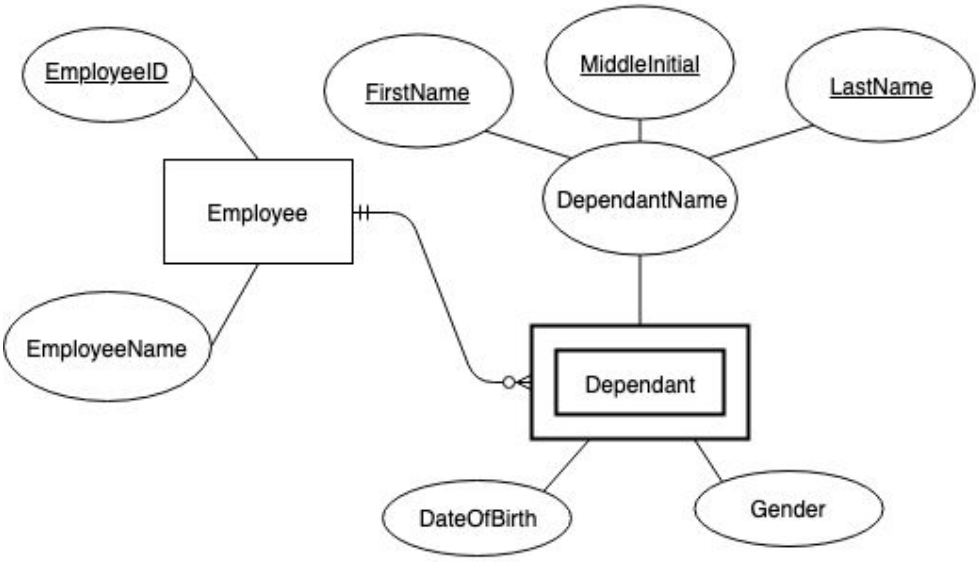
Test Case 2



Customer
pathFD (CustomerID) -> self*
CustomerID : Integer CustomerName : String CustomerState : String CustomerCity : String CustomerStreet : String CustomerPostalCode : Integer

---

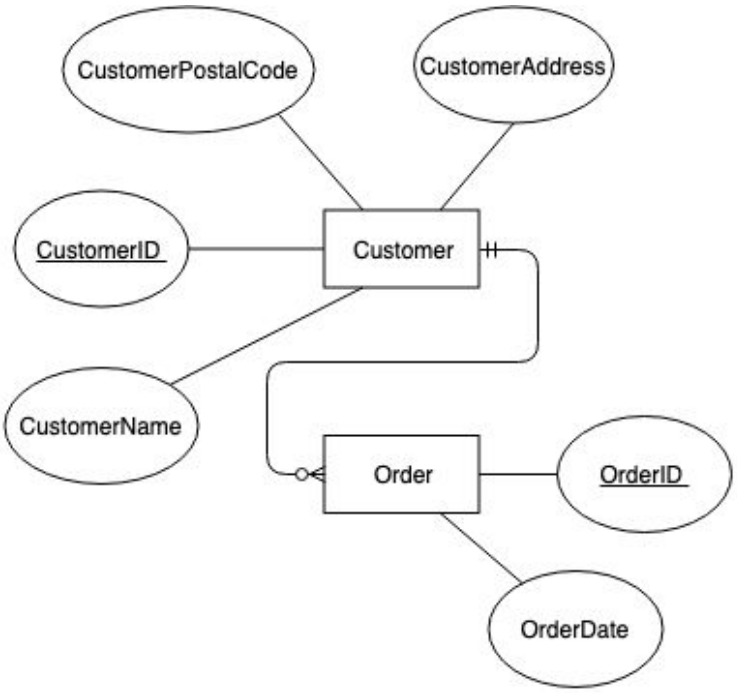
Test Case 3

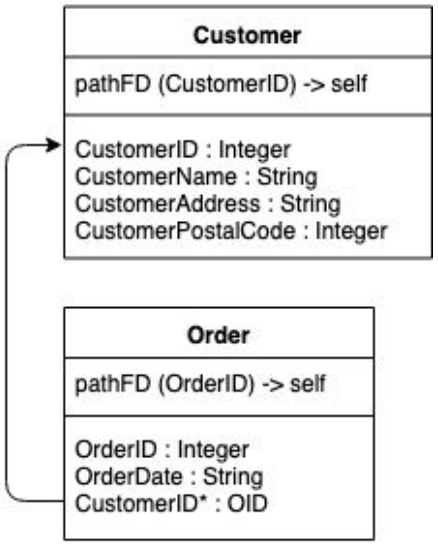


Employee
pathFD (EmployeeID) -> self*
EmployeeID : Integer EmployeeName : String

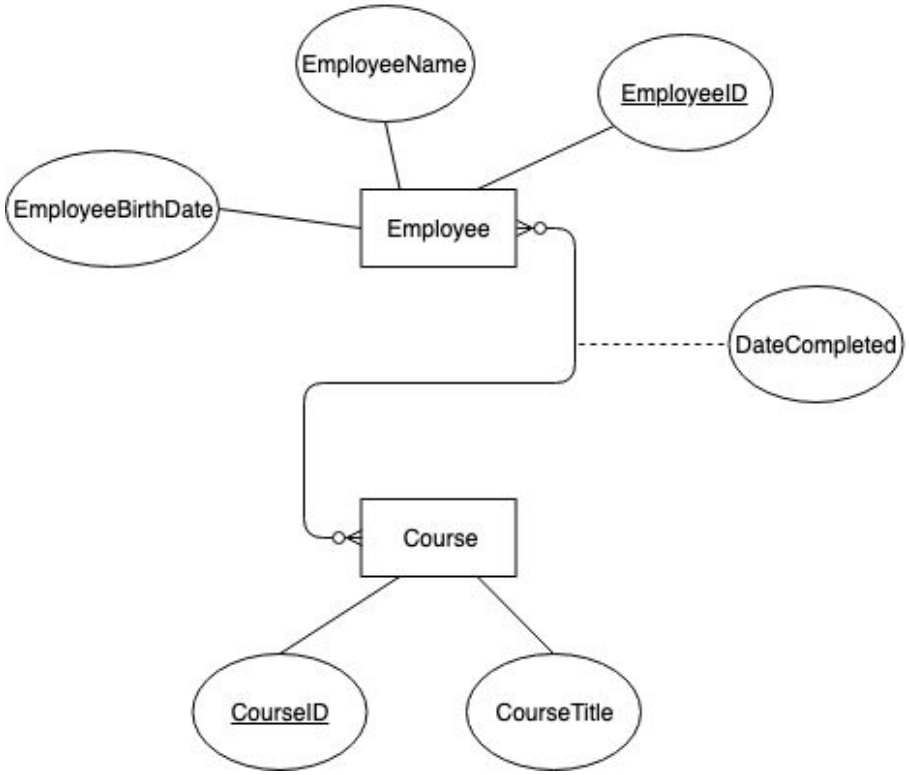
Dependant
pathFD (FirstName, MiddleInitial, LastName, EmployeeID) -> self*
FirstName : String MiddleInitial : String LastName : String EmployeeID* : OID DateOfBirth : String Gender : String

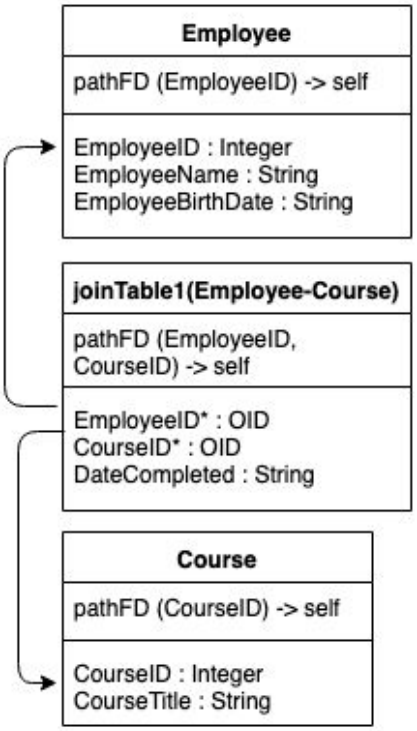
Test Case 4



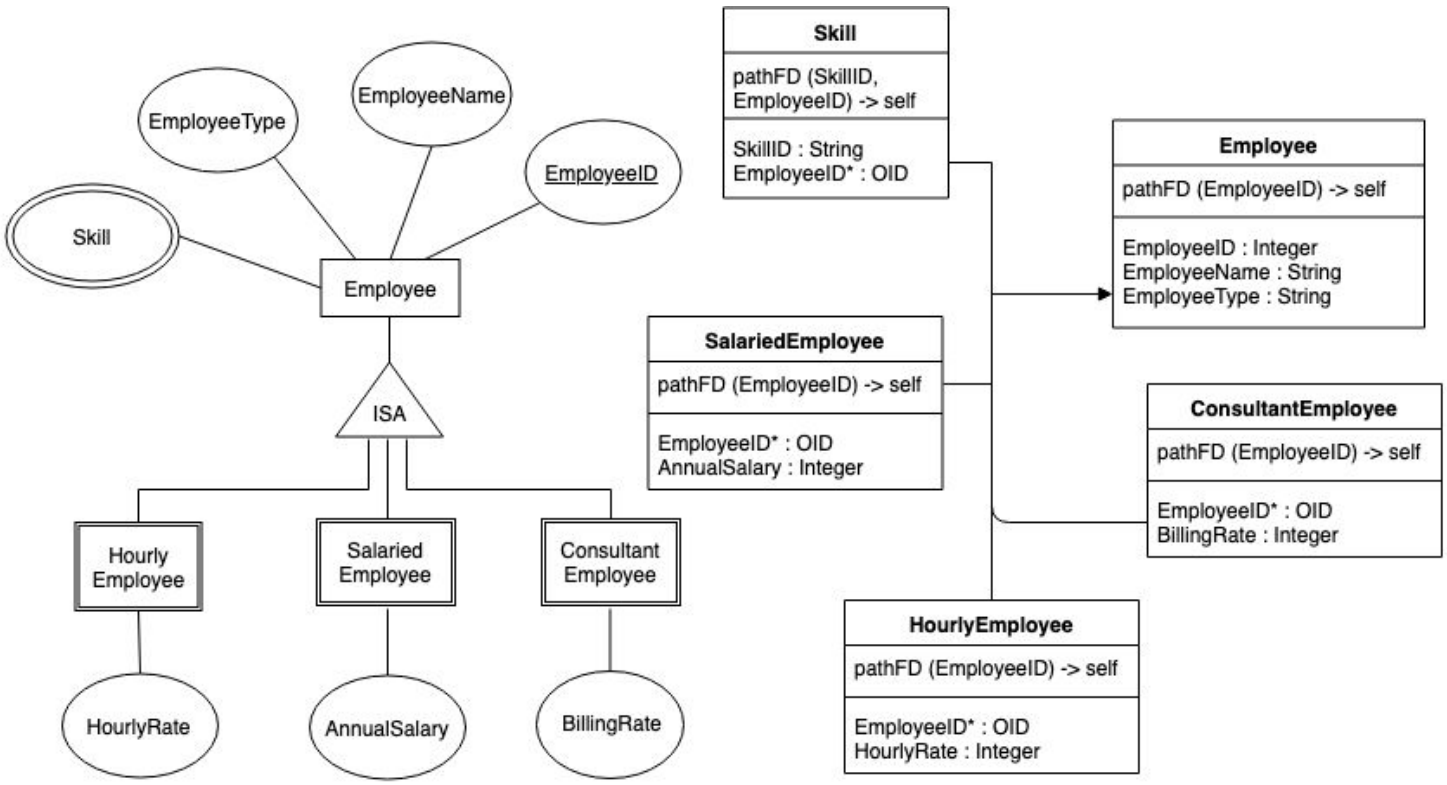


Test Case 5

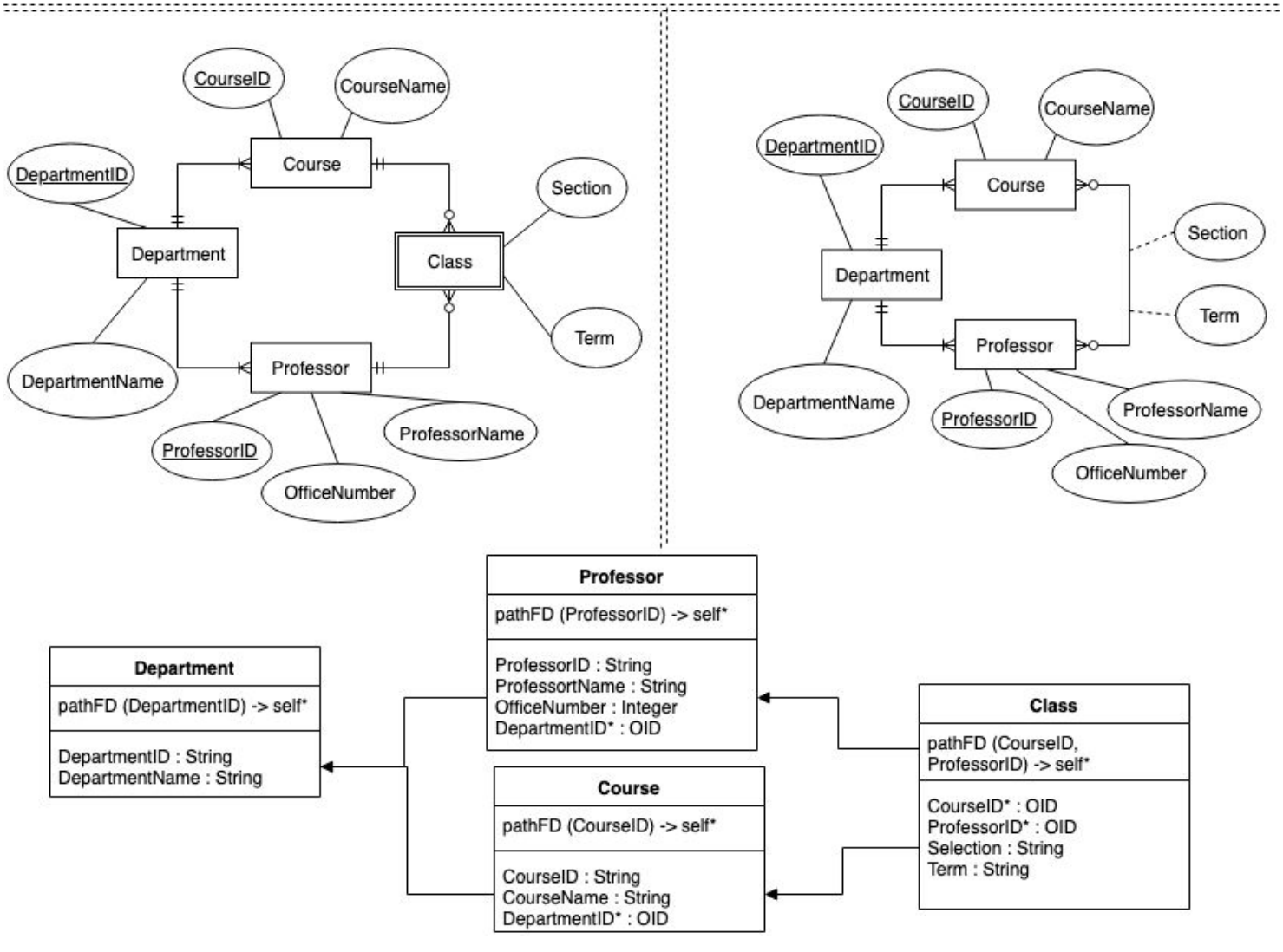




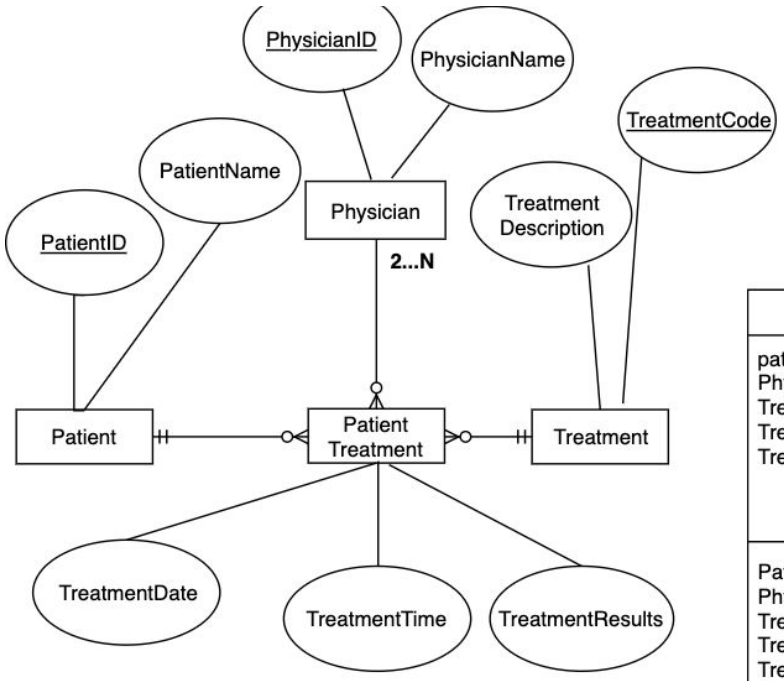
Test Case 6

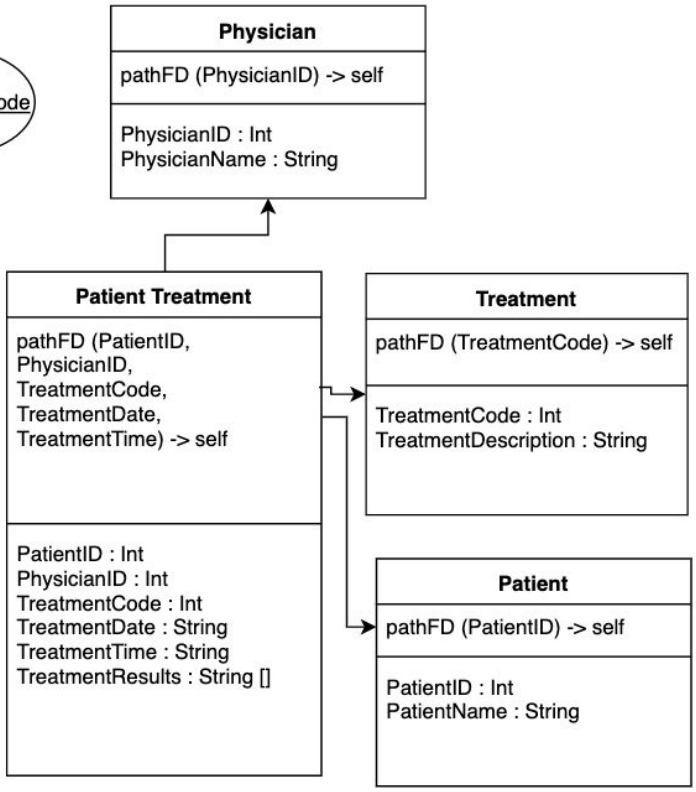


Test Case 7

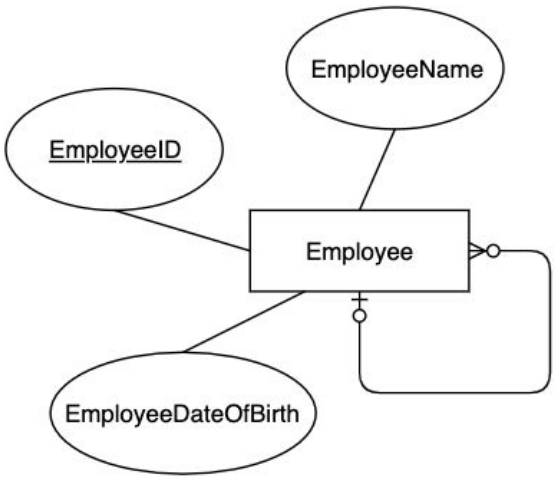


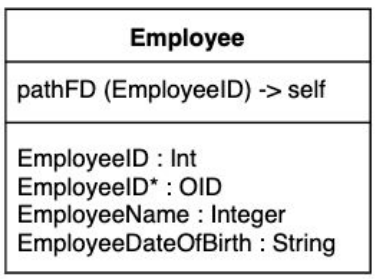
Test Case 8





Test Case 9





B. User Manual

ViKER User Manual

How to start the python server (back-end)

Using the terminal:

```
$ cd Back-end/
$ source venv/bin/activate
$ python WebServer.py
```

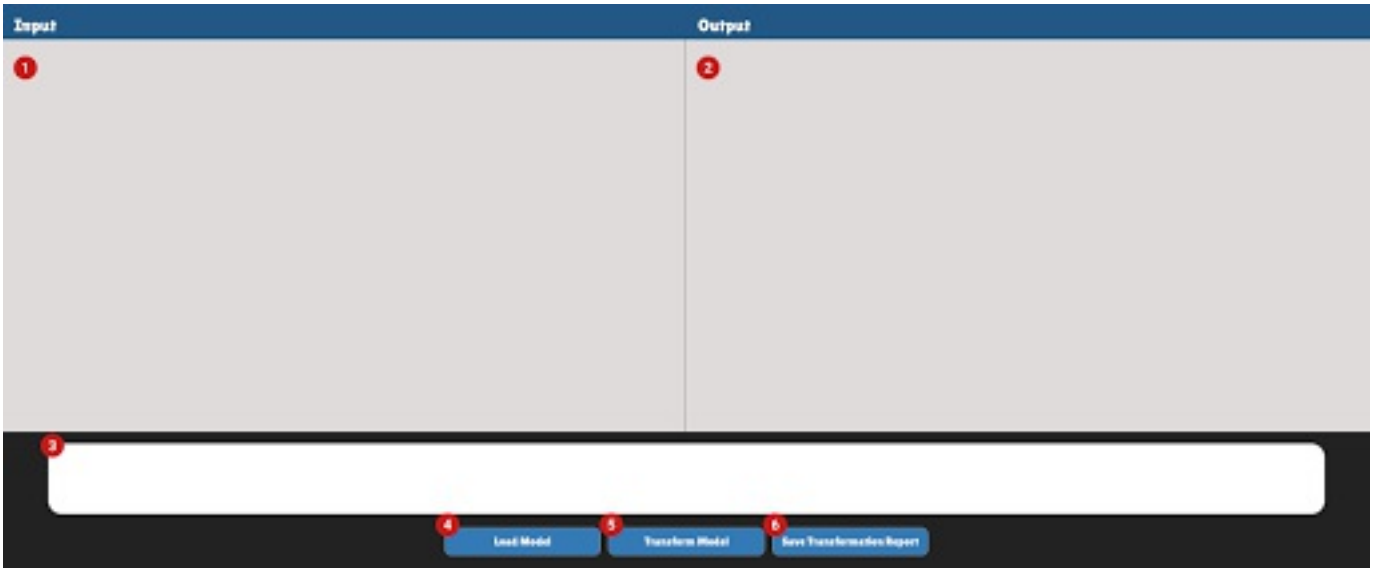
How to start the node server (front-end)

Using the terminal:

```
$ cd Front-end/
$ npm install
$ npm start
```

Your browser will then open on http://localhost:3000/

User Interface



1. Where the input model is rendered
2. Where the output model is rendered
3. Where the error log is printed
4. Button to load model as a JSON file
5. Button to send input model to server and transform to output model
6. Saves the output model as a JSON file along with the error log

What the models look like in JSON

ARM representation

```
{
"relations": [
    {
        "attributes": [
            {
```

```
            "AttributeName": "self",
            "dataType": "OID",
            "isConcrete": false,
            "isFK": false,
            "isPathFunctionalDependancy": false
        },
        {
            "AttributeName": "CustomerID",
            "dataType": "Integer",
            "isConcrete": true,
            "isFK": false,
            "isPathFunctionalDependancy": true
        },
        {
            "AttributeName": "CustomerName",
            "dataType": "String",
            "isConcrete": true,
            "isFK": false,
            "isPathFunctionalDependancy": false
        },
        {
            "AttributeName": "CustomerAddress",
            "dataType": "String",
            "isConcrete": true,
            "isFK": false,
            "isPathFunctionalDependancy": false
        },
        {
```

```
                "AttributeName": "CustomerPostalCode",
                "dataType": "Integer",
                "isConcrete": true,
                "isFK": false,
                "isPathFunctionalDependancy": false
            }
        ],
        "coveredBy": [],
        "disjointWith": [],
        "inheritsFrom": "none",
        "name": "Customer"
    }
  ]
}
```

EER representation

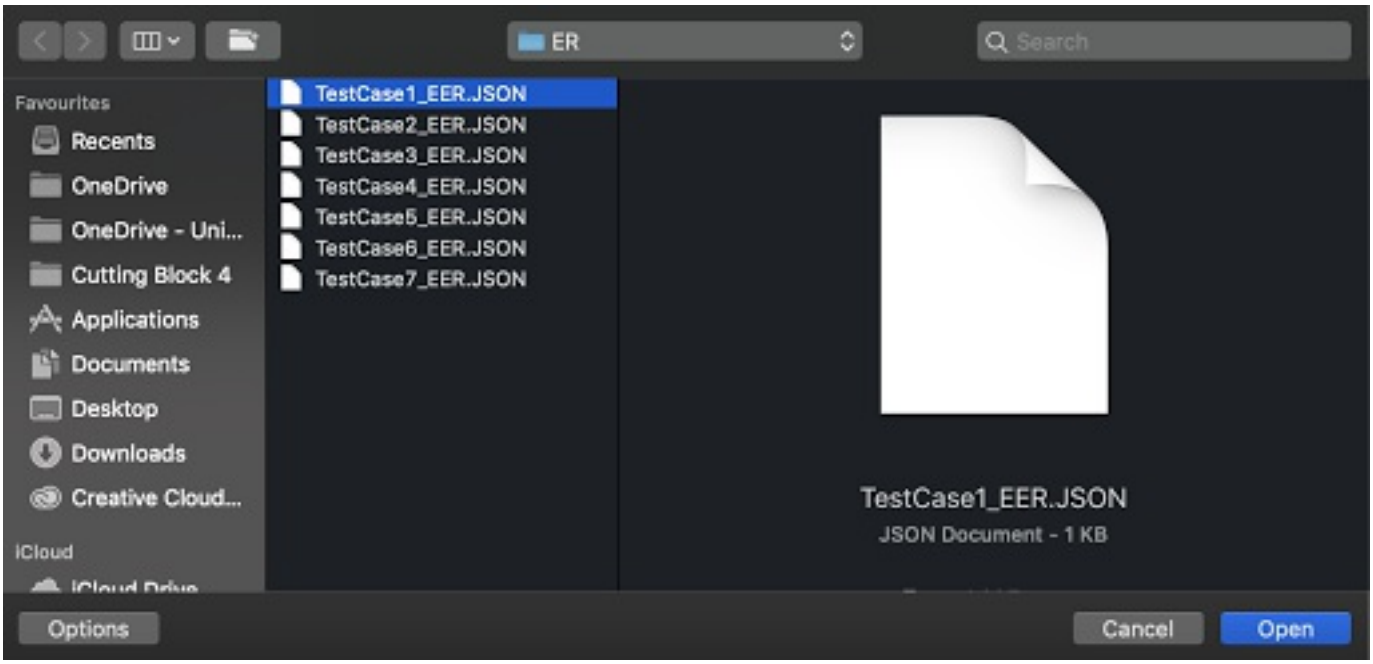
```
{
"entities": [
    {
        "attributes": [
            {
                "AttributeName": "CustomerID",
                "composedOf": [],
                "isIdentifier": true,
                "isMultiValued": false
            },
```

```
            {
                "AttributeName": "CustomerName",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "CustomerAddress",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "CustomerPostalCode",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": true,
        "name": "Customer",
        "relationships": []
    }
  ]
}
```

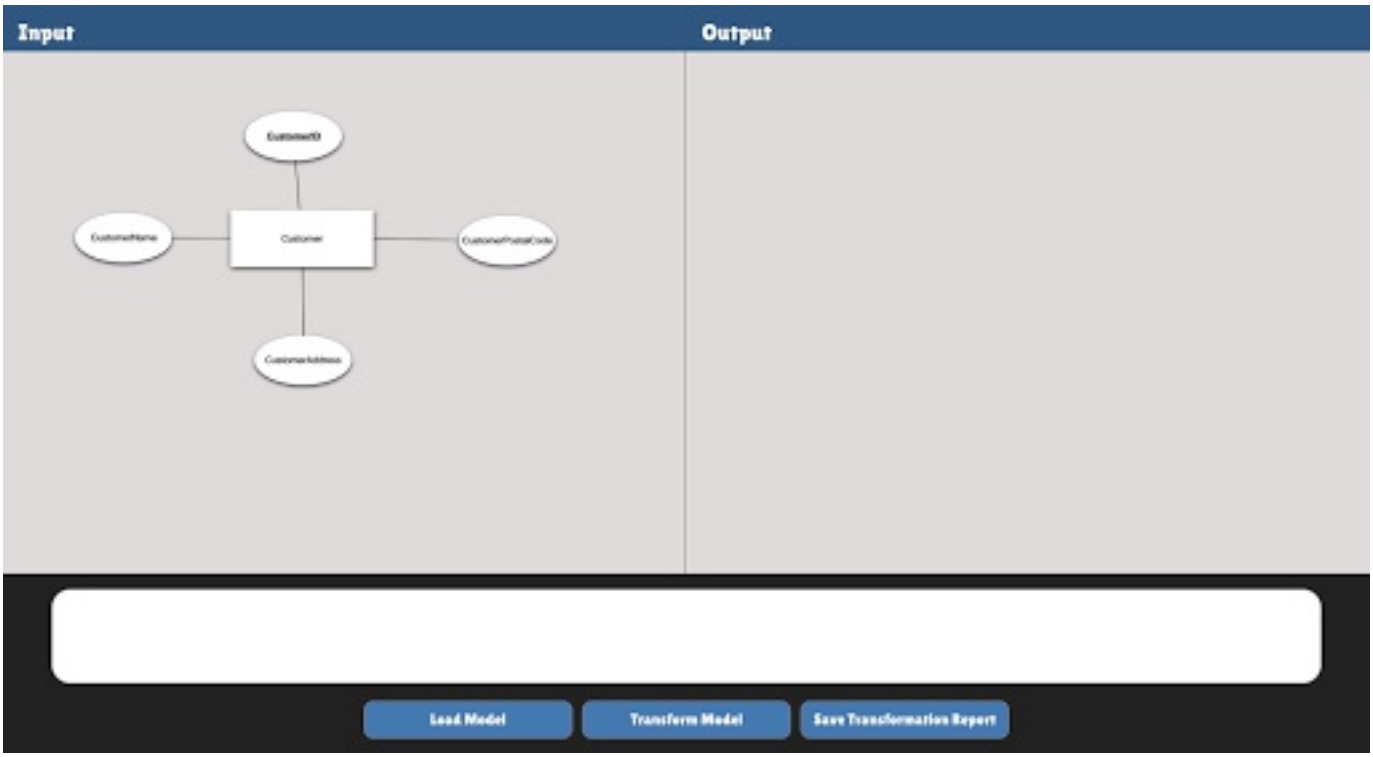
Loading a model

1.) Click the ‘Load Model’ button

2.) The file chooser will open such as below:

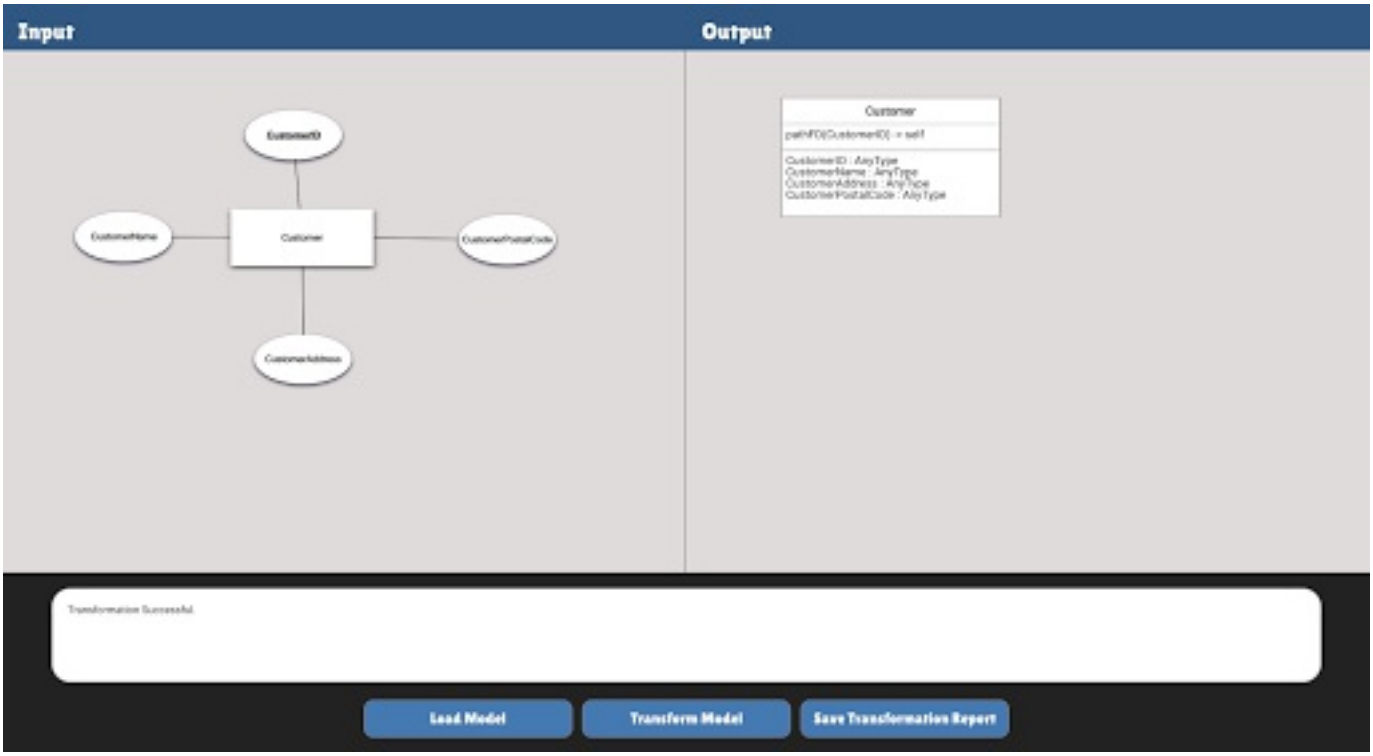


3.) The model will then render such as below:



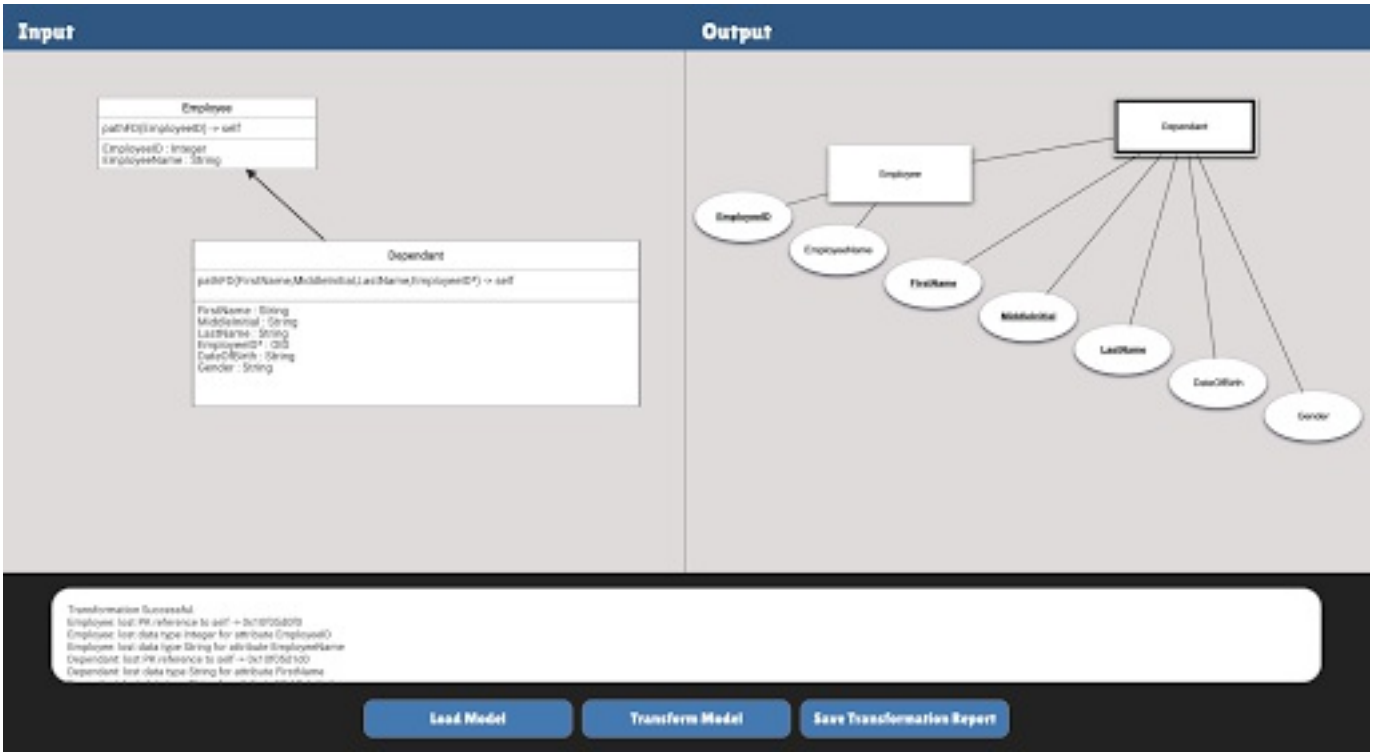
4.) Click the ‘Transform Model’ button to transform the model and

render the output model, such as below:



5.) You can then save the output model and the error report as JSON, by clicking the ‘Save Transformation Report’ button. You will see the file being downloaded, such as below:

6.) You can also transform from ARM to EER, the same way!



*Note: All diagrams can be moved around and interacted with!

Examples

EER

ISA & Multivalued attribute

JSON

```
{
"entities": [
    {
        "attributes": [
            {
                "AttributeName": "EmployeeID",
                "composedOf": [],
```

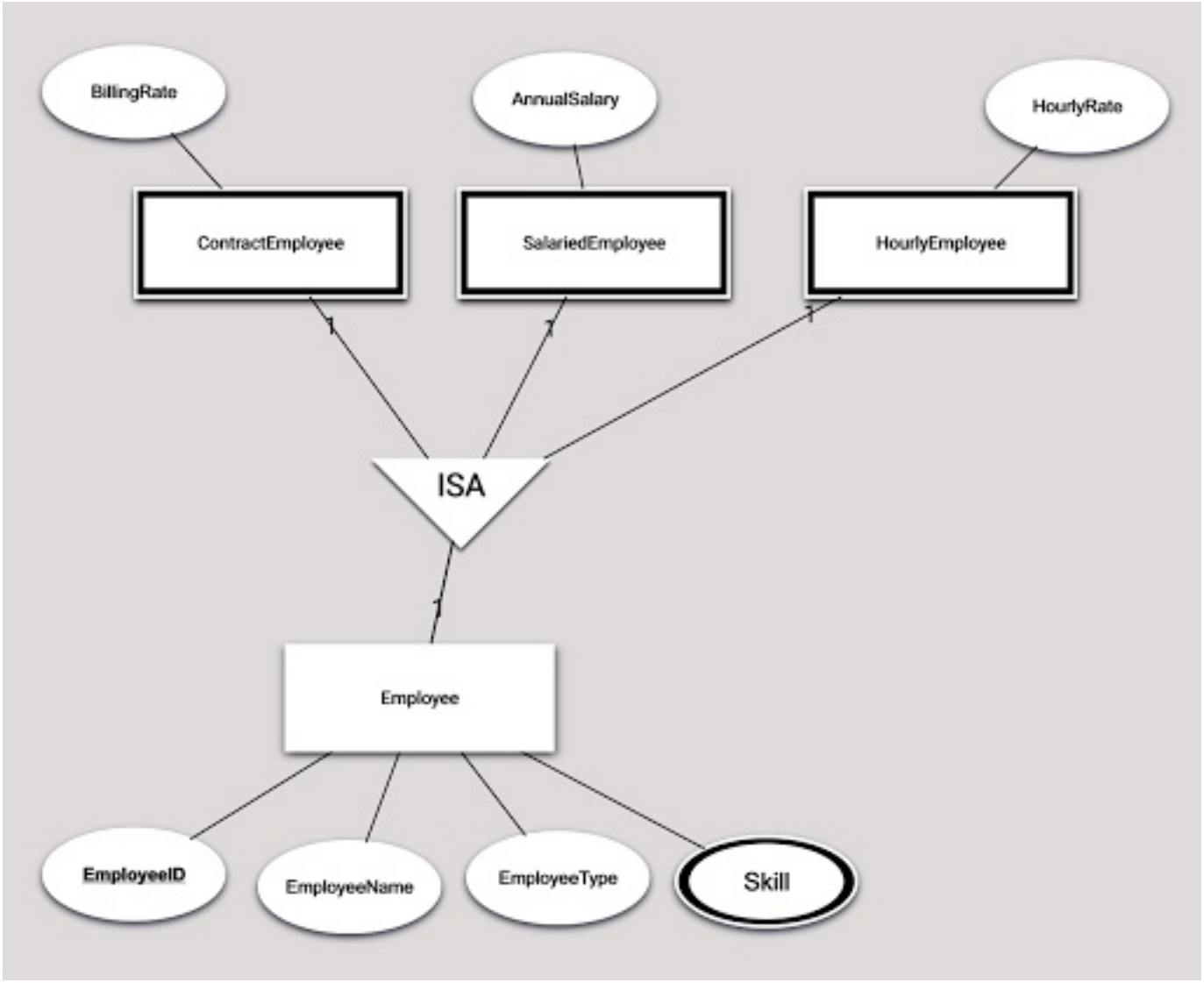
```
                "isIdentifier": true,
                "isMultiValued": false
            },
            {
                "AttributeName": "EmployeeName",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "EmployeeType",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "Skill",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": true
            }
        ],
        "isStrong": true,
        "name": "Employee",
        "relationships": [
            {
                "Entity": "HourlyEmployee",
                "RelationTypeForeign": "ISA",
```

```
                "RelationTypeLocal": "ISA",
                "relationAttributes": []
            },
            {
                "Entity": "SalariedEmployee",
                "RelationTypeForeign": "ISA",
                "RelationTypeLocal": "ISA",
                "relationAttributes": []
            },
            {
                "Entity": "ContractEmployee",
                "RelationTypeForeign": "ISA",
                "RelationTypeLocal": "ISA",
                "relationAttributes": []
            }
        ]
    },
    {
        "attributes": [
            {
                "AttributeName": "HourlyRate",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": false,
        "name": "HourlyEmployee",
```

```
        "relationships": [
            {
                "Entity": "Employee",
                "RelationTypeForeign": "ISA",
                "RelationTypeLocal": "ISA",
                "relationAttributes": []
            }
        ]
    },
    {
        "attributes": [
            {
                "AttributeName": "AnnualSalary",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": false,
        "name": "SalariedEmployee",
        "relationships": [
            {
                "Entity": "Employee",
                "RelationTypeForeign": "ISA",
                "RelationTypeLocal": "ISA",
                "relationAttributes": []
            }
        ]
```

```
    },
    {
        "attributes": [
            {
                "AttributeName": "BillingRate",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": false,
        "name": "ContractEmployee",
        "relationships": [
            {
                "Entity": "Employee",
                "RelationTypeForeign": "ISA",
                "RelationTypeLocal": "ISA",
                "relationAttributes": []
            }
        ]
    }
  ]
}
```

Model



Composite attributes

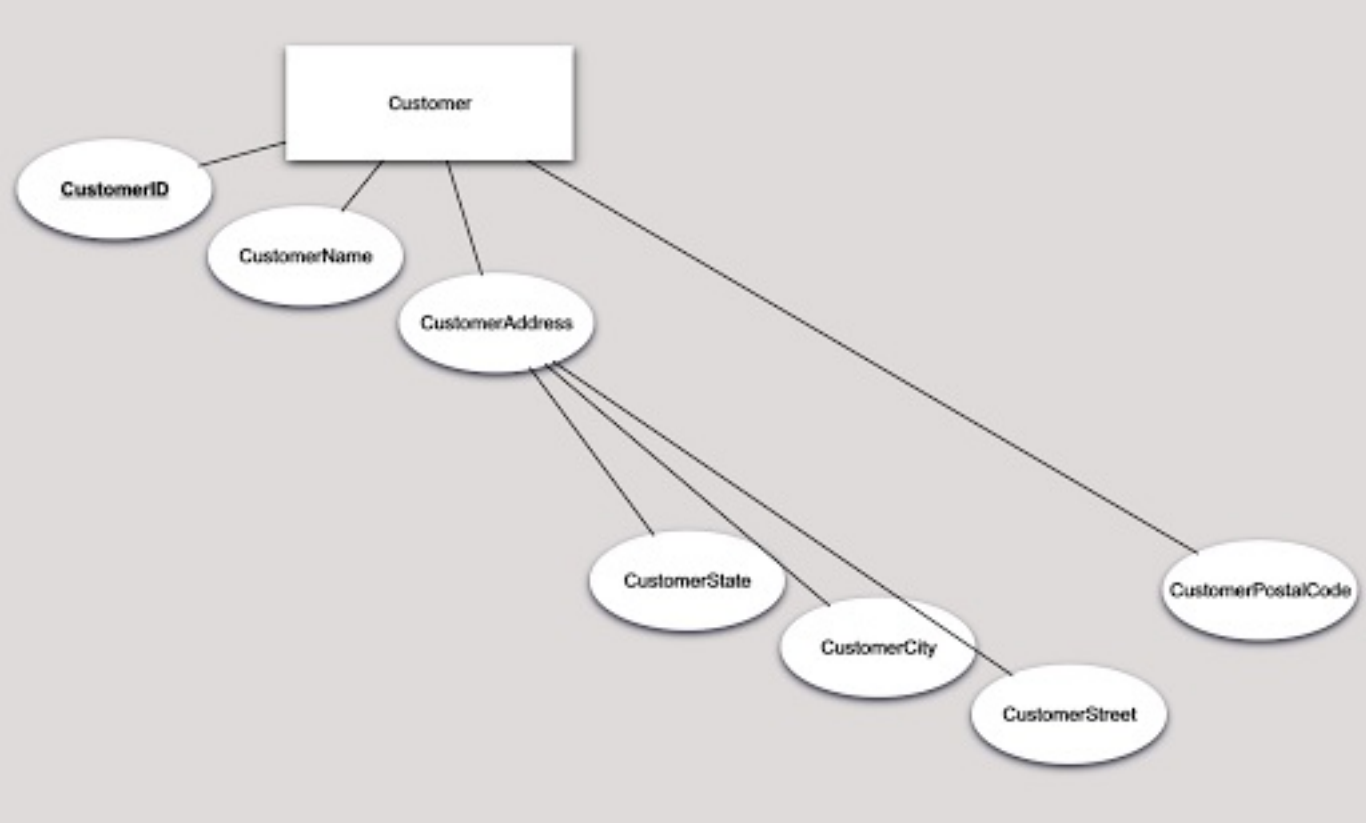
JSON

```
{
"entities": [
    {
        "attributes": [
            {
                "AttributeName": "CustomerID",
                "composedOf": [],
                "isIdentifier": true,
```

```
                "isMultiValued": false
            },
            {
                "AttributeName": "CustomerName",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "CustomerAddress",
                "composedOf": [
                    "CustomerState",
                    "CustomerCity",
                    "CustomerStreet"
                ],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "CustomerState",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "CustomerCity",
                "composedOf": [],
                "isIdentifier": false,
```

```
                "isMultiValued": false
            },
            {
                "AttributeName": "CustomerStreet",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "CustomerPostalCode",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": true,
        "name": "Customer",
        "relationships": []
    }
  ]
}
```

Model



Weak entity

JSON

```
{
"entities": [
    {
        "attributes": [
            {
                "AttributeName": "DepartmentID",
                "composedOf": [],
                "isIdentifier": true,
                "isMultiValued": false
            },
            {
                "AttributeName": "DepartmentName",
```

```
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": true,
        "name": "Department",
        "relationships": [
            {
                "Entity": "Course",
                "RelationTypeForeign": "ZeroOrMany",
                "RelationTypeLocal": "ExactlyOne",
                "relationAttributes": []
            },
            {
                "Entity": "Professor",
                "RelationTypeForeign": "ZeroOrMany",
                "RelationTypeLocal": "ExactlyOne",
                "relationAttributes": []
            }
        ]
    },
    {
        "attributes": [
            {
                "AttributeName": "ProfessorID",
                "composedOf": [],
                "isIdentifier": true,
```

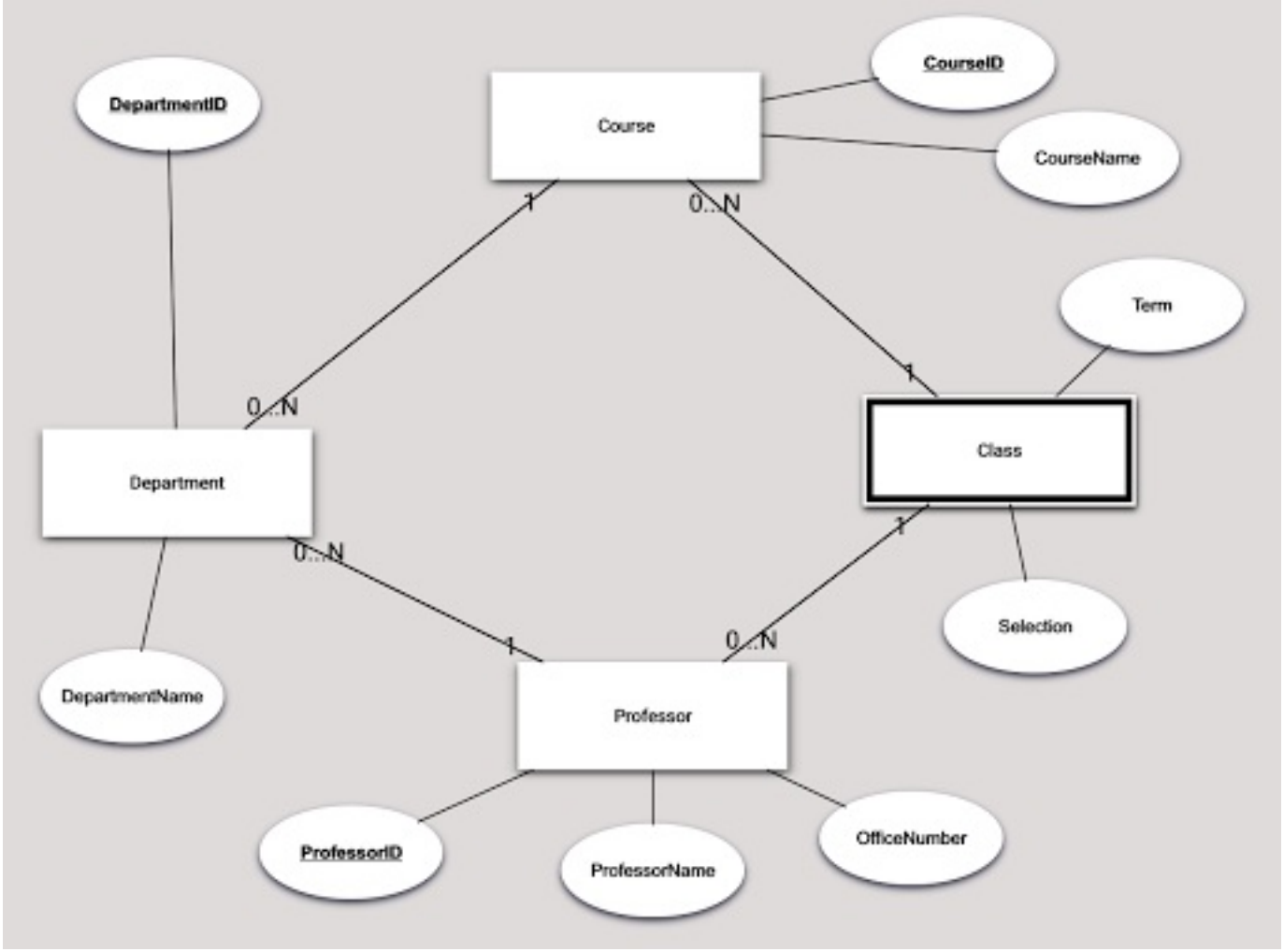
```
                "isMultiValued": false
            },
            {
                "AttributeName": "ProfessorName",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "OfficeNumber",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": true,
        "name": "Professor",
        "relationships": [
            {
                "Entity": "Department",
                "RelationTypeForeign": "ExactlyOne",
                "RelationTypeLocal": "ZeroOrMany",
                "relationAttributes": []
            },
            {
                "Entity": "Class",
                "RelationTypeForeign": "ZeroOrMany",
                "RelationTypeLocal": "ExactlyOne",
```

```
                "relationAttributes": []
            }
        ]
    },
    {
        "attributes": [
            {
                "AttributeName": "CourseID",
                "composedOf": [],
                "isIdentifier": true,
                "isMultiValued": false
            },
            {
                "AttributeName": "CourseName",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": true,
        "name": "Course",
        "relationships": [
            {
                "Entity": "Department",
                "RelationTypeForeign": "ExactlyOne",
                "RelationTypeLocal": "ZeroOrMany",
                "relationAttributes": []
            },
```

```
            {
                "Entity": "Class",
                "RelationTypeForeign": "ZeroOrMany",
                "RelationTypeLocal": "ExactlyOne",
                "relationAttributes": []
            }
        ]
    },
    {
        "attributes": [
            {
                "AttributeName": "Selection",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "Term",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": false,
        "name": "Class",
        "relationships": [
            {
                "Entity": "Course",
```

```
                "RelationTypeForeign": "ExactlyOne",
                "RelationTypeLocal": "ZeroOrMany",
                "relationAttributes": []
            },
            {
                "Entity": "Professor",
                "RelationTypeForeign": "ExactlyOne",
                "RelationTypeLocal": "ZeroOrMany",
                "relationAttributes": []
            }
        ]
    }
  ]
}
```

Model



Relationship attribute

JSON

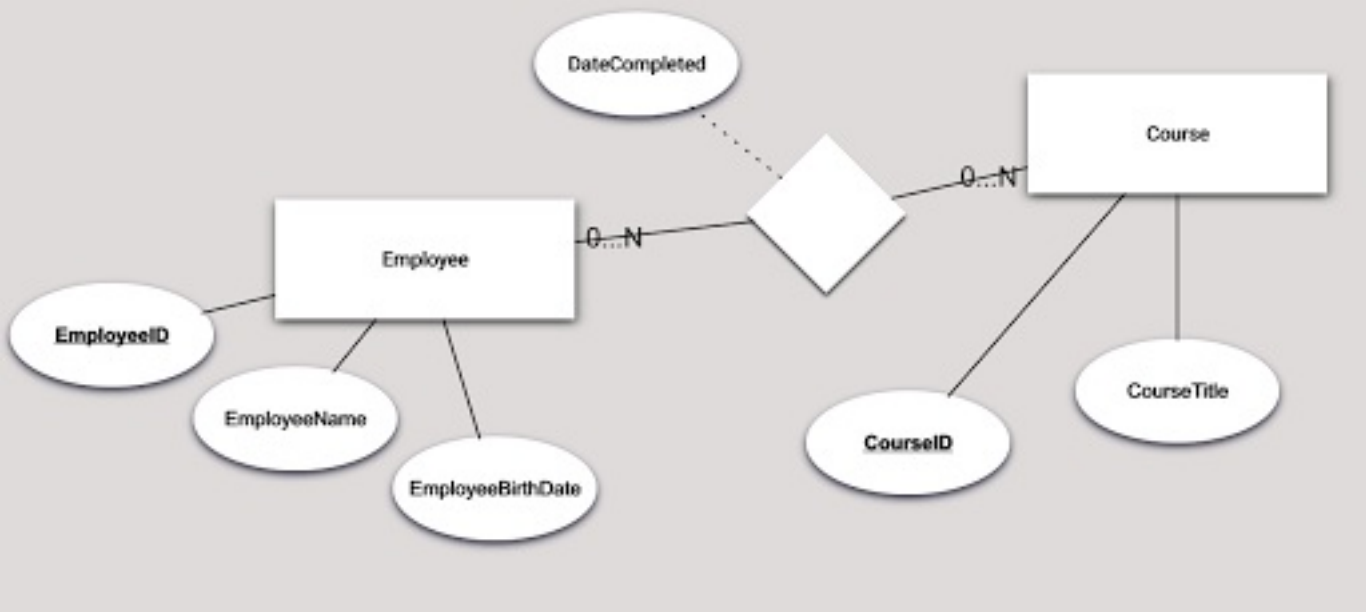
```
{
"entities": [
    {
        "attributes": [
            {
                "AttributeName": "EmployeeID",
                "composedOf": [],
                "isIdentifier": true,
                "isMultiValued": false
```

```
            },
            {
                "AttributeName": "EmployeeName",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            },
            {
                "AttributeName": "EmployeeBirthDate",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": true,
        "name": "Employee",
        "relationships": [
            {
                "Entity": "Course",
                "RelationTypeForeign": "ZeroOrMany",
                "RelationTypeLocal": "ZeroOrMany",
                "relationAttributes": [
                    "DateCompleted"
                ]
            }
        ]
    },
    {
```

```
        "attributes": [
            {
                "AttributeName": "CourseID",
                "composedOf": [],
                "isIdentifier": true,
                "isMultiValued": false
            },
            {
                "AttributeName": "CourseTitle",
                "composedOf": [],
                "isIdentifier": false,
                "isMultiValued": false
            }
        ],
        "isStrong": true,
        "name": "Course",
        "relationships": [
            {
                "Entity": "Employee",
                "RelationTypeForeign": "ZeroOrMany",
                "RelationTypeLocal": "ZeroOrMany",
                "relationAttributes": [
                    "DateCompleted"
                ]
            }
        ]
    }
]
```

```
}
```

Model



ARM

JSON

```
{
"relations": [
    {
        "attributes": [
            {
                "AttributeName": "self",
                "dataType": "OID",
                "isConcrete": false,
                "isFK": false,
                "isPathFunctionalDependancy": false
            },
            {
```

```
                "AttributeName": "DepartmentID",
                "dataType": "String",
                "isConcrete": true,
                "isFK": false,
                "isPathFunctionalDependancy": true
            },
            {
                "AttributeName": "DepartmentName",
                "dataType": "String",
                "isConcrete": true,
                "isFK": false,
                "isPathFunctionalDependancy": false
            }
        ],
        "coveredBy": [],
        "disjointWith": ["Professor", "Course", "Class"],
        "inheritsFrom": "none",
        "name": "Department"
    },
    {
        "attributes": [
            {
                "AttributeName": "self",
                "dataType": "OID",
                "isConcrete": false,
                "isFK": false,
                "isPathFunctionalDependancy": false
            },
```

```
            {
                "AttributeName": "DepartmentID",
                "dataType": "OID",
                "isConcrete": false,
                "isFK": true,
                "isPathFunctionalDependancy": false
            },
            {
                "AttributeName": "CourseID",
                "dataType": "String",
                "isConcrete": true,
                "isFK": false,
                "isPathFunctionalDependancy": true
            },
            {
                "AttributeName": "CourseName",
                "dataType": "String",
                "isConcrete": true,
                "isFK": false,
                "isPathFunctionalDependancy": false
            }
        ],
        "coveredBy": [],
        "disjointWith": ["Professor", "Department", "Class
"],
        "inheritsFrom": "none",
        "name": "Course"
    },
```

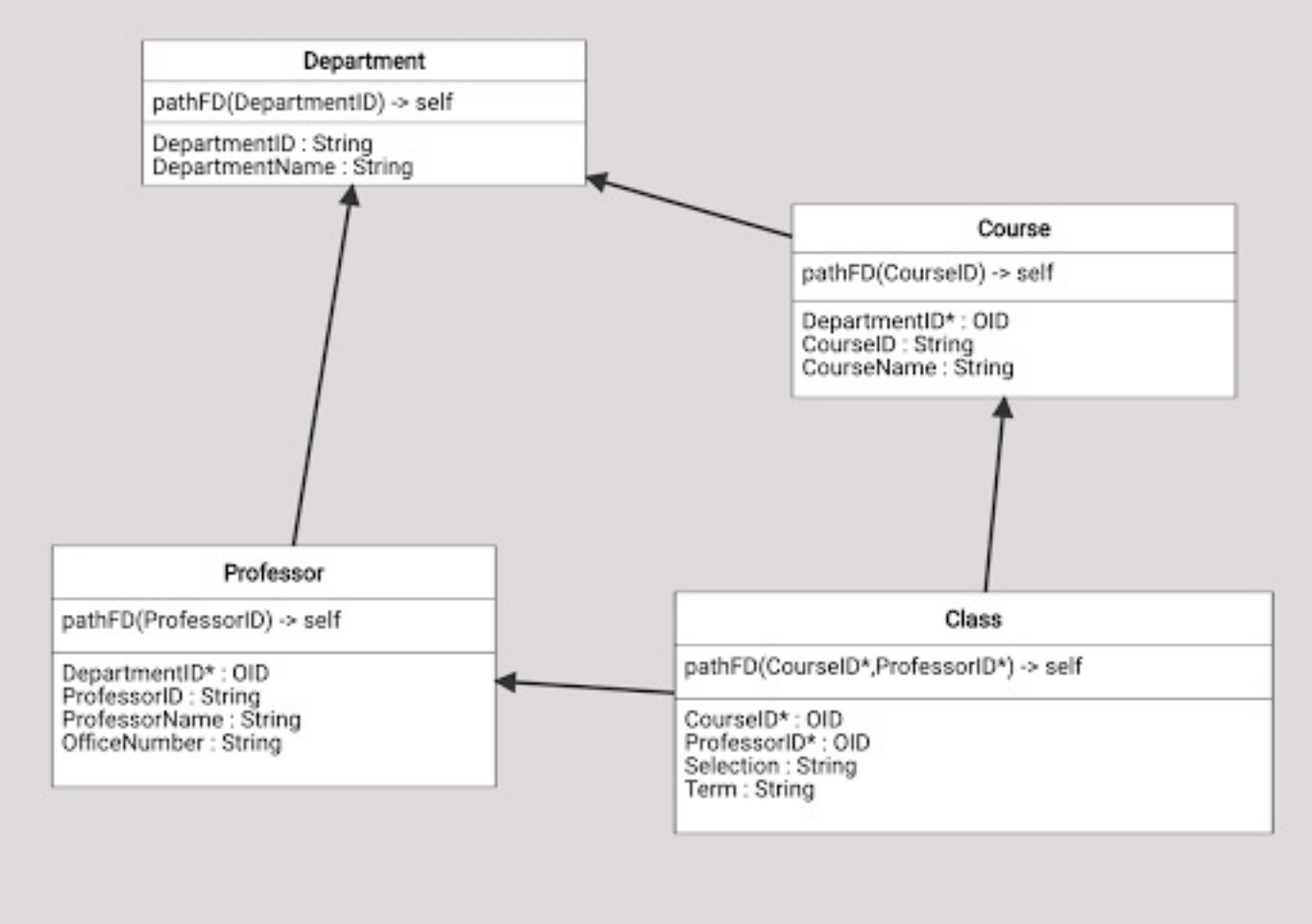
```
    {
        "attributes": [
            {
                "AttributeName": "self",
                "dataType": "OID",
                "isConcrete": false,
                "isFK": false,
                "isPathFunctionalDependancy": false
            },
            {
                "AttributeName": "DepartmentID",
                "dataType": "OID",
                "isConcrete": false,
                "isFK": true,
                "isPathFunctionalDependancy": false
            },
            {
                "AttributeName": "ProfessorID",
                "dataType": "String",
                "isConcrete": true,
                "isFK": false,
                "isPathFunctionalDependancy": true
            },
            {
                "AttributeName": "ProfessorName",
                "dataType": "String",
                "isConcrete": true,
                "isFK": false,
```

```
                "isPathFunctionalDependancy": false
            },
            {
                "AttributeName": "OfficeNumber",
                "dataType": "String",
                "isConcrete": true,
                "isFK": false,
                "isPathFunctionalDependancy": false
            }
        ],
        "coveredBy": [],
        "disjointWith": ["Course", "Department", "Class"],
        "inheritsFrom": "none",
        "name": "Professor"
    },
    {
        "attributes": [
            {
                "AttributeName": "self",
                "dataType": "OID",
                "isConcrete": false,
                "isFK": false,
                "isPathFunctionalDependancy": false
            },
            {
                "AttributeName": "CourseID",
                "dataType": "OID",
                "isConcrete": false,
```

```
                "isFK": true,
                "isPathFunctionalDependancy": true
            },
            {
                "AttributeName": "ProfessorID",
                "dataType": "OID",
                "isConcrete": false,
                "isFK": true,
                "isPathFunctionalDependancy": true
            },
            {
                "AttributeName": "Selection",
                "dataType": "String",
                "isConcrete": true,
                "isFK": false,
                "isPathFunctionalDependancy": false
            },
            {
                "AttributeName": "Term",
                "dataType": "String",
                "isConcrete": true,
                "isFK": false,
                "isPathFunctionalDependancy": false
            }
        ],
        "coveredBy": [],
        "disjointWith": ["Course", "Department", "Professo
r"],
```

```
        "inheritsFrom": "none",
        "name": "Class"
    }
  ]
}
```

Model



Packages used

Front-end

- Node js
- React js

- Jointjs

Back-end

- Flask
- virtualenv
- numpy
- Flask-CORS