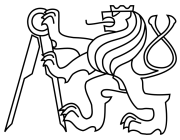
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	ER diagrams web component II
Student:	Bc. Tomáš Fedor
Supervisor:	Ing. Jiří Hunka
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2017/18

Instructions

A tool for education support with a utility for modeling basic ER diagrams (logical database model), used for example for testing students in a semester, is being developed for the BI-DBS course. The goal of the thesis is to create a new, extended, and superior version of this web component.

Analyse requirements of the ER diagram creation utility for roles student, examined student, and teacher and compare them with the current functionality.
Analyse the scope of the work done in a logical model in the BI-DBS classes during student examination and testing.
Analyse the user interface of other existing modeling tools.
Based on the analysis design a new web component with a suitable user interface, while taking into consideration potential future changes, extensions, and usability.
Implement the component and properly test both the functionality and the user interaction.
Analyse possible future extensions of the component, e.g., transformation from the logical to the relational model.

References

Will be provided by the supervisor.

L.S.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 30, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

ER Diagrams Web Component II

Bc. Tomáš Fedor

Supervisor: Ing. Jiří Hunka

9th January 2017

Acknowledgements

I would like to thank my supervisor Ing. Jiří Hunka for his professional guidance, organization of user testing and for not giving up on me, Oldřich Malec and Pavel Kovář for deploying data modeler into DBS Portal, for functionality testing and ideas for improvements and my family for not asking “how’s it going?” too often.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for nonprofit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 9th January 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Fedor, Tomáš. *ER Diagrams Web Component II.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Táto práca rozoberá vývoj webového nástroja na tvorbu Entity-Relationship diagramov, ktorý slúži na podporu edukačného procesu predmetu Databázové systémy Fakulty Informačných Technológií Českého Vysokého Učení Technického. Práca sa zaoberá všetkými časťami vývojového procesu: analýzou, ktorá slúži ako základ pre návrh a implementáciu; návrhom užívateľského prostredia a princípmi, ktorých sa návrh držal; implementáciou s ohľadom na ďalší vývoj; teoretickým aj praktickým testovaním výsledného nástroja.

Kľúčové slová Entity-Relationship Diagram, Logický model, Užívateľské prostredie, Testovanie, Tvorba modelov, Webová komponenta

Abstract

This thesis discusses the development of a web component for Entity-Relationship diagrams modeling. The purpose of this component is to support the education process of Database Systems course at Faculty of Information Technology of Czech Technical University. Thesis covers all parts of the development process: analysis as a base for both design and implementation; user interface

design and principles that were followed; implementation with regards to future development; testing, including heuristic evaluation as well as empirical testing with users of the modeler.

Keywords Entity-Relationship Diagram, Logical Model, User interface, Testing, Data modeler, Web component

Contents

List of Figures

List of Tables

Introduction

Models are a vital part of engineering and science. Software engineering is no exception. In software development, models are being utilized during every stage of projects life cycle. Not only do they help to properly analyse and understand the problem domain or design the product, they also serve as a communication tool between all parties, as well as a documentation tool. The advantage of models is that they provide different view of the same system based on the needs of the user.

Database, a collection of large amount of real world data, is itself a model of the real world. However, a proper understanding of the data is needed prior to building a database. This understanding is achieved by data (logical) models which show objects and their characteristics, relationships between objects, and constraints found during prior analysis and needs assessment. Data models thus provide crucial high-level understanding of that part of the real world that is important for the project and serve as a logical view of a database for database designers. Logical models are by their nature impartial against the technology that will be used to build and deploy the database.

However, logical view achieved by data models is not the only view that models could provide during database development. If relational database management system is used, relational models show further, more technical, details needed for final implementation, and even lower levels of view could be provided by models showing physical workings of the database management system if needed.

During bachelor programme at Faculty of Information Technology of Czech Technical University in Prague, students take mandatory Database Systems course, an introductory course to database management systems and database design. To aid an educational process, a portal, which students use to submit semestral projects as well as take exams is being developed. A crucial part of this portal is a data modeler web component used for creating logical models of a database, which, however, does currently not reach required quality.

In this thesis, we will start by assessing requirements on the modeler utility.

The portal is used not only by students but also by teachers and examinators. Different roles of users and states of the system will have to be considered during the whole development process. Next, we will analyse current state of the system. We will analyse how well it satisfies the assessed needs and identify its shortcomings. The analysis chapter will end by taking a look at similar tools for drawing diagrams and their approach to the user interface.

With analysis complete we will design new data modeler utility based on acquired knowledge. Extra care will be given to the usability and extensibility of this tool.

Second part of this thesis will be dedicated to implementation details where we describe architecture of the data modeler and technologies it is built upon. We will continue by exploring some of the more interesting implementation details and challenges that were faced during implementation.

After the implementation, new modeler will be properly tested. In a dedicated chapter we will provide results of both heuristic analysis and user tests, which were taken in real scenario. We will discuss findings of these tests and application of the results to the data modeler utility.

In the last chapter of this thesis we will suggest possible expansions and improvements of the tool and discuss their viability and importance for the students.

Chapter 1

Analysis

In this chapter we will analyse requirements for the resulting data modeler utility. First, we will overview the Database Systems course of bachelor programme at Faculty of Information Technology of Czech Technical University in Prague. We will get a better understanding of the course's contents and scope, as well as what requirements are made on students taking the course to successfully pass it.

Next, we will discuss how is the educational process supported by technical means, specifically the education support portal (DBS Portal[1]), what part it plays in the course's run and how it helps both students and teachers during semester and exam period.

With the acquired knowledge we will summarize both functional and non-functional requirements of the data modeler utility used for creating logical models. Proper understanding of needs of all roles of users during whole course's run is vital for proper design of the final product.

At the end of this chapter, we will assess the state of the data modeler utility that is currently deployed at the DBS Portal to see how well it covers the needs of its users. Additionally, we will explore user interface of Oracle SQL Developer Data Modeler, which is used as a primary software during Database Systems course, as well as some other, web-based, diagram modeling tools.

1.1 Database Systems Course

Database Systems course (DBS) is an introductory course to databases, aimed to teach students the basic concepts of database design and management. It is a mandatory course rated with 6 ECTS credits, recommended for second semester of bachelor programme. [1]

[1] Located at `https://dbs.fit.cvut.cz/`

During the course, students will get basic understanding of theory behind databases as well as practical experience for typical user roles, both designer and user. Although students will get an overview of the different database models used nowadays, the course is focused on relational database model, which is to this day still the most popular category of the database engines [2, 3]. At the end of the course, students will be able to properly design small databases from the ground up, starting with logical model of the real world, through the relational schemas used to describe properties of the data specifically for the target database engine with all integrity constraints and data types, to the creating final database in the engine itself. Students will grasp the basic rules of good database design. They will learn about normal forms and will be able to apply them to given relational model. They will also learn basic transformation principles between logical and relational schemas. To gain even more insight into a lower-level workings of database engines, students are taught basic principles of physical data storage, management of parallel access to shared resource and its problems and recovery from failure.

As users, students are required to understand Structured Query Language (SQL) as a querying tool of relational databases, as well as understand the relational algebra, the theoretical foundation of the SQL. They will be able to use SQL to define, create, manipulate and control database data.

To successfully pass the course, students have to hand in semestral work, pass the exam in semester and final exam during examination period. Goal of the semestral work is to develop smaller information system and fill it with test data. The topic of the work is not restricted, but the work itself has to contain:

- domain description
- at least 25 example queries in natural language
- logical schema
- implementation in selected database management system (while Oracle's database is preferred)
- SQL script which will fill database with example data
- at least 10 queries in relational algebra
- at least 25 queries in SQL, including data manipulation queries

Exam during semester consists of subset of problems found in final exam, while both are more focused on practical aspects of the course.

Nowadays, semestral work is submitted through dedicated web interface of DBS Portal. Likewise, exams are no longer written on paper, but are handled by this portal as well, which allows easier management, marking and is more convenient for both students and examinators.

From the standpoint of this thesis it is important that both semestral work and exams require students to understand and be able to create logical schemas of given problem. Thus, in this thesis we will further focus solely on the data modeler utility for creating logical schemas, in the form of diagrams.

1.2 Logical Data Model

Logical schema, or logical data model, is a specific view of a problem domain. While different logical models exist, we will focus on Entity-Relationship Model (ER Model), because this is the one that is being lectured during the course. ER Model, created by Peter Chen, is based on set theory and relation theory and adopts natural view of the real world which consists of entities and relationships between them, and incorporates semantic information about the real world [4].

Chen has defined his model to work on two levels of views of data: "Information concerning entities and relationships which exist in our minds" and "Information structure - organization of information in which entities and relationships are represented by data".

At first level he defines *entities* as distinctly identifiable "things" which are organized in *entity sets*. Each entity set has a predicate, which specifies whether entity belongs to given entity set. *Relationships* are defined as association among entities and they are organized in *relationship sets*, while entities may take a *role* in a relationship. Formally, each relationship set is defined as a mathematical relation among n entities, while each relationship is one tuple of entities.

Further information are conveyed via *attribute-value* pairs, while key realization is that they not only convey information about *entities* but also about *relationships*. Values are, again, organized in *value sets*, which allows different values from two distinct value sets to be equivalent (e.g. value 0 from value set CELSIUS is equivalent to value 32 in value set FAHRENHEIT).

At second level Chen defines how conceptual objects from the first level are represented. We consider the main take-away to be the definition of a *primary key*, which is such set of attributes, which "mapping from the entity set to corresponding group of value sets is one-to-one". That means that by looking at values of attributes of primary key we can uniquely identify the entity itself. Chen continues by defining *weak entity relations*, if relationships are used to identify entities and *weak relationship relation*, if entities are identified by other relationships.

Good understanding of this model and its concepts is required by students, however, the formal definition is not. Students are required to use concepts of this model in creation of visual diagrams.

In the following parts of this thesis, we will simplify the terminology and refer to "Entity sets" as "Entities" and "Relationship sets" as "Relationships".

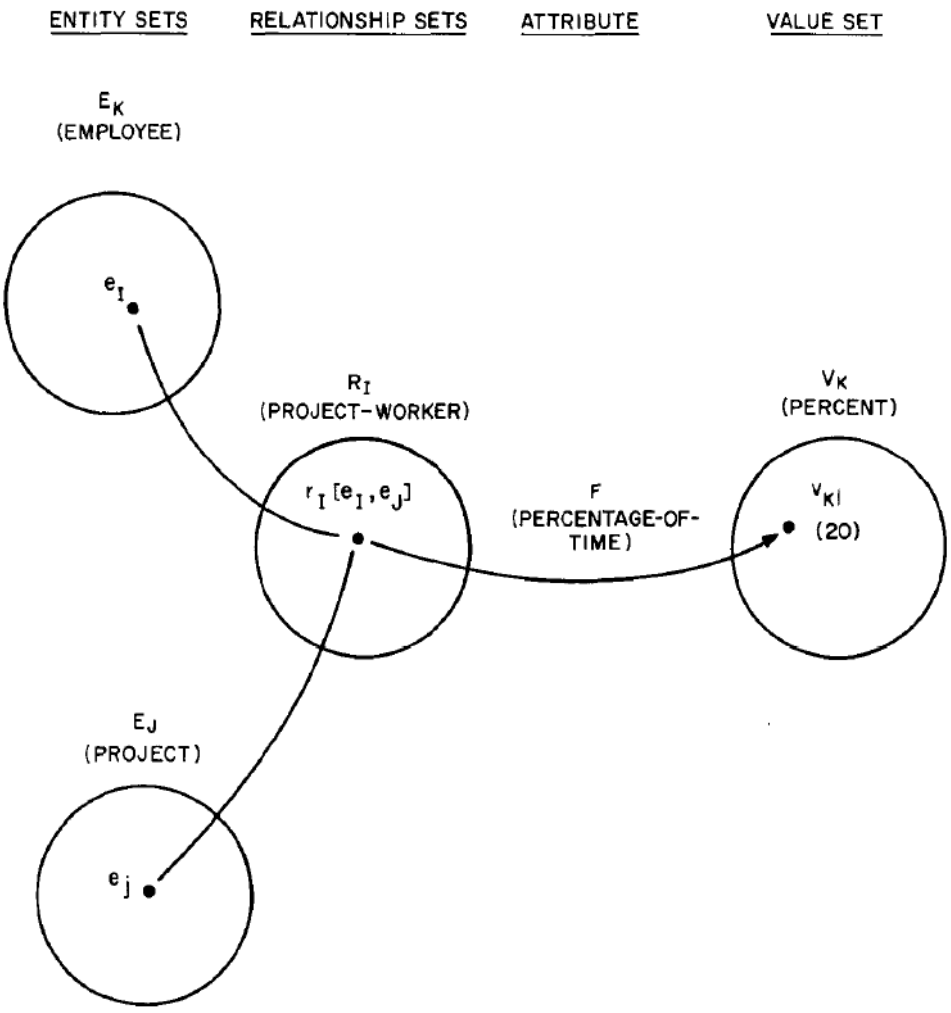


Figure 1.1: A visual representation of concepts of the ER Model as defined by Chen [4]. In this figure two entities from entity sets EMPLOYEE and PROJECT form a relationship from relationship set PROJECT-WORKER with attribute PERCENTAGE-OF-TIME.

1.3 Barker Notation

Although Chen has defined his own notation for drawing ER diagrams, our data modeler will use Barker notation, developed by Richard Barker in 1990, both because it is more commonly used in database world [5], but mainly because it is being taught during Database Systems course. This notation is also being used by Oracle SQL Developer, which is used as primary software during the course. Since original proposition of the notation may differ from commonly used practice, we will mainly refer to SQL Developer's dialect.

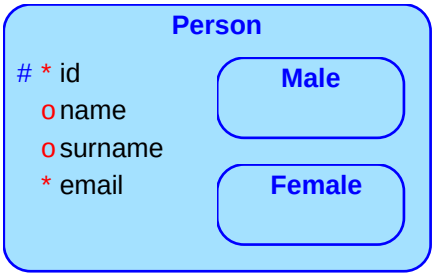


Figure 1.2: Example of entity with attributes and two subtypes created in SQL Developer. Entity *Person* is identified by a primary key *id* and described by optional attributes *name* and *surname* and mandatory attribute *email*. *Male* and *Female* are subtypes of the *Person* entity.

Barker notation represents entities as rectangles with rounded corners. If entity is a subtype of another entity, it will be drawn inside supertype's rectangle. Text at the center of the very top of the entity represents entity's name, while entity's attributes are listed bellow. Attributes are preceded by one or two symbols. Asterisk (*), sometimes heavy dot (•), indicates mandatory attribute (i.e. NOT NULL), while circle (○) indicates optional attribute. Attributes preceded by hash symbol (#) are part of the primary key of this entity. SQL Developer also uses letter U to indicate unique attributes that are not part of primary key. Although this extension was not defined in original notation proposal, we will also cover it.

Relationships are shown as lines between two entities. Relationships are binary, meaning they strictly have source and a target, albeit source and target may be the same entity, effectively creating recursive relationship. Relationships are split in halves, which may be named. Each half of the relationship reflects optionality and cardinality of the role in the relationship for given entity. Optional relationship is drawn as dashed line, as opposed to solid line for mandatory relationship. Cardinality is represented by crow's foot notation, where crow's foot represents *many* and simple line represents *one*. Vertical bar symbol (|) at one and of the relationship is used to reflect that the relationship

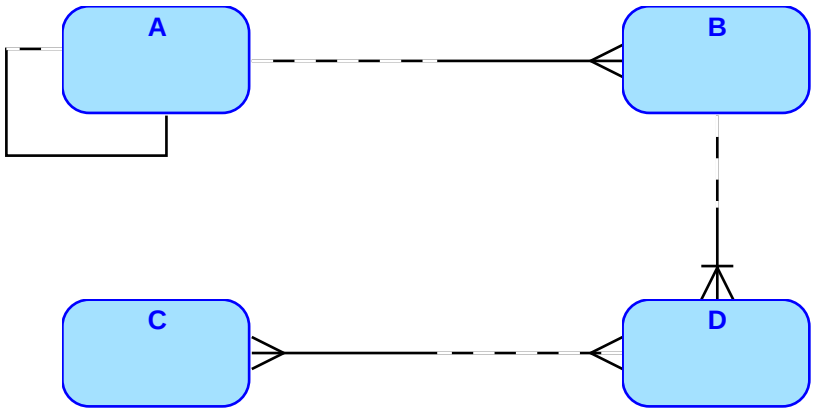


Figure 1.3: Example of relationships with different cardinality and optionality as well as example of identifiable relationship, created in SQL Developer

takes part in the identification of the entity.

Finally, an arc across relationships of one entity represents Exclusive OR (XOR). This constraint says that if two or more relationships are in one arc, entity may (or must) "use" only one of these relationships.

Compared to Chen notation, Barker notation has two significant drawbacks. Although Chen's ER model (and notation) allows relationships to have attributes, Barker notation does not. These relationships have to be decomposed and new, "artificial", entity has to be created. Second drawback stems from the fact that Barker notation is binary, and thus each relationship can be made only between two entities, unlike in Chen notation which does not have this restriction.

1.4 Data Modeler

In section 1.1 we have established that student is tasked with creation of logical models for semestral work and during exams. In section 1.2 we have discussed basic concepts of the ER model and finally in section 1.3 we have discussed how the ER Model is represented in a form of diagram.

In this section we will use acquired knowledge to specify requirements of the data modeler in regards to different roles of users that will be using it. We will also cover current state of the utility deployed at DBS Portal and analyse how other tools are handling data modeling.

1.4.1 Functional requirements

The output of the data modeler is a visual 2D diagram. It is clear from previous analysis that data modeler has to recognize three main objects: entities, attributes and relationships. First, we will cover the full feature-set and functional requirements, after which we will discuss how does the functionality change in different contexts. We have derived following functional requirements, which have been grouped together by the object they are related to:

Entity

Entity is the first object user will create. Entities have attributes and they take part in relationships.

E.1 Entity may be created and deleted

E.2 Entity has a name, which can be edited

E.3 Entity may have attributes

E.4 Entity may take part in a relationship with other entities

E.5 Entity may be a subtype of another entity

E.6 Entity may be identifiably dependent on a relationship

E.7 Position on the canvas and visual size of the entity may be changed

E.8 Entity may be marked as incorrect

Attribute

Attributes convey additional information about entities and their properties. Visually they are displayed as a list inside the entity.

A.1 Attribute may be created and deleted

A.2 Attribute has a name, which can be edited

A.3 Attribute is associated with one and only one entity

A.4 Attribute is either optional or mandatory

A.5 Attribute may be a part of a primary key of given entity or it may be marked as unique

A.6 Order of attributes may be changed

A.7 Attribute may be marked as incorrect

Relationship

Relationships in Barker notation are strictly binary and drawn as a line.

R.1 Relationship may be created between two entities – source and target. Source entity may be the same one as target entity. Relationship has two distinct halves, one attached to source entity, the other to the target entity

R.2 Each half of the relationship may have a name, representing a role of the attached entity, which can be shown or hidden

R.3 Name of the relationship half may be edited

R.4 Each half of the relationship is specified by partiality (optional/mandatory) and cardinality (one/many), which may be changed

R.5 Relationships of one entity may be grouped into Exclusive OR. Relationships may be added to XOR relationships and removed from them

R.6 Control points for each half of the relationship may be created, moved and removed. Control points specify how the relationship will be drawn. Relationship is drawn as a straight line between each two successive control points

R.7 Relationship half may be marked as incorrect

Diagram

Diagram requirements relate to whole diagram. These requirements are not manipulating model directly, instead they support modeling process.

D.1 Diagram may be imported and exported

D.2 Diagram may be automatically laid out ("sorted") in a way that would minimize overlap of objects and maximize readability of the diagram

D.3 Diagram may be zoomed in or out

D.4 Diagram has a fullscreen mode

D.5 It is possible to turn on or off certain functionality of the diagram, depending on the user role or context in which the modeler is being used

Although DBS portal defines multiple user roles including student, guarantor, lecturer, teacher and so on, previous analysis has shown that data modeler will be used only by student and examinator and finer definition of

user roles is not needed. Instead, there is a need to restrict certain functionality depending on the context in which the data modeler is being used. Student will work with data modeler during his work on semestral project and during exam, while examinator will be using data modeler to create reference solution of the test and review diagrams made by students during exams or tests.

To simplify following discussion about available functionality in different contexts, we will categorize functional requirements: *Data* type requirements directly manipulate or change data of the underlying ER Model. *Visual* type requirements only change how does the model's diagram look like, but they keep data untouched. *Review* type requirements are meant to let the student know what part of the diagram is incorrect and, last, *file* type requirements work with physical files.

Free creation mode

This mode will be applied in two contexts, for student when working on semestral project and for examinator when creating reference solution. In this case, data functionality of the modeler does not need to be limited and user can use full feature-set of the modeler, along with saving the model and continuing from the previous state. The only restriction is *review* functionality, which does not make sense in this context and thus it should be turned off.

Student during test or exam

In this context, student should not be able to export or import any type of data, to prevent cheating. Saving of the resulting diagram will be handled in the background by DBS portal. Students also can not review their own work. To sum it up, only *data* and *visual* functionality is allowed in this context.

Diagram review

After the test, diagrams created by students are reviewed by examinator. Examinators can not be able to change underlying data model, they are only allowed to mark errors. To make the reviewing process easier, they can change the appearance of the diagram. Thus, in this context, only *visual* and *review* functionality is allowed.

1.4.2 Non-functional requirements

We have to keep in mind that data modeler is not a standalone application, but only a component of DBS portal. It has to be interactive, real-time and run in web browsers. We have concluded following requirements:

N.1 Data modeler is implemented in JavaScript, HTML and CSS so the integration into DBS Portal is possible

N.2 It is possible to display larger number (150+) of diagrams at one page. This requirement will ensure as simple as possible review of diagrams after exams

N.3 It is possible to use data modeler on resolutions as low as 1024x768

N.4 Data modeler is able to import logical model from Oracle SQL Developer

N.5 Data modeler is able to save the model in a textual form. Equivalent model will always output same textual representation. Model is considered to be equivalent with another model if it fulfills following conditions:

- Each entity in the first model has equivalent entity in the second model. Entities are equivalent if names of the entities are equal and both entities have equvivalent attributes, while order of the attributes does not matter. Attribute is equivalent with another attribute if they have same name, optionality, uniqueness and whether they form primary key or not
- Each relationship in the first model has equivalent relationship in the second model. Relationships are equivalent if both relationship halves are equivalent, however source and target distinction is not important (e.g. source in first model may be equivalent to target in second model). Relationship halves are equivalent if attached entity is equivalent, cardinality, optionality and identifiability of both halves are the same and both halves form Exclusive OR relationship with equivalent relationships

N.6 Data modeler has to be able to display equivalent diagrams in exactly the same way (i.e. with same placement of entities and relationships, same order of attributes, same capitalization of names etc.) in same browser session. Let's consider two equivalent diagrams. If one is displayed in Firefox and other in Chrome, they need not to be displayed in the same way, however when both are displayed in the same browser during one browser session, they have to be displayed in exactly the same way.

N.7 Data modeler has to work correctly and fully in Chrome version 55 and Firefox version 50 and newer

1.4.3 Current State

Current version of the data modeler was developed by Jiří Slavotínek as his bachelor's thesis project. Since the state of the modeler did not change after the project was completed, we will use the thesis itself [6], supervisor's evaluation [7], reviewer's report [8] as well as the component itself to review its current state.

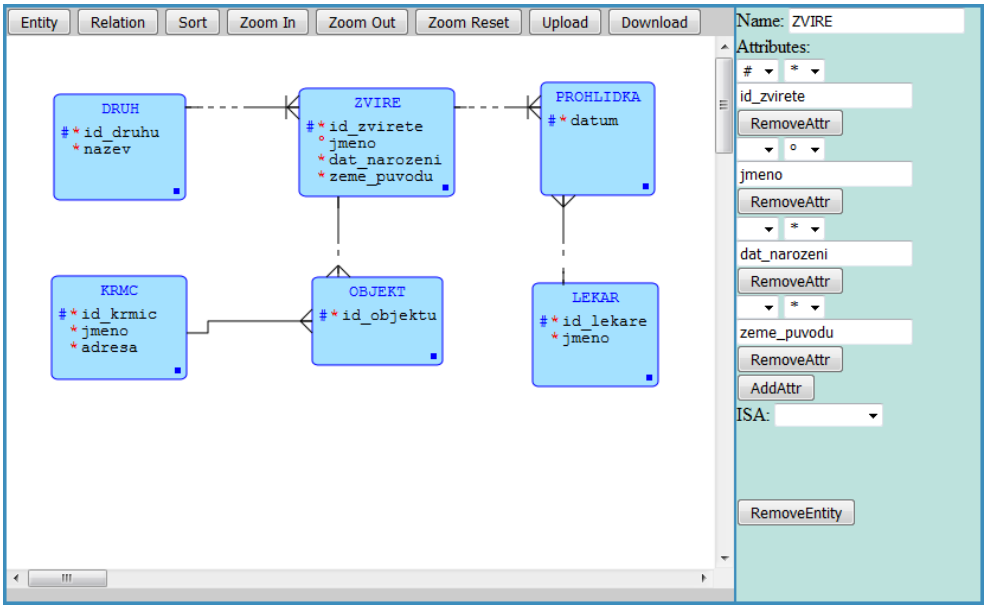


Figure 1.4: Sample model created in current data modeler. Entity ZVIRE has been selected. In this screenshot you can see main control elements – toolbar and property window at the right side. If no object (entity or relationship) is selected, property window is empty.

Area of the data modeler is vertically split into two distinct parts. Left side takes about three quarters of the area and forms a canvas, where the diagram is being created. At the top of the canvas we can find toolbar, which contains actions for creating objects (Entity and Relation buttons) and manipulating with diagram itself (sort, zoom and file actions). Right side of the screen is used as object property window.

To create an entity, user clicks the *Entity* button in the toolbar. Placement of the entity is handled by the data modeler and its size is predefined. When selected, more options are shown in property window on the right side of the modeler. Here, user can change entity's name, edit attributes, create ISA hierarchy by setting parent of the entity or remove entity.

Relationships are created in similar fashion. First, user has to click the button in the toolbar and then select two entities in the diagram by mouse click. Once the relationship is created, it is automatically selected and its properties may be changed in property window. More options are unveiled when mouse is over the relationship. User is allowed to drag each end of the relationship to another entity or delete relationship altogether.

Author in the thesis expresses his satisfaction with the result, but in the same breath adds that due to his inexperience with JavaScript the im-

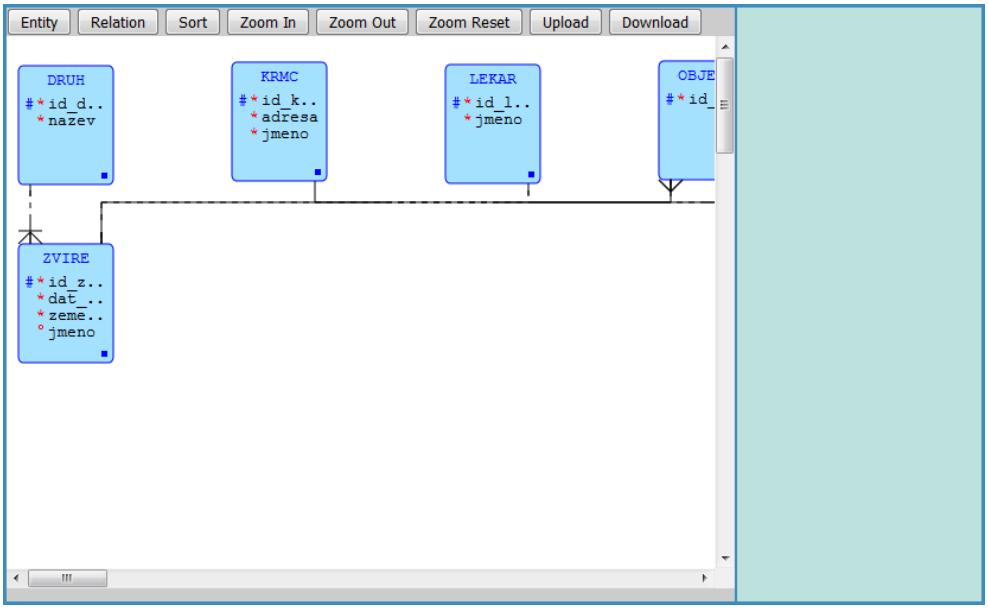


Figure 1.5: Automatically sorted diagram from figure 1.4. Several issues can be seen from this screenshot. Entities are resized to default size, which is too small for names of attributes and are therefore truncated. Relationships are overlapping and can not be distinguished from each other.

plementation is lacking. He also admits that another betterment could be made in the user interface front.

Supervisor in his evaluation voices his concerns about the quality of the code and possible future development of the modeler. In his opinion, the biggest weakness of the utility is the user interface, which, however, was not part of the assignment. The reviewer's report has similar tone. Missing functionality – Exclusive OR relationship – is criticized and he finds the interface hard to work with, especially when manipulating with relationships. Another major flaw is found in the sorting functionality, which tends to produce unexpected states with hardly distinguishable details of the diagram.

Missing ability to create XOR relationship is not the only missing functionality. Data modeler in its current state further lacks reviewing functionality, therefore the examinator is not able to mark incorrect parts of the diagram. Attributes' order can not be changed either, user needs to write attributes in the correct order or rewrite the whole list of attributes. When attribute is added, the name input field does not take focus and requires additional action from user.

Placement of relationships in the diagram is managed by the data modeler, which, however, often leads to creation of overlapping relationships. When edit-

ing relationships, it is not clear which half of the relationship is source and which target, so proper set up is often a trial and error process.

Data modeler is also lacking fullscreen support and functionality is restricted by manually changing the HTML code for the toolbar. No obvious support for multiple diagrams on a single page was found. Bugs during import of the diagram from SQL Developer were also discovered, the modeler seemed to have problems with correctly displaying optional attributes and attributes of parent entity were displayed in both parent and child when entities were in ISA hierarchy.

Subjectively, author of this thesis finds work in current implementation of the data modeler laborious. User needs to constantly move his attention from one side of the screen to another and there is very little control about appearance of the diagram. Resizing of the entity is often difficult, since the reaction area is very small. Moreover, feedback is in some cases non-existent and user has no idea about what state the modeler is currently in, or what action he should take next, which is most apparent during creation of relationships.

Due to found design flaws and overall quality of the implementation we find the best solution for further development to be to start from the ground up and completely redesign the data modeler.

1.4.4 SQL Developer

Oracle SQL Developer is full-featured IDE for database development and management, currently in version 4.1.5. It "offers complete end-to-end development of your PL/SQL applications, a worksheet for running queries and scripts, a DBA console for managing the database, a reports interface, a complete data modeling solution, and a migration platform for moving your 3rd party databases to Oracle" [9]. Although during the Database System course students will use more of its features, in this thesis we will focus solely on Data Modeler.

Data Modeler in SQL Developer allows creation of different models of the database, two most important for the topic of this thesis are logical model and relational model. Users are able to transform one to the other. Relational model creation and transformation is one of the possible future expansions of our data modeler utility, and therefore out of scope of this work. However, interface and way the software works is identical or very similar for both types of models.

By default, user interface consists of three parts. At the top is menu and toolbar. Menu contains options related to the application itself, toolbar contains icons with actions related to current view. Left side contains various widgets, which the user can hide, show, move to different part of the view or close. These can be various browsers, for e.g. database connections or objects created in current diagram.

Requirement	Notes
E.8	Not implemented
A.6	Not implemented
A.7	Not implemented
R.5	Not implemented
R.6	Partially implemented; it is possible to create control points but the way the relationship will be drawn is hard to predict
R.7	Not implemented
D.2	Partially implemented; it is possible to sort the diagram, but the implementation produces many overlaps
D.4	Not implemented
D.5	Only possible by removing parts of HTML code of the toolbar; it is not possible to restrict editing of object's properties
N.2	No native support has been found
N.4	Partially implemented; it is not possible to import XOR relationships due to missing functionality
N.5	Partially implemented; import to JSON is available only from code, download functionality is trying to mimic SQL Developer format. Quality and comparability of these exports was not analysed
N.6	Partially implemented; relationships tend to be drawn differently

Table 1.1: Overview of requirements not implemented or not implemented fully in current data modeler

Entity is created by selecting action from the toolbar and then either clicking or clicking and dragging ("drawing") on a "canvas", the third part of the screen which occupies the majority of space. Drawing method allows user to also specify the size of the entity already during its creation. To add attributes, user needs to either double click on the entity or select "Properties" from context menu. This will open another window, where user can completely set up all properties of the entity, which are grouped into categories. Name of the entity and super type (for creating ISA hierarchy) can be changed in the first, *General* category. Attributes are managed in *Attributes* category, where they can be also set as a part of primary key. Unlike primary key, to set attribute as unique, *unique key* has to be created first in a separate section, after which, in property window of this key, attributes can be added to it.

Creation of the relationship is easy, user can select one of four types of relationships (M:N, 1:N, 1:N Identifying, 1:1) from the toolbar and then click on two entities. Upon creation of the entity, new *relation properties* window

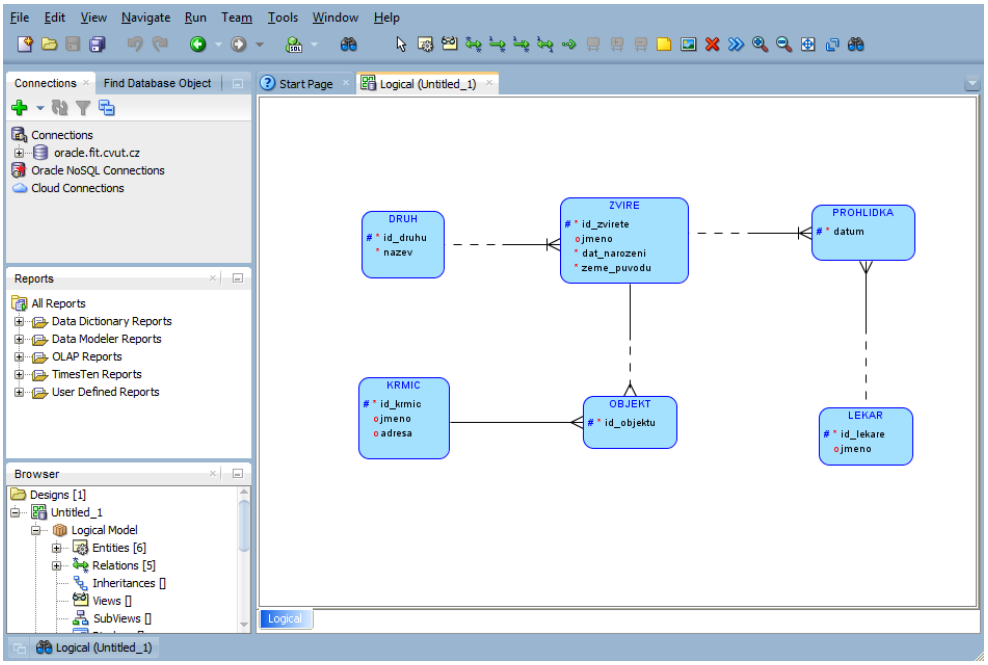


Figure 1.6: Same diagram as shown in figure 1.4 drawn in SQL Developer Data Modeler with default setup of the user interface

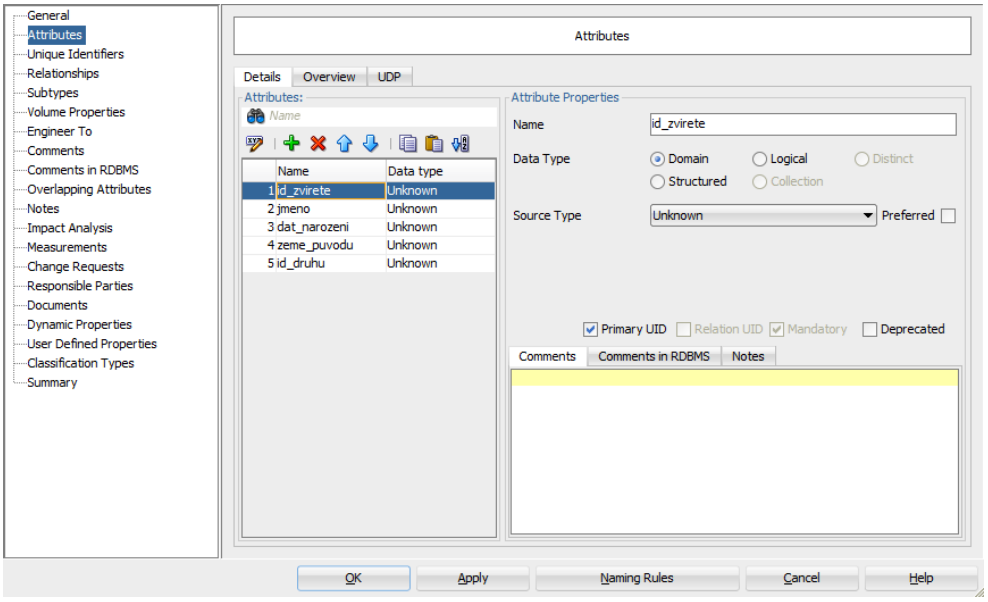


Figure 1.7: Example of properties window of SQL Developer Data Modeler. In the screenshot we can see the Attributes section of entity properties window.

is automatically opened, from which user can further set up properties of relationship.

To create an Exclusive OR relationship, user selects (either by clicking and dragging with select tool selected or by holding *Ctrl* key and clicking on objects in canvas) one entity and all relationships that should be part of XOR. If the selection is valid, *Arc* button, which creates XOR, will be activated in the toolbar.

SQL Developer Data Modeler is not very easy nor intuitive to use. That is caused by the sheer number number of functions this software offers. Even the most simple actions may take a lot of steps, often through multiple windows. Sometimes the functionality is disabled and it is not clear how to enable it, which is the case of XOR, for example. On the other hand, relationship creation is handled quite well since user can choose what type of relationship he wants to create and therefore no additional setup may be needed (which is however also suppressed by the fact, that the property window is automatically opened and has to be closed, even if the relationship works as is). The robust design of SQL Developer works well for big projects but modeling smaller diagrams of few entities that students will encounter could greatly benefit from simpler interface. On the other hand, students are already used to work in SQL Developer and the design of DBS Portal data modeler should reflect that.

1.4.5 Online Data Modelers

Last part of the analysis is devoted to analysis of user interface design of online data modelers. Four web-based modelers have been selected.

1.4.5.1 Creately

Creately [10] is an online modeler with support for all kinds of diagrams, mind maps, trees, infographics and various charts, written in Flash. For ER modeling it uses Chen notation for attributes but crow's foot for relationships. Toolbar with universal diagram actions (open, new, save, undo etc.) is located at the top of the screen. Rest of the screen is divided into three parts. Shape picker is located on the left side, canvas for drawing diagram is in the middle and takes majority of the screen and an expandable bar with properties, notes, comments and other tabs can be found on the right side.

Objects are created by dragging them from shape picker into canvas or from overlay menu of each object – when object in the canvas is selected, it will display menu above itself. Actions in the menu depend on the type of the object that was selected, e.g. entity contains following actions: Edit Text (allows editing of Entity name), Connect (starts drawing line from this entity), Order (moves objects in the canvas to front or to back), Link to Diagram or

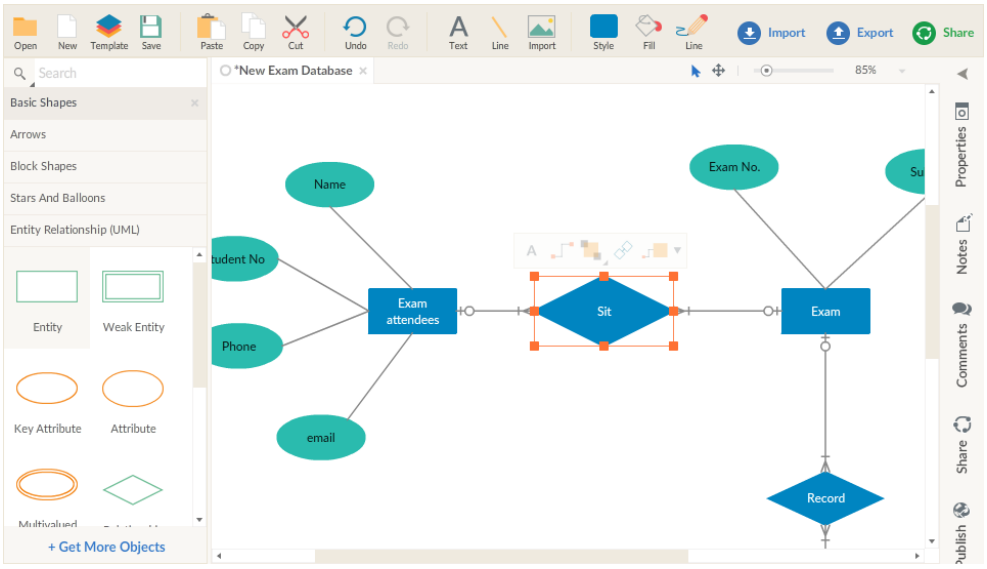


Figure 1.8: User interface of Creately with example ER Diagram. Relationship *Sit* is selected with visible menu above the selection.

URL, and Create Relationship. More options are hidden in under a caret icon, from where e.g. ISA hierarchy can be created.

Overall, drawing diagrams in Creately is easy. However, major drawback is that Creately is universal diagramming tool, and in general does not know about objects themselves. For example, it is possible to draw a line between two attributes. Control of the objects is good, user can move objects freely on the canvas and resize them, it is even possible to set absolute position and size in the properties window. That can not be said about relationships though. Their placement on objects is very restricted, each object has defined points where the relationship can be drawn from. In case of entity, there are four points, so if the entity has to have five relationships they will overlap.

1.4.5.2 Vertabelo

Unlike Creately, Vertabelo [11] is specifically build to support database development. Among its features we can find collaboration support, versioning, SQL generation, live validation, search and more [12]. First major drawback of Vertabelo is its support. Vertabelo only works in Google Chrome and Safari. Screen setup is familiar, with the toolbar at the top, *model structure* window on the left, properties on the right and canvas in the middle. Vertabelo builds relational models – support for logical modeling was not found at least in preview version of the application.

Entity is created when user clicks on the canvas while "Add new table" is

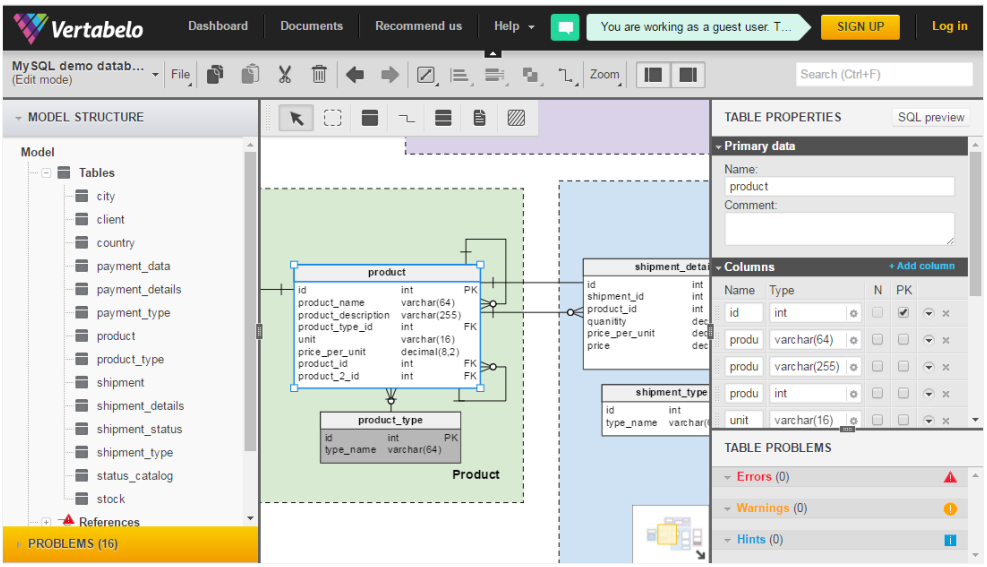


Figure 1.9: User interface of Vertabelo with example ER Diagram. Entity *product* has been selected and its properties are visible on the right side of the window.

selected from the toolbar. It is not possible to set size of the entity by holding left mouse button clicked and moving the mouse, entity is always created with the same default size. Upon creation, entity is automatically selected and can be resized or its properties may be changed, including addition of attributes, from the panel on the right. Relationship is created with, again, appropriate tool selected and then clicking and dragging from source to target entity.

Interface is similar to current DBS portal data modeler, however, it is much more mature, albeit it might be considered to be somewhat "claustrophobic" due to amount of options and panels. We consider the biggest weakness to be the need to shift focus and attention throughout whole screen when setting up entities and relationships. Relationships do not allow creation of control points and they are managed mostly automatically. On the other hand, model structure which shows clearly organized all objects in groups is useful feature, however, it is questionable whether it would be useful also for smaller models made during Database Systems course.

1.4.5.3 Draw.io

Draw.io [13] is open-source universal diagramming tool. It supports wide variety of shapes, however, same as Creately has no understanding of the modeled data. Interface may be very familiar to users of Google Documents[2],

[2] `https://docs.google.com/`

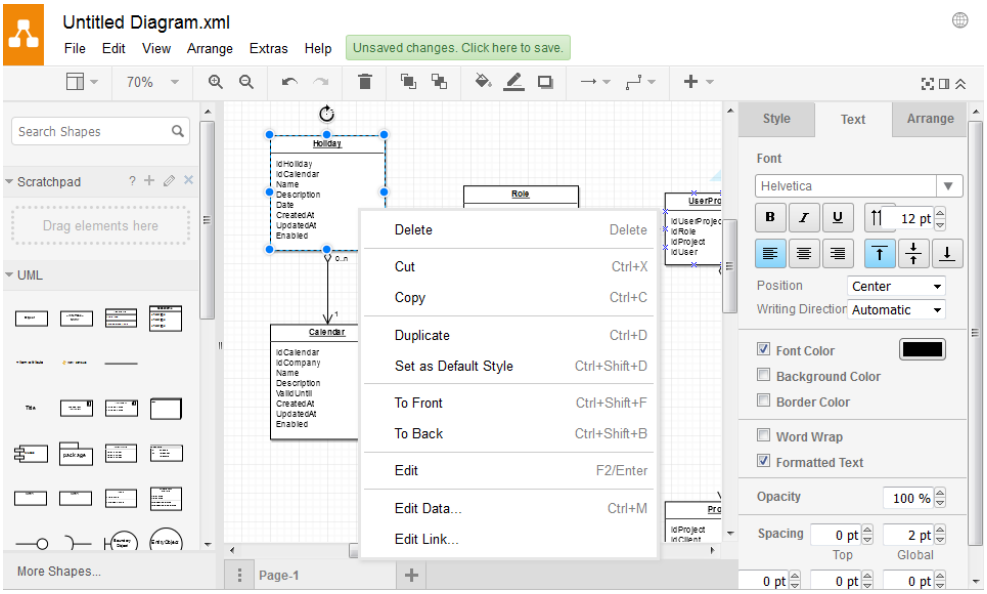


Figure 1.10: User interface of Draw.io with context menu for *Holiday* entity open.

since it is apparent Draw.io's interface was inspired by Google. To draw an entity, shape is dragged from the shape browser into canvas. Entity's content is simple text, so the user has to be wary of the correct format. No data validation is possible due to the nature of the application.

Relationships are drawn from the center of one of the four sides of an entity. When mouse is over the entity, little arrows and control points are displayed. When either of them is clicked and dragged, relationship starts to be drawn. When there is a relationship drawn from the center of the edge already, the arrow stays hidden, in which case the relationship has to be drawn from specific control point. Relationship themselves are just mere lines with a lot of style options for both line and its ending and the modeler allows also lines that do not connect to the entity.

Interesting feature is context menu of drawn objects which is typical for desktop software but not so much for web applications. Context menu can be open via right click and contains easily accessible options, e.g. delete.

1.4.5.4 TinyModeler

Last tool, Tiny Modeler [14] is feature-wise the simplest one out of the compared alternatives. It is not limited to database modeling and can be used e.g. to create class diagrams. Its interface is also the most minimalistic one out of all compared tools, there is only a toolbar at the top of the window and

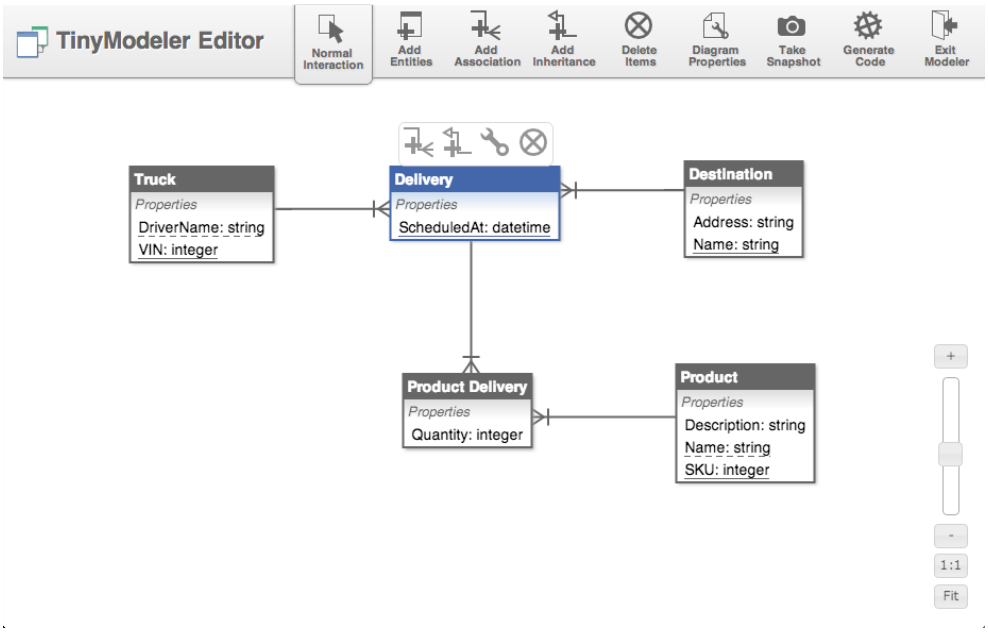


Figure 1.11: User interface of TinyModeler with small diagram and overhead menu displayed for entity *Delivery*. This screenshot was taken from TinyModeler's website [14].

the rest is left for canvas.

To create an entity, user needs to select *Add Entities* from the toolbar and click on the canvas. Entities can not be resized at all (size of the entity is handled automatically), but can be freely moved. Relationships are created either from toolbar or directly from entity, which has overhead menu displayed on mouse over. This menu contains four actions represented by icons: new relationship (called "association" by this modeler), inheritance, properties and delete. When entity is clicked, property window is opened. From this window user can add attributes, change entity's name and set reflexive relationship. Relationships have similar properties window, from which cardinality, identifying relationship or a label may be set.

While the interface is very simple, it has fundamental flaws. First of all, consistency is an issue in at least two places. First, relationships between entities are created differently than recursive relationships. Not only creation is the problem, but also their set up. Relationships between entities have their own property window, while recursive relationships are set up strictly from entity. Second consistency issue is with the menu above entities. First two actions (new relationship, new inheritance) are meant to be dragged to target entity, while other two actions (properties, delete) are simple buttons.

1.4.5.5 Summary

We have explored four web based data modelers and their solution of a user interface design, from the minimalistic TinyModeler to feature-rich Vertabelo. Although all four modelers allow creation of diagrams, they solve different problems. Creately and Draw.io are general-purpose tool that allows creation of basically any type of diagram, but because of that they do not know anything about data they model. On the other hand, TinyModeler focuses on domain modeling not neccessarily just for database development and Vertabelo is mature database design tool.

As well as the problem they are solving, their approach to user interface also varies. Entities were drawn either by dragging a shape into canvas (Creately, Draw.io), or by using dedicated tool and clicking at the place where the entity should be drawn (Vertabelo, TinyModeler).

Relationships could be drawn from menu shown above entity (Creately, TinyModeler), from the entity's edge (Draw.io) or by dedicated tool (all but Draw.io). However, we did not find relationship handling to be satisfying in any of the tools. Usual problem is restricting the manual control over how the relationship is drawn.

All modelers offered some way to set up additional properties of the drawn object, whether it was by popup window or by dedicated panel.

Chapter 2

User Interface Design

In this chapter we will focus on user interface design. We now know what work is being done in the Database Systems course and what is required of the data modeler. We have also analysed the interface of the current data modeler, SQL Developer Data Modeler and other online tools, which should provide good base for laying out familiar looking and easy to work with interface. First, we will discuss design principles that have been attempted to be achieved, then we will look at design itself.

2.1 Principles

In the user interface design, user comes always first. When working in a stressful environment like exams often are, where users are pressed by time and have a (maybe unfamiliar) problem to solve, it is even more important to design an interface, that will be easy to understand and simple to use. Minimalism, simplicity and efficiency were all the staples that were strived for.

Minimalism meant minimization of any kind of toolbars, windows or views. Interface should be as clean as possible, non-intrusive and provide minimum distractions, yet to still show users all they need to see. Good interface design should draw as little attention to itself as possible.

Simplicity principle was kept in mind to prevent so called "feature creep" or "scope creep". Feature creep is a name for process of ongoing addition of new features and broadening the scope of the project, which may cause losing the focus or shifting the emphasis from what are critical parts of the software. Only the subset of features which are necessary for either accomplishing given tasks or those that allows users to use the software more efficiently and effectively should be kept and focused on.

Inspired by Asimov's laws of robotics, in the book The Humane Interface [15] Jef Raskin defines two laws of user interface design. Second law[3] says

[3]Raskin's first law is "A computer shall not harm your work or, through inactivity, allow

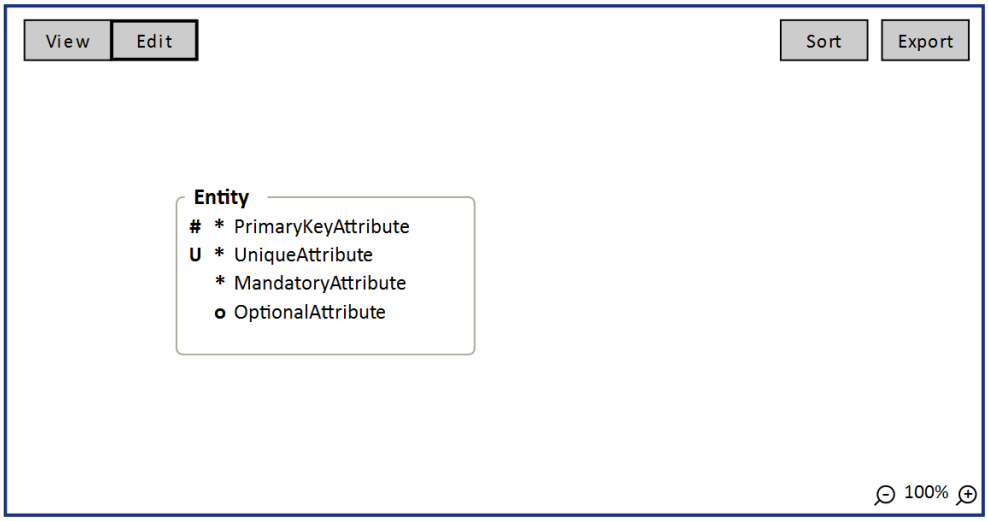


Figure 2.1: First version of the user interface of new data modeler

> A computer shall not waste your time or require you to do more work than is strictly necessary.

In accordance to this law, last of the main design principles is *efficiency*. User should be able to achieve their goal in minimal number of steps, clicks or key presses and all possible actions should be always easily accessible.

2.2 First Design

On the very first look, the first design did not differ from the final one in too many ways. The whole screen of the data modeler works as canvas and buttons are located in three of its corners.

Top left corner contains mode switching options. In the view mode user can not make any changes, neither visual nor to underlying data model, but can zoom or move canvas. In edit mode all editing capabilities are permitted.

Top right corner contains diagram options: sort and export. Sort runs automatic layout of the diagram's objects, export saves the underlying model in a text format. Import works by dragging a file onto canvas, which was kept in final design.

Bottom right corner contains zoom controls. Common symbols for zoom are used – magnifying glass icons with minus and plus. In the middle, between the icons is a percentage showing current zoom level. Click on the percentage resets zoom level back to 100%.

your work to come to harm."

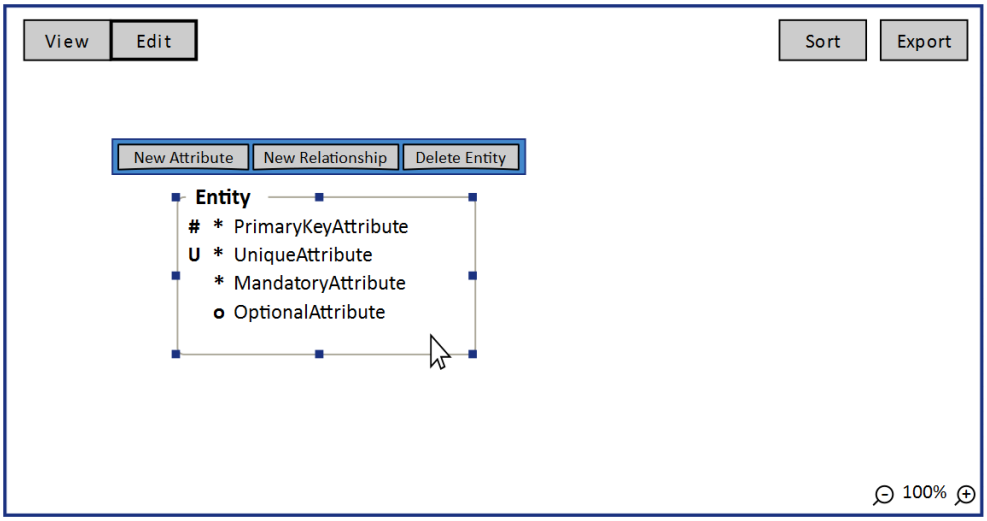


Figure 2.2: Entity menu of the first version of the interface. Both menu and resize controls are shown when the mouse is located over the entity.

Holding true to the minimalism principle we set for ourselves, we wanted to minimize toolbars and various panels. However, user needs to be able to interact with objects. Solution for this problem was inspired by Creately and TinyModeler, which use *object menus* – menus shown above the object which has mouse pointer over it, usually showing icons. We find the advantage of this menu style to be its simplicity and placement, since user does not need to travel to other parts of the screen. Moreover, user always knows what object is being edited, since the menu is shown directly above it.

Unlike Creately and TinyModeler, we have extended object menus to all objects: entities, attributes and relationships. This was achieved by ability to show menu not only above the object also on the left and right, as can be seen in figure 2.3, where editing of an attribute is being demonstrated. We can see both entity's menu with options to create new attribute, new relationship and delete entity as well as attribute menus, one on the left side used for set up of attributes properties and one on the right side containing option to delete attribute. Backgrounds of the object menus can be color-coded.

Entities are created either by single click on the canvas, which would create entity of the preset size, or by clicking and dragging ("drawing"). Entity's name is edited in-place, *double click* on the entity's name shows text input which the user can edit. New name is confirmed by *Enter* key. When mouse cursor is over entity, resize control points at the borders of the entity become visible and menu above entity is shown. To resize an entity, user clicks on one of the control points and drags it, to change entity's position user drags the entity itself in similar fashion.

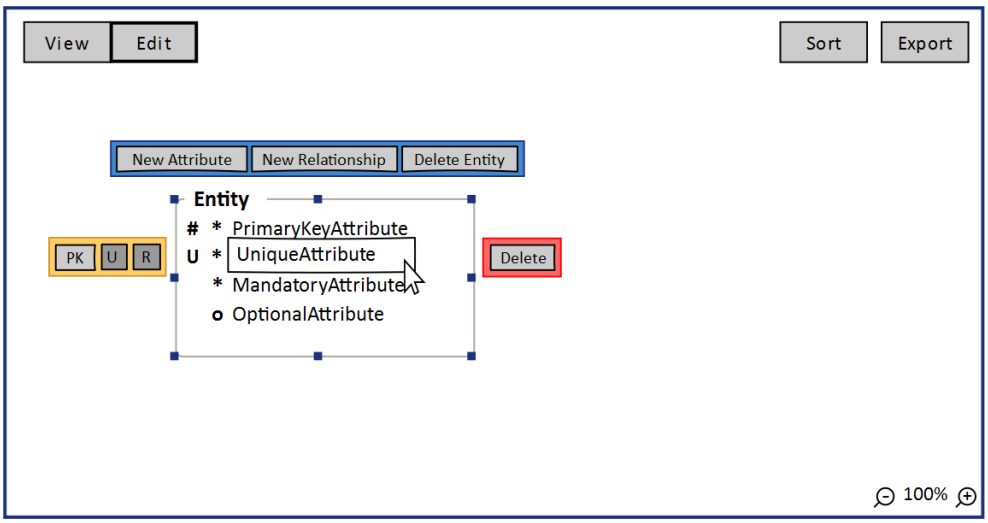


Figure 2.3: Attribute setup in the first version of the interface

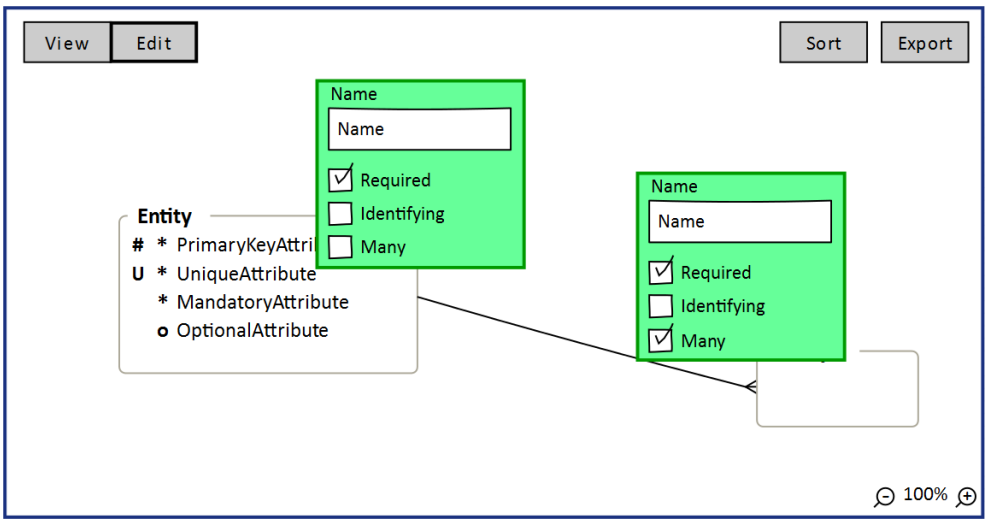


Figure 2.4: Relationship setup in the first version of the interface

New attributes automatically enter edit mode and the input field for attribute name takes focus. Attribute names are edited in-place, same as entity names. Attribute's position may be changed by dragging the attribute in the entity.

Lastly, relationships are created also from the entity's menu. When relationship is being created, line from the source entity is being drawn until user clicks on the canvas to cancel it or on the target entity to confirm creation. To set up relationship, user clicks on it, which shows menu above each end of the relationship. From this menu, user can set role (relationship name) and change other properties (identifying, required, cardinality). The line which represents relationship may be changed by dragging it, which automatically creates new control point. Also, the end of the relationship may be dragged, however, the movement is restricted to the entity and it can not be dragged to another entity.

2.2.1 Evaluation

Some of the problems of this design were hinted by lo-fi prototype, but they became truly apparent in hi-fi prototype. We will discuss found problems in no particular order.

First problem was *consistency*. Entity menu contained only buttons, attribute menus contained buttons and switches and relationship menus contained only textbox for editing role of the relationship and checkboxes. Placement of the menu was also an issue, since it requires inherent knowledge of the system from users. They have to focus on what type of object they are editing, since each object has menus at very different place. Lastly, although not the major issue, entity and attribute menus were shown when mouse was over an object and relationships required click, however, this could be easily fixed by showing all menus on click.

Furthermore, it was problematic to work with objects' menus when objects were close to the edge of the canvas. Menus could overflow out of the canvas and some or all options were not accessible.

When considering future extensibility, the menu system did not provide any room for more options. It could be condensed a little bit by using icons, however, that brings another set of problems and further requirements on the user, where he would have to figure out what each icon means and then remember it.

One of the strengths of this design is that maximum area of the screen is left for diagram drawing itself. No space is taken by additional windows or views. In-place editing of entity and attribute name is also convenient, however, double click may be unnecessary and single click could work at least as well, if not better. This system should be also used for relationship roles, to ensure consistency.

2.3 Final Design

To solve the problems of the first iteration, menu system has been completely thrown out and replaced, better and more modular system has taken its place. Canvas window was also further simplified and additional functionality which will help new users was added.

Canvas now only have icons in upper right corner. From right to left, first icon is new *info* button. This button toggles informational window containing shortcuts overview. Shortcuts are often used by power users, but there is a problem of how to convey the information about available shortcuts. Therefore, one encompassing and minimal view with available shortcuts was created.

Second from the right are zoom controls. Functionally unchanged from the first iteration, there are two icons with current zoom level in between. Only difference is that they were moved here from the bottom right corner.

Last button, which will only be available for examinators, is a switch to *marking mode*, in which examinator will be able to mark incorrect parts of the diagram. View and edit modes from first iteration were completely abandoned, since they were not really needed, because the option to create entity by single click was removed.

Another change from the first iteration is entity creation, which can now be created only by click and drag. Click-to-create option was removed because evaluation of previous prototype showed that new entities could be easily created when user did not mean to, i.e. misclicks. Along with that, entity is now selected on mouse click, after which control points for resize are displayed. This change was made because in last prototype control points were often hidden when user moved cursor a little bit too much away from the entity, which in turn made resizing a much harder than it should have been.

The biggest change, however, is already mentioned revamp of the menu system. Proprietary menu style was removed in favor of typical context menus, which are shown on right click and have, compared to previous solution many advantages:

- Context menus are typical in all current OSs and users are accustomed to them, in some situations they even expect context menus. This may not hold for regular users but each user that will use this data modeler is expected to have above average computer experience. However, this presumption will still need to be tested in proper user test.

- Addition of more options is no longer a problem. Context menus are easily vertically as well as horizontally (via submenus) scalable.

- Menus are shown at a place where user clicks, which minimizes cursor travel, especially in case of long relationship or large entity.

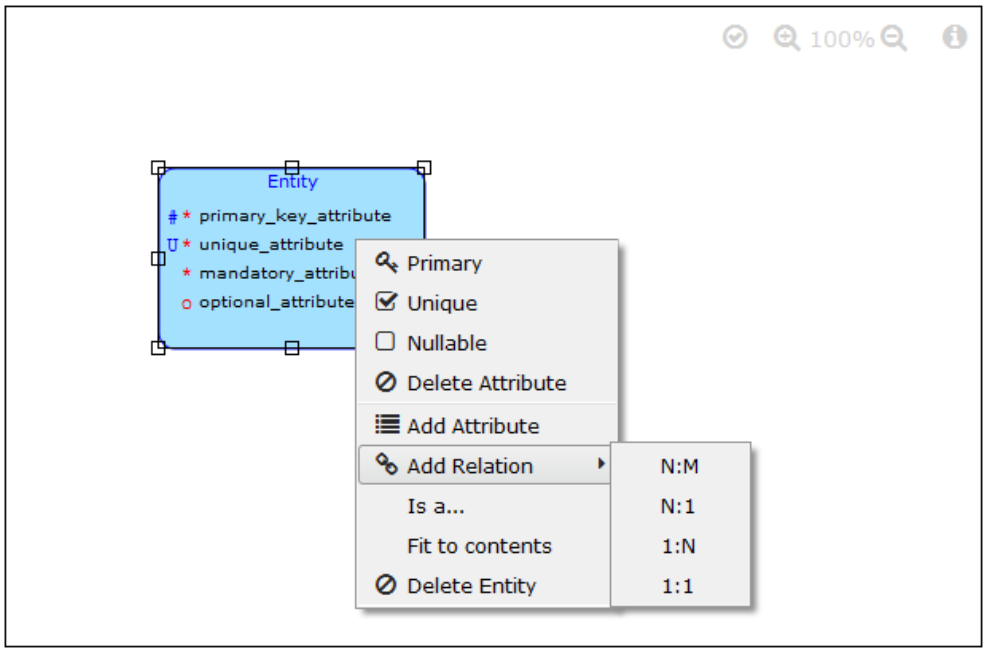


Figure 2.5: Final version of the application demonstrating context menu of an attribute

- Items may have icons without needing too much extra space. Furthermore icons help user memorize available options and access them more quickly. They are also used to show current state directly in menu.
- Context menus are consistent across whole data modeler. User will always know that on right click he will get same-looking menu with appropriate options.
- Because context menus are displayed on click, they are no longer overlapping parts of the diagram unless user wants to use some option from the menu.

Each object in diagram will now have its own context menu options attached, including diagram itself as well as secondary objects like relationship's control points. Moreover, context menu is "built" top-down, meaning that when user induces menu for an attribute they will also get options for attribute's entity as well. The only exception is diagram's menu, which is shown only when directly induced. Inner workings of menu system will be discussed in more detail in the next chapter.

Last issue to solve from previous iteration is in-place names edit, which was not consistent for relationships roles (those were edited from object menus). Although relationships do not show their names by default, they can be toggled

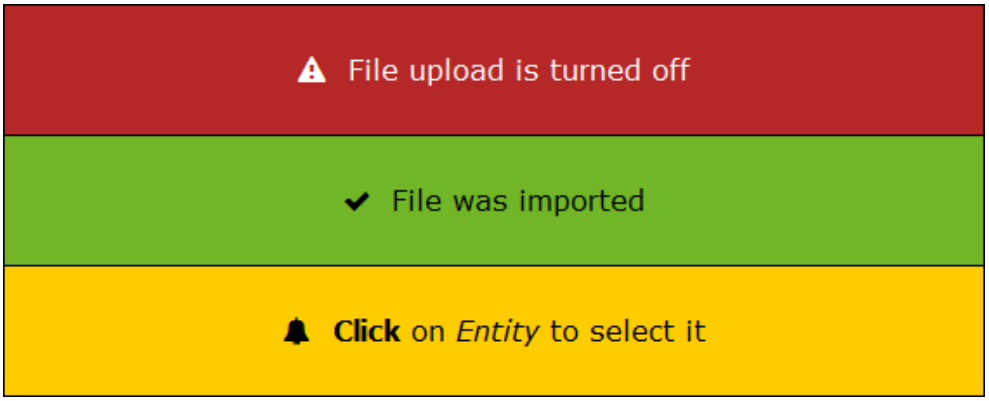


Figure 2.6: Example of error, success and hint message from final application

from the menu. Once the name is shown, it can be edited by simple mouse click, which shows text input. Name change may be confirmed by clicking in the canvas or by pressing *enter* key and canceled by pressing *Esc* key, which is now same system used across all editable texts in the diagram. The only change from previous prototype is that doubleclick is no longer required, edit is initiated by single click.

New in this iteration is also a messaging system, which is used to show error messages, success messages and can also be used to show tips or tutorial. These messages are shown across top of the canvas with centered text. Icon next to text and color of the background of the message denotes what type of message is being displayed.

Chapter 3

Implementation

In this chapter we will discuss the implementation of the diagramming utility itself. We will start from the ground up, overviewing used technologies and how were they used, including programming patterns, then we will discuss overall architecture, after which we will delve into implementation details and explanations of some of the algorithms and inner workings of the modeler.

3.1 Technology

Whole data modeler utility is built on four technologies: HTML, CSS, JavaScript and SVG. One of the non-functional requirements of the data modeler is that it is to be written in JavaScript. This requirement stems from the need to run the modeler online, in web browser and on client to ensure sufficient interactivity. This requirement also indirectly leads to the rest of the technologies. Since data modeler has to work in a browser, HTML[4] and CSS[5] are used to render the data modeler utility. Lastly, SVG is used to render and manipulate the diagram itself.

3.1.1 JavaScript

JavaScript has its roots in Netscape. It was invented by Brendan Eich, who was supposed to "do Scheme in a browser" [16]. JavaScript was first implemented in Netscape Navigator 2.0 in May 1995 under a name "Mocha" [17]. Nowadays it is an implementation of ECMAScript standard[6], first edition of which was released in 1997 and, currently, 7th edition is being developed [18]. As of writing of this thesis, all modern browsers fully support 5th edition of ECMAScript [19].

[4] HyperText Markup Language

[5] Cascading Style Sheets

[6] Maintained by Ecma International – European association for standardizing information and communication systems

JavaScript is a prototype-based scripting language with first-class functions, thanks to which it supports object-oriented, imperative and functional programming styles [17]. It needs a host environment that provides objects to which JavaScript may connect and gain programmatic control of [18, 20]. In a client-side implementation the host environment is usually a web browser, giving JavaScript access to Document Object Model (DOM) and some of the browser's functionality. Server-side implementations also exist, one of the most popular ones is Node.js which is built on Chrome's V8 JavaScript engine [21].

In this project we have used object-oriented style as it was deemed most appropriate. We presume that further development and maintenance will be done by students of Faculty of Information Technology who, based on current curriculum, should have most experience with this programming style.

Further in this section we will also discuss some of the main concepts of the object-oriented programming that were used in our JavaScript implementation.

3.1.1.1 Namespacing

Oxford Dictionary defines *namespace* as "a class of elements (e.g. addresses, file locations, etc.) in which each element has a name unique to that class, although it may be shared with elements in other classes". [22] We could also think of namespaces as "container" which bundles together code for the application or its part to prevent naming conflicts. We have already discussed that data modeler will be deployed to a DBS Portal and proper namespacing is vital for smooth deployment and prevention of conflicts with the rest of the code base.

Namespace concept does not exist in JavaScript, namespaces are simulated by objects instead, which is allowed by first-class functions. As a result, there is no difference between regular objects and namespaces in Javascript [23]. Following pattern is used to define namespace:

```
// global namespace for whole data modeler code
var DBSDM = DBSDM || {};

// sub namespace Model
DBSDM.Model = DBSDM.Model || {};
```

First, the check is performed whether a `DBSDM` global object is already defined. If it is not, empty object is created and assigned to `DBSDM` variable, which will now serve as a global namespace. Similarly, sub namespace `DBSDM.Model` is created. The shorthand assignment with logical OR works, because in JavaScript *logical OR* returns left expression if it can be converted to `true` and otherwise returns right expression.

3.1.1.2 Classes, Encapsulation

Like namespaces, JavaScript does not have a `class` keyword like many other Object-oriented programming (OOP) languages do[7] (e.g. C++ or Java). Since JavaScript is a prototype-based language, new classes are created as an assignment of anonymous function to the variable. This function will serve as a constructor and will be invoked each time new instance of the class is created.

Typically, new instance of a class is created with `new` operator. When `new Foo()` is executed, following happens [25]:

1. New object inheriting from `Foo.prototype` is created

2. Constructor function with `this` variable bound to newly created object is called

3. Result of the whole expression is object returned by the constructor. If no object is returned, the created object is returned instead.

All properties of the class are defined inside constructor function. If a variable is defined as a property of an object attached to `this` variable it will become *public property*, while variables declared with *var* keyword will be private and accessible only to private methods.

Methods in JavaScript are of three types: public, private and privileged. *Public methods* are defined as a property of *object's prototype*, while both *private* and *privileged* are defined in constructor. Private methods are defined as regular functions inside the constructor, have access to private variables but are not accessible to public methods. The bridge between public and private methods is created by privileged methods, which have access to private methods and properties and are accessible to outside and public methods. They are defined as property of the *instance*. [26] These concepts are demonstrated in code snippet in listing 1.

Although this approach works, there is a problem with performance and memory footprint. Unlike public methods which are defined on object's prototype and thus are created only once for all instances of the class, for each private and privileged method new object and closure needs to be created every time `Foo` is instantiated. Since one of the requirements of the data modeler is the ability to display large number of diagrams at one page, (true) private and privileged methods were not used in this form. Instead, to gain some ability to restrict access, a variation on *module pattern* and *naming conventions* were used.

[7] Syntactic sugar for creating classes, along with `class` keyword for class definition and `extend` for inheritance has been added in ECMAScript 6 [24], bringing whole syntax closer to other OOP languages

```
var DBSDM = DBSDM || {};

DBSDM.Foo = function() {
    this.publicProperty = "World";
    var privateProperty = "Hello";

    // private property used to make public members accessible
    // to private and privileged methods
    var that = this;

    function privateMethod() {
        return privateProperty;
    }

    this.privilegedMethod = function() {
        return privateMethod() + " " + that.publicProperty;
    }
}

DBSDM.Foo.prototype.publicMethod = function() {
   return this.privilegedMethod();
}

var foo = new DBSDM.Foo();
console.log(foo.publicMethod()) // prints "Hello World"
```

Listing 1: Code snippet demonstrating basic class definition and encapsulation in JavaScript

3.1.1.3 Module Pattern, Prototypal Pattern and Naming Conventions

Since its creation in 2003, module pattern has been popularized by Douglas Crockford in his lectures [27]. It is being widely used and its usefulness and variability has been proven by the wide adoption in many popular JavaScript libraries and frameworks, including e.g. YUI [28] or jQuery [29]. Private members are achieved via *closures* and *Immediately-invoked function expressions (IIFE)* [27].

The simplest form of this pattern can be seen in listing 2. In this code snippet, new module called `Module` was created by defining and immediately invoking anonymous function. Inside this function, private members which are not accessible from outside are defined as standard JavaScript variables and functions. The function returns an object (using object literal notation in this case), which defines public API to the module. Public methods may access

```
var Module = (function(){
    var privateVariable = 0;

    function privateMethod() {
        return privateVariable++;
    }

    // anonymous function returns an object
    // which defines public interface
    return {
        publicVariable: "Hello",
        publicMethod: function() {
            console.log(this.publicVariable);
            return privateMethod();
        }
    }

})(); // immediately execute anonymouse function
      // and assign the result to Module variable
```

Listing 2: The basic form of module pattern in JavaScript

other public members via `this` keyword as well as all private members, while private methods may access only other private members[8]. Thanks to closures, all private functions are created only once, but the pattern in this form does not allow instantiating of the created objects since its missing constructor, so in OOP terms we could say it simulates *static class*.

When module pattern is combined with class definition covered in previous section, proper ability to create classes with private members and ability to instantiate object could be achieved to some degree. Although private members are possible, private variables declared in a module pattern fashion would only work as private class variables, instead of private instance variables. This problem is demostrated in codes snippet in listing 3.

As you can see, when calling `getName` method on first object, we will get unexpected output `Second` instead of `First`. This is caused by the fact that both `f1` and `f2` share the same closure in which `privateName` is defined, thus both instances of the `Foo` class reference same variable.

Due to the nature of JavaScript it is not an easy task to achieve proper access restrictions. It could be argued whether this is even something that should be tried to simulate. One of the main disadvantages of private members

[8]It would be possible for private method to access object properties, if it were invoked by `call` method, which assigns object given as first argument to `this` keyword: `privateMethod.call(this)`. This is often used to e.g. chain constructors.

```
var Foo = (function(){
  var privateName;

  function Foo() {} // constructor function

  Foo.prototype.setName = function(name) {
    privateName = name;
  }

  Foo.prototype.getName = function() {
    return privateName;
  }

  return Foo;
})();

var f1 = new Foo();
var f2 = new Foo();

f1.setName("First");
console.log(f1.getName()); // prints "First"

f2.setName("Second");
console.log(f2.getName()); // prints "Second"

console.log(f1.getName()); // prints "Second"
```

Listing 3: Demostration of private class variables in module pattern.

in JavaScript is that they suppress dynamic nature of the language. One of the effects of this is that they restrict patching without need to change the original (3rd party) code [27, 30]. They also create further complexity, because they are being called differently (private methods directly, public as a property of `this`), which could become a problem when one decides to change access restrictions of the member.

One of the ways that private members are simulated in dynamic languages that do not have native access restrictions is *naming convention*. One of the examples is Python, which, by convention, prefixes private members with an underscore [31]. This, naturally, does not create "true" private members, since they are still accessible from the outside of the class, but they let programmers know that certain members should be *treated* as private (or protected).

In our implementation of the data modeler it was decided to use module pattern for all data modeler classes. Static classes are used for helper ob-

jects, private class variables in regular classes are often used as shorthands to namespaces or as constants. The naming convention inspired by Python is being used for instance private members, where each private member is prefixed by single *underscore* character. Decision to use primarily naming conventions to manage access was done in regards to future development. Since there is no "best" solution for having instance private variables similar to other OOP languages in JavaScript and prefixing is one of the most viable options, prefixing method names as well will keep consistency and good code readability. This will also eliminate other issues with scope and visibility of other class members and it should be easily understandable to future code maintainers.

3.1.2 SVG

HTML5 introduced two methods for drawing graphics in a web page, raster-based and vector-based one. The `canvas` element provides ability to draw bitmap images. With its low-level API scripts are able to draw basic shapes and even create simple animations [32]. It can be also used by WebGL to draw hardware-accelerated 3D graphics [33].

SVG [34], while older standard than HTML5[9], was in HTML5 allowed to be drawn inline via `svg` element. SVG stands for *Scalable Vector Graphics*, from which it is apparent that this technology is vector-based. Graphics are described in XML, which creates full scene graph (Document Object Model), which can be fully accessed and manipulated via scripting languages (in our case JavaScript). Similarities with HTML do not end there, SVG elements are stylable and they may have events attached, which makes them great for creating interactive graphical applications.

It should be clear from this description itself why the SVG was chosen as the technology used to draw diagrams. Data modeler is interactive, which means that user will have to manipulate objects in some way. That is supported by both Document Object Model and ability to attach events (`mousedown`, `mouseup` etc.). Moreover, stylability allows, in addition to visually format elements via CSS, to also hide certain elements when they are not needed, without the need to expensively create or destroy them.

Although a lot of JavaScript libraries for SVG manipulation exists, initial research showed that they are not suitable for this project, due to their abstraction from SVG DOM and focus on features that are not key for data modeler, like animations. It was decided to not use any library and instead manipulate SVG DOM directly. As an additional advantage, future development will not rely on 3rd party libraries.

[9] Current revision of SVG standard, 1.1, was released in August 2011 [35], while HTML5 standard was released three years later, in October 2014 [36]

`svg` element

`svg` element is used to create a new SVG canvas (SVG document fragment) inside a HTML document. In this canvas, other SVG elements may be created and the resulting graphics would be rendered. Canvas is infinite in both directions, but rendering occurs relative to finite region, which is called *viewport* and basically defines what the user will see [37]. Each viewport also defines *coordinate system* in relation to which all child objects are positioned. `svg` elements may be descendants of another `svg` element, thus creating new SVG document fragment with its own viewport and coordinate system.

`g` element

`g` element is used for grouping of other SVG elements. Some of the attributes may be used to manipulate whole group at once, e.g. `transform` attribute used to translate, scale or rotate SVG elements. Unlike `svg` element, which essentially groups objects as well, `g` element does not create new viewport and all grouped elements are still being transformed relative to the closest `svg` element in hierarchy.

`defs` and `use` elements

These elements form a part of the mechanism for content reuse. `defs` is similar to `g` element with the exception that all elements in former are not directly rendered and they may be later referenced by other elements. Elements may be referenced from other properties by their `id`s, e.g. gradient defined in `defs` may be referenced from `fill` property.

The `use` element is used to reference entire objects. This template object indicates that another object will be rendered at that place of the document.

Referenced objects do not need to be defined in same SVG document fragment. This means that we may define "shared" objects once and use them in multiple data modelers, if they were to be displayed at once in one page. This technique was exploited in our implementation as well, shared elements are created during the initialization of the diagram and are later referenced from individual canvases.

3.2 Architecture

The most important part of the data modeler is, naturally, diagram handling. Creation of underlying model, user interaction and rendering of the diagram as a whole is the main purpose of whole application, and thus the good implementation is vital. An effort was made to produce code with possibility of future extension, which would be also easy to maintain. In this section we will describe the basics of how the code is structured, although it is not meant as a complete documentation.

```
DBSDM.Diagram.init();
(new DBSDM.Canvas()).create();
```

Listing 4: Minimal usage example. This code will initiate data modeler with default settings and create one empty canvas for modeling.

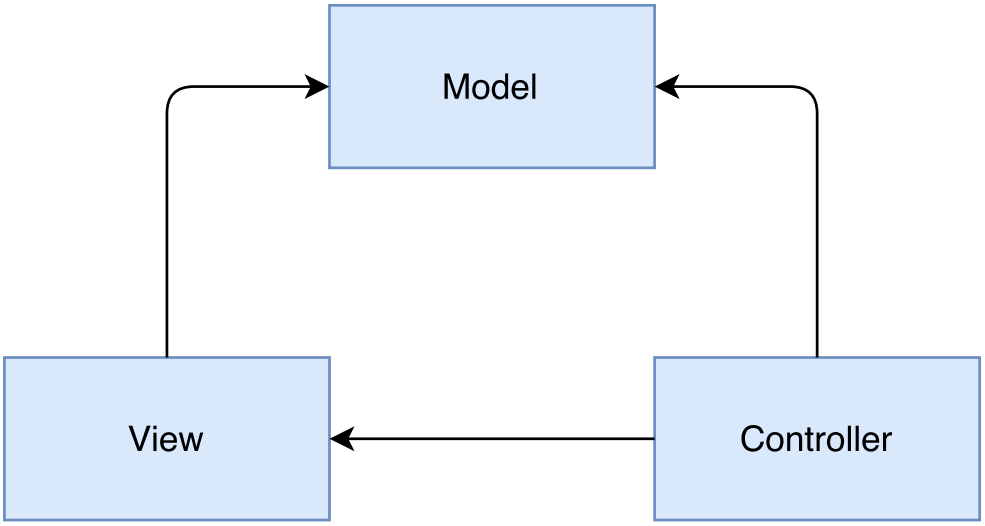


Figure 3.1: Model-View-Controller pattern

Before any canvas can be drawn, modeler has to be initiated. That is the responsibility of `DBSDM.Diagram`, which also maintains shared objects and keeps track of created canvases. Each canvas is in this case one model, one diagram. Once the data modeler has been set up, new canvas may be created by instantiating a `DBSDM.Canvas` class and calling `create` method on this object. This will create new `svg` element that will serve as container for other SVG objects.

Once the canvas is created, user may start drawing their logical model. At this point we are implementing an application with graphical user interface (GUI). One of the most popular patterns for this implementation, especially for web based applications, is Model-View-Controller (MVC) pattern[10], which was originally created by Trygve Reenskaug for Smalltalk [27]. This pattern solves the problem of separation of concerns between data, their representation and user input handling and decouples objects of the application into three layers: model, view, controller.

Model is a top-most layer. Objects on this layer do not know anything about how the data will be presented to the user or how the user will be able

[10]Other popular patterns are e.g. Model-View-Presenter or Model-View-ViewModel

to interact with them. The responsibility of this layer is implementation of the data structures and business logic of the problem domain of the application. Objects on this layer may create one to one or one to many relationships.

View is responsible for visually presenting data to the user.

Controller is the intermediary between model and view layer. Controller objects are responsible for handling user interaction and may be also allowed to govern objects on other layers.

Several interpretations and implementations of the MVC pattern exist [38, 39, 40, 27]. While all agree with the basic separation and responsibilities of each layer, they may have different opinion about how these layers communicate with each other. There is no doubt that model is a separate layer, that must not rely on other layers. However, if the *model* in the application may change without the user prompting it (e.g. by periodical network update), model may implement *Observer* pattern to notify rest of the application about the changes. Another question is whether *view* should be able to directly make requests to the model, or whether all queries should be handled via controller layer and, similarly, whether events should be defined in view or in the controller (however, handling of the events is strictly done in controller). If the application needs to implement persistency of the data, the point of contention may also be whether that is the responsibility of the model layer or the controller.

In our implementation, the model does not implement observer pattern. Data in the model may change only by two ways – directly by the user, where the updates would be handled by controller, or via import, which is handled by `DBSDM.Canvas` class. After import, all objects are created anew, and thus notifications from model are not needed. Furthermore, view can directly query model and only persistency option is export, which is also handled by `DBSDM.Canvas` class

The decoupling and separation of the application into three layers allows for better flexibility, reuse and expansion. For example, Barker notation is used in current implementation of data modeler. However, that is just the one of the possible presentations of the underlying data – model. Same diagram in another notation would have exactly the same underlying model, so if we would decide that we want to have the ability to draw diagram in, say, Chen's notation, all that would be needed to do is to change the current implementation of the view layer. Similarly, the model layer could be extended to keep track of more data, for example data types of the attributes. This addition would not affect other layers of the application, but another view could be created to display e.g. relational model. Therefore, two very different presentations of the same data would be possible.

3.3 Mouse Handling

Primary control method of the data modeler is the mouse, which is made possible by HTML and SVG events. The main, most frequently used events in the application are `mousedown`, `mousemove` and `mouseup`. Once the view creates an SVG elements, appropriate event listeners may be added to them. User input is thus made possible by events created in view and handled by controller objects. However, not all user inputs are handled directly.

In essence, events are fired when a mouse cursor is located over an element, which has appropriate listeners attached. Let's consider entity drag. When user wants to drag an entity, following happens from the user's point of view:

1. User moves mouse cursor over an entity and presses down the left mouse button

2. With the left mouse button still pressed, user moves the mouse, which will translate entity's position

3. User depresses the button, making the entity's position final

In ideal case, this could be handled directly by mouse events. The `svg` element that forms the container of the entity would have all three mouse events attached, and when fired they would be handled by entity's controller object.

Now, consider the user makes a rapid mouse movement. If the movement is fast enough, browser might not have a chance to execute whole update loop in time and the user might see the entity dragging behind cursor so much, that the cursor would no longer be over entity. If the user would not wait for the entity to "catch up" and depresses the mouse button, `mouseup` event would not fire for the entity but for whatever object that might be located at cursor's position.

To solve this problem we have created the `DBSDM.Mouse` class, which serves as a layer handling all mouse events. The basic idea is that `mousemove` and `mouseup` events are only attached to the canvas itself, while `mousedown` event is fired on target element as usual. Let's consider entity drag again, now with an added explanation of how are these events handled:

1. *User moves mouse cursor over an entity and presses down the left mouse button.* `mousedown` event is fired on an `svg` element forming the container of the entity. The event is handled by `down` method of the `DBSDM.Mouse` object, which will receive reference to the controller object of the entity as an argument. Controller object reference is saved.

2. *With the left mouse button still pressed, user moves the mouse, which will translate entity's position.* Thanks to event bubbling[11], `mousemove` event is propagated to the canvas and fired. The event is handled by `move` method of the `DBSDM.Mouse` object, from which a handler for `mousemove` events on the controller object is invoked.

3. *User depresses the button, making the entity's position final.* `mouseup` event is fired on the canvas. The `up` method of the `DBSDM.Mouse` object handles the event. Proper handler is then called on entity controller object, after which the reference to this object is nullified.

With this approach even if the user makes rapid movement, events are always handled by right objects.

This added layer has several other advantages. Since each canvas on the page has its own instance of the `DBSDM.Mouse` class, it is possible to transform document coordinates into canvas' viewport coordinates, further simplifying event handling. It adds versatility, because objects may be attached programmatically instead of through the `mousedown` event. Finally, it saves memory and improves performance, because overall number of event listeners is cut down by only requiring attachment of `mousedown` listener on target objects, while still allowing these objects to handle `mousemove` and `mouseup` events.

3.4 Context Menu

Context menus are the main method of providing *options* for the user. Despite each object offering different options, there is only one context menu for the whole page, which is managed by a static class `DBSDM.MenuController` that handles menu creation, display and invocations of appropriate handlers on diagram objects.

Menu is defined by a JavaScript object. Each property of this object defines one section of the menu – options for one type of the object on the diagram. Options themselves are required to have *name* and *id*, while *icon* and *permissions* (user for enabling or disabling menu options) are optional. When diagram is initialized, the definition of the menu is used to create its HTML representation.

The menu is set up in a way to allow showing multiple sections at once. That means that when user gets menu for e.g. attribute, they will also see options for the entity itself. The only exception are canvas options, which are displayed only when menu for canvas is directly invoked.

Display of the menu is handled by listening to `contextmenu` events on elements of the diagram. When the event is fired, controller object, section

[11] When fired, event propagates from the element on which it was fired up the DOM tree and fires for all ancestors

```
{
  /* ... */
  entity: [ // definition of menu options for entity
      // Add Attribute option, with id ``attr'', icon
      // ``list'' and requiring permission ``allowEdit''
      ["Add Attribute", "attr", "list", "allowEdit"],
      [
          // definition of Add Relation item
          // which creates submenu
          "Add Relation",
          [
              // definition of submenu items
              ["N:M", "rel-nm"],
              ["N:1", "rel-n1"],
              ["1:N", "rel-1n"],
              ["1:1", "rel-11"]
          ],
          "link", "allowEdit"
      ],
      ["Is a...", "isa", null, "allowEdit"],
      // icon and permission are optional
      ["Fit to contents", "fit"],
      ["Delete Entity", "delete", "ban", "allowEdit"]
  ],
  /* ... */
}
```

Listing 5: An excerpt from menu definition

name and optionally parameters for setting the state of the menu item, are sent to `DBSDM.MenuController.attach` to attach controller object to given section, essentially marking it for display. By bubbling through the DOM, multiple sections may be attached. Once the event reaches canvas, the menu is displayed by positioning it at the coordinates of the mouse click.

The user actions are handled by listening to `click` events on the menu's container. Once the user clicks the item, `DBSDM.MenuController` figures out in which section the item resides and calles `handleMenu` for attached object of that section.

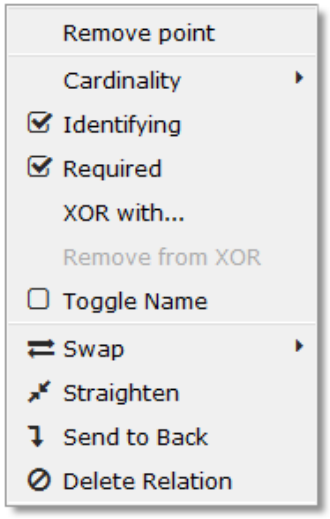


Figure 3.2: Context menu of relationship's control point, demonstrating the hierarchical system. Menu has three sections, first for control point itself, second for relationship half, third for whole relationship.

3.5 Strategy for Placement of Relationships Anchors

When creating new relationships, the application has to decide where should the relationships attach to the entities. We will refer to the point, where the relationships attach as an *anchor*. In fact, we will consider an anchor to be whole "crow's foot" from the Barker notation. The problem that needs to be solved therefore is, how to find an appropriate location for new anchor.

There are two main requirements we have to consider: anchors should not overlap and relationships, drawn as a straight line, should be as short as possible. First requirements stems up from the diagram readability. Enforcing second requirement should produce more visually pleasing diagrams. Without it, relationships would be allowed to be drawn in a way, that the line would be crossing the entity it is attached to.

Our algorithm works in a following way:

1. Given source and target entity, find the edges that could be considered for the anchor placement. Let's consider source entity to be placed to the upper left of the target entity. In this case, bottom and right edge of the source entity and upper and left edge of the target entity should be considered. However, relative position of the entities is not enough, instead, edges themselves have to be tested in order to take into account various possible sizes of the entities. Hence, opposite edges of the entities are tested to determine entities position. Left and right edges of these

entities are selected if the left edge of one entity is located to the right of the right edge of the other entity. Top and bottom edges are tested analogically.

In special cases when entities overlap or form ISA hierarchy and no edges were selected, all edges of both entities will be used to find the best position of the anchors.

2. Find one possible location for each new anchor on each selected edge. In a trivial situation when no anchor is yet created on the edge, use middle of the edge.

   In other cases, selecting one point is not as straightforward. If we were strictly trying to satisfy no-overlap requirement, we could place each point at a specified minimum distance from already created anchors. This strategy would, however, fail, if the edge gets saturated. True, anchors would not be overlapping, but also no new relationships would be possible to make, until the entity is resized. Moreover, diagrams tend to be more readable and "nice to look at" when the anchors are evenly space, although this is matter of personal opinion and taste.

   The strategy we have ended up using consists of interval halving and works as follows: split the edge into intervals by current anchors; the new point is at the middle of the largest interval. Notice that this algorithm can not produce anchors at the very end of the edge[12]. This may not be a problem for larger entities, but could lead to overlapping anchors more quickly than necessary on smaller ones. Because of that, when the new point's distance from the closest anchor falls below certain threshold, edge's ends are also considered as possibility. Out of these three locations, the one that is most distant from other anchors is selected as a new point.

3. Out of the all possible combinations of points, select the closest two, one from each entity. This ensures that the relationship will be as short as possible.

Same algorithm is used when entities move and relationships have to be recomputed[13]. The only difference is that the new point is not being computed for the edge on which the anchor currently exists. Instead, anchor's current location is used for that edge.

[12] When placing anchors, some offset from the end of the edge is always considered to prevent visual overflow of the anchors outside the entity

[13] Relationships that the user manually manipulated (changed their position or moved their control points) are the exception and their positions are not changed when entities move until user straightens them.

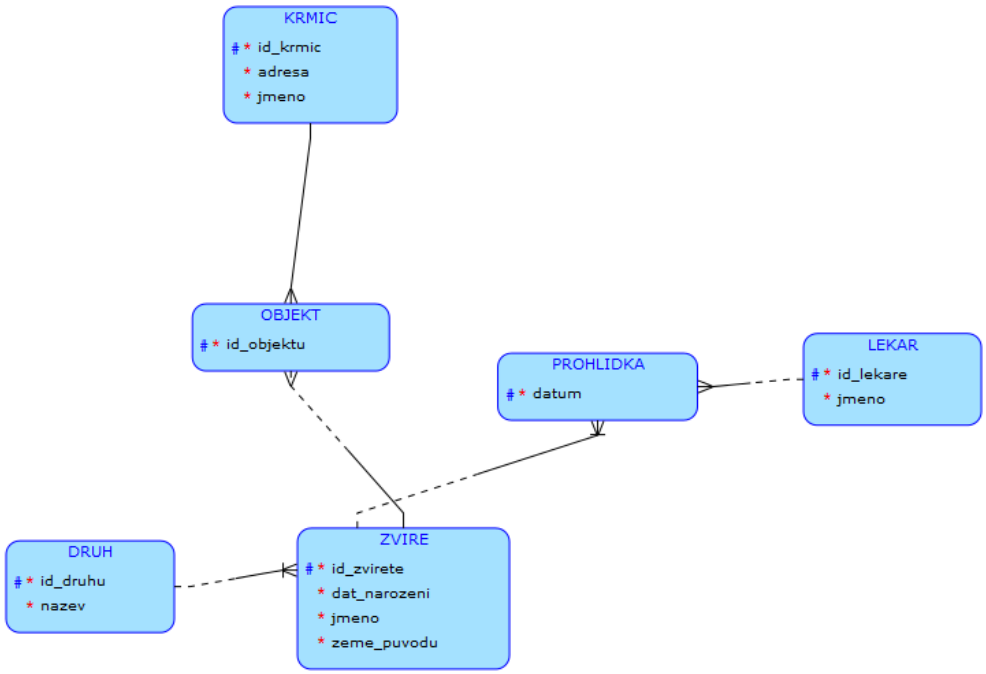


Figure 3.3: Example of automatically laid out diagram

3.6 Automatic Diagram Layout

Automatic diagram layout ("sort") is one of the functional requirements, stemming up from the examinator role. Examinators usually have to mark tens of tests, which may include one or more diagrams. Although automatic comparison of answers with reference solution is possible by comparing the textual representation of the model, it is still required to manually check how major the mistake the student made was. To aid in this process, diagrams need to be visually laid out for easier comparison with reference solution. The functional requirement says, that it has to be possible to display equivalent diagrams in the exactly the same way, when displayed in one browser session[14].

Same display in itself would not be very useful though. Diagram with all entities at the same place would pass this requirement. The layout has to be readable – all elements have to be discernible. Additionally, relationships should be drawn in a way that they would be minimally crossing other relationships and entities should be evenly spaced. In addition to these obvious rules, more criteria for drawing aesthetic diagram could be created, however further research and study, which is beyond the scope of this thesis, would

[14]Diagram layout may differ between various browsers, browser versions, operating systems etc. due to minor differences in floating point arithmetic.

have to be performed to support them.

To be able to lay out the diagram, we need to consider entities and relationships. We may think of a diagram as a *graph*, where each entity of the diagram is a node, while relationships between entities are edges. Direction of the relationships is not important, the graph is undirected. Loops in graph are created by recursive relationships, multiple edges between same nodes are not prohibited. Although user may add control points and change the line that is created between two entities, we may force straight lines during sort. One more thing to consider is that although we are thinking of the diagram as of graph, we are not transforming it. The graph drawing algorithm has to be possible to use directly on diagrams themselves, which, unlike graphs, have nodes (entities) of various sizes.

Various approaches have been proposed for sorting small undirected graphs [41, 42]. In our implementation we have chosen to use one of the force-direct methods, which is a class of algorithms that tends to satisfy the rules we have laid out for drawing visually pleasing diagrams. Methods of this class tend to not scale well for large graphs and other techniques and refinements have to be used for graphs with more than a few hundreds of nodes [42]. Since diagrams made via the data modeler will never reach this scale, classic force-directed algorithms are more than sufficient.

The core of the layout algorithm is implementation of the Fruchterman and Reingold's method proposed in 1991. It extends the spring-based algorithm of Eades by repulsive forces to achieve even distribution of graph's nodes. The basic idea of this spring-electrical model is to consider each edge of the graph to be a spring and each node to behave like atomic particle, that exerts repulsive forces on other particles (entities) [41, 42].

Springs are, in our case, relationships between entities. Springs are trying to keep the predefined, optimal, length – when entities are too close, springs will repel them, when they are too far apart springs will attract them. In addition to proposed model we are also adding additional force to try to put entities on a grid. That is because entities are rounded rectangles and we consider it to be more visually pleasing, when the relationships form angles as close as possible to 90 degrees with the edge of the entity.

Algorithm 1 Diagram Layout Algorithm

```
fit()                                  ▷ Resize all entities to fit their contents
for i ← 1, iterations do
    computeSpringForces()
    computeEntitiesRepulsions()
    applyForces()
end for
moveToOrigin()    ▷ Move all entities to the origin of the canvas' viewport
```

$$length = |\vec{r}.x| + |\vec{r}.y| \tag{3.1}$$

$$\vec{a} = \vec{r} \cdot \text{attractionScale} \cdot (length - \text{optimal}) \tag{3.2}$$

$$\vec{v} = \begin{cases} \vec{r}.x & |\vec{r}.x| < |\vec{r}.y| \\ \vec{r}.y & \text{otherwise} \end{cases} \tag{3.3}$$

$$\vec{s} = \vec{v} \cdot \text{straightenScale} \cdot (|\vec{v}.x| + |\vec{v}.y| - \text{optimal}) \tag{3.4}$$

$$\vec{f_R} = \frac{\vec{a} - \vec{s}}{2 \cdot length} \tag{3.5}$$

Figure 3.4: Spring force $\vec{f_R}$. $\vec{r}$ is a vector representing relationship, computed from positions of its anchors, $\vec{a}$ is attraction vector of the spring and $\vec{s}$ is straightening vector trying to keep entities in a grid. Final force vector is halved, since the same force of opposite orientation is being applied to both source and target entity at once.

$$\vec{v} = (C_B - C_A) \tag{3.6}$$

$$length = \sqrt{\vec{v}.x^2 + \vec{v}.y^2} \tag{3.7}$$

$$\vec{f_E} = \begin{cases} 0 & length > \text{optimal} \cdot \text{repulsionDistanceScale} \\ \vec{v} \cdot \frac{\text{repulsionScale} \cdot optimal}{length^2} & \text{otherwise} \end{cases} \tag{3.8}$$

Figure 3.5: Repulsion force $\vec{f_E}$. C_A and C_B are the centers of entities.

Repulsion forces are computed for all entities that are in certain radius of the entity. Fruchterman and Reingold proposed ignoring repulsion from distant entities to reduce the complexity of the algorithm, but the other reason we are ignoring them is to prevent the "float-away" of disconnected entities.

The algorithm is iterative. In each step, the forces exerted on each node via springs and repulsions are calculated. Once the calculations are complete, the forces are applied and entities move. The number of iterations influences the quality of the layout, a trade-off between higher quality of the layout and performance, which is one of the limiting factors of the algorithm. In each iteration, repulsive forces for each relationship and each pair of entities must be computed, which makes the per-iteration cost of the algorithm $\mathcal{O}(|V|^2)$, where $|V|$ is number of entities.

The other problem of not only this algorithm but whole class is that they are prone to getting stuck in *local minimums*. Improvements could be made

by using better initial configuration[15], simulated annealing or other techniques which could move whole parts of the graph [41, 42]. These options were not deeply explored, since they are out of the scope of this thesis and basic algorithm is providing sufficient results. It is, however, one of the fields that could be improved in future work.

Constant name	Value
iterations	100
attractionScale	0.25
straightenScale	0.1
optimal	100
repulsionScale	25
repulsionDistanceScale	25

Table 3.1: Set up of the diagram layout algorithm

Figure 3.3 shows the result of the sorted diagram with settings listed in table 3.1. This diagram can be also seen in figure 1.5, which shows the result of the sort by the current implementation. More results of the layout algorithm can be found in Appendix C.

3.7 Import

Currently, there are two ways to import diagrams. One is to use JSON, which was previously exported from the data modeler, the other is to import Data Modeler Design from SQL Developer.

3.7.1 JSON

JSON is a lightweight, data-interchange, text format built on collections of objects (name-value pairs) and arrays [43]. Since it is based on the JavaScript object notation, it can be easily generated and parsed by JavaScript (and other programming languages), and is readable by humans, it was selected as a format for storing data models created in our data modeler.

To export data, data modeler creates one object with two properties: *entities* and *relations*. *Entities* property is an array of objects, where each object represents one entity storing its name, name of its parent in ISA hierarchy and list of attributes. *Relations* property is array of arrays describing all relationships that exist in the model. Each relationship is stored as an array of two objects, each object representing one half of the relationship. When model was

[15]Currently, the layout algorithm is automatically called after each import, which places entities alphabetically in a rectangle on the canvas, which becomes initial configuration for the layout algorithm

```
{
  "entities": [
    /* ... */
    {
      "name": "Object",
      "parent": null,
      "attr": [
        {
          "name": "object_id",
          "primary": true,
          "unique": false,
          "nullable": false
        }
      ]
    },
    /* ... */
  ],
  "relations": [
    /* ... */
    [
      {
        "entity": "Object",
        "identifying": false,
        "optional": true,
        "cardinality": 0,
        "xor": null
      },
      {
        "entity": "Animal",
        "identifying": false,
        "optional": false,
        "cardinality": 1,
        "xor": "17e7eb61cc2d7d7a9fbde5bed789a80f"
      }
    ],
    /* ... */
  ]
}
```

Figure 3.6: Excerpt of JSON format used by import and export

Algorithm 2 Generation of XOR Relationship Hash

```
hash ← Array

for all relationshipHalf in XOR do
    relationship ← GetRelationship(relationshipHalf)
    push(hash, GetHash(relationship))
end for

sortedHash ← sort(hash)
return join(sortedHash)
```

marked by the examinator, incorrect objects will be exported with additional `incorrect` property.

Notable is also a storage of XOR relationships. During lifetime of the diagram, XOR relationships are tracked as an array of object references by the entity they are attached to. Simple solutions, like storing list of relationship's role can not be used, since there is no guarantee that the role will be set, nor that if set it would be unique. A solution that would not only allow correct import of XORs but also textual comparison was achieved by creating hash of the whole XOR relationship, which is used as an identifier during import. This hash consists of hashes of all *relationships* that take part in XOR – it is important to use hash of relationship instead of just relationship half attached to the entity, since there may be multiple equivalent halves forming different relationships.

Export may be initiated by user from the context menu of the canvas, or programmatically from the outside of the data modeler. It is also possible to export JSON string into variable instead of prompting download of a file, which may be used to store model in a database or to compare it with another model. Textual comparability is achieved by normalizing data[16] and sorting object's properties before its string representation is created. Similarly, JSON can be imported by user by dragging the JSON file onto canvas, or programmatically. Programmatic export and import is handled by `import` and `export` method of the `DBSDM.Canvas`.

3.7.2 SQL Developer DataModeler

SQL Developer offers multiple ways to save or export created designs. One of the possible options is the export to *Data Modeler Design*, which creates a folder structure containing XML documents. We were not able to find any official, public or private, description of this format, but exploring files of several exported models provided sufficient knowledge to be able to parse and recreate the model in data modeler of DBS Portal.

[16] In fact, normalization of user input is done during modeling

Logical model description is located in folder `logical`. In this folder, `arc`, `entity` and `relation` folders may be found, containing XML files describing XOR relationships, entities and relationships respectively. Each of these files contains description of one object that is assigned unique identifier and may reference other objects by their identifiers. In order to import Data Modeler Design, the zip file of this folder structure[17] is uploaded by dragging it onto the modeler's canvas. The zip file is then extracted via `JSZip` library [44] to allow parsing of XML object files themselves. In order to import the model, the *"import" object* described in previous section and in figure 3.6, is created and imported in the same way as if it was created from uploaded JSON file.

The creation of the import object has two phases, since we can not ensure that objects will be parsed in such a way, that all referenced objects are already known.

In the first phase all XML object files are parsed and saved in a map. Following maps/objects are needed:

entityMap A collection of entities in an identical format as they are stored in the import object. Both name and attributes are already set, `parent` property is yet `null`

parentMap An array of pairs, where the first item in pair is ID of the child entity in the ISA hierarchy and second item is ID of the parent entity

relationsMap An array storing relationships identically with their storage in the import object. In the map, entity names are entity IDs, since it is not guaranteed that the referenced entity was already parsed. `xor` is `null`, because information about XOR relationships are stored in *arc* files.

relationsRef A collection mapping IDs of relationships to their index in the `relationsMap`

arcMap Map of XOR relationships – arcs in the terminology of SQL Developer. Each XOR is an array of entity ID and relationship ID pair.

In second phase, all objects are already known and thus the references may be resolved. Maps created in the first phase are used to build a resulting import object.

3.8 Adaptation of Messaging System for Tutorial

Feedback is one of the principles of good interface design [45, 46]. One of the mechanisms implemented to provide feedback is messaging system, which

[17] It does not matter where in the archived folder hierarchy the folder `logical` is located. User may decide to upload Data Modeler Design in its entirety or only the folder `logical` – no objects outside this folder are used during import

can show three types of messages: *success*, *error* and *hint*. Each message has different icon and color of the background, helping user more easily evaluate the importance of the message at the first glance. While success and error messages are handling mainly status of the import, hint messages have been adapted to use in a tutorial, created to teach new users about basics of the modeler.

The basic idea of the tutorial is to tell the user what to do and how to achieve it. Currently, four steps of the tutorial are defined:

1. Start by drawing an entity or dragging an exported JSON or SQL Developer zip file into canvas

2. Click on Entity to select it

3. Right click on any element of the canvas to get more options

4. Click and drag middle mouse button to move the layout

Each step of the tutorial teaches user something new. In the first step user learns that he can either create an entity (and how) or import diagram. He also learns what import formats are supported. In second step user learns that to manipulate with objects he should use left mouse button and that some objects may be selected. In third step of the tutorial context menu is introduced and via menu user gains knowledge of more actions he can take. By telling the user that menu can be initiated on *any* element, he is lead to explore the modeler by himself. In last step of the tutorial the user learns that the area for drawing diagram is not limited by the size of the window. These four steps should give users enough information to comfortably use and create even complex diagrams.

We believe that users should not be punished by knowing more than is expected of them. User's first action may be to try right click, even if tutorial tells him to draw entity or import diagram. In this case he would already learn that context menus exist and he does not need to be shown the third item of the tutorial.

Implementation of tutorial system is handled by `DBSDM.UI` class. Tutorial messages are properties of the private `tutorial` object; property names are used as their identifiers to reference them from the rest of the code. Messages left to be shown and their order are defined as an array of property names of the `tutorial` object. Tutorial is initiated by a call to `advanceTutorial` method, which will show the first tutorial message. Once the user completes the task set by tutorial, `acceptTutorialAction` method will remove it from the list of unfinished tasks. If the completed action was currently the displayed one, tutorial is advanced to new task, if there is any task left.

Option	Default	Description
`allowEdit`	*true*	Allow changes to the diagram. If *false*, only actions that do not change the underlying data model will be allowed, e.g. visual transformation of entities
`allowFile`	*true*	Allow import and export actions. If *false*, import and export may be initiated only programmatically
`allowCorrectMode`	*false*	Allow switching to marking mode, in which examinators are able to mark incorrect parts of the diagram
`showTutorial`	*true*	Determines whether tutorial should be shown or not
`confirmLeave`	*false*	When *true*, user will be asked to confirm they want to leave, if the diagram was not exported after last edit; if user decides to stay, error message will be shown in the canvas with unsaved changes

Table 3.2: Available options during initialization of data modeler

3.9 Distribution and Integration of Data Modeler

Source code of the data modeler is formed by multiple files across different folders, majority of which are JavaScript files defining all application logic. It would be very impractical, error-prone, not to mention inefficient[18] and not easily update-able, to require programmers to manually point to each source file from the HTML. The application is therefore distributed in single merged JavaScript file, which needs to be loaded along with stylesheet of the data modeler and other libraries[19] that are used by the data modeler.

In listing 4 of section 3.2 of this chapter we have already provided minimal example of canvas creation. Listing 6 provides more realistic usage example of the integration into another project, while table 3.2 shows available settings. In this example, data modeler is initialized with non-default settings, which will apply to all canvases created in a document. Two canvases are created, a model defined by JSON string is parsed into JavaScript object and imported into second canvas. After import, the model is exported and non-prettified JSON string is printed into console.

[18] Possibly only until HTTP/2 is widely implemented

[19] Font Awesome v4.6.3 and above[47] for icons, JSZip v3.1.3[44] for extracting zip files, saveSvgAsPng[48] library for saving diagram as an PNG image

```
/**
 * Initialize Diagram
 */
DBSDM.Diagram.init({
    allowEdit: true,
    allowFile: true,
    allowCorrectMode: true,
    confirmLeave: true,
    showTutorial: true
});

/**
 * Create empty canvas
 */
var canvas1 = new DBSDM.Canvas();
canvas1.create();

/**
 * Create canvas and programmatically import diagram
 * Note that JSON string is truncated
 */
var json = JSON.parse('{"entities": [...],"relations": [...]}');
var canvas2 = new DBSDM.Canvas();
canvas2.create();
canvas2.import(json);

/**
 * Export contents of second diagram
 * and print JSON string into console
 */
var promptDownload = false;
var prettify = false;
console.log(canvas2.export(promptDownload, prettify));
```

Listing 6: More complete example of integration of data modeler, showing initialization with non-default settings, programmatic import and export

Chapter 4

Testing

Although the data modeler has been designed and developed with the best of intentions, in accordance with findings in initial analysis and evaluations of prototypes, it is still unknown how usable the application is. Proper evaluation and testing of the user interface is needed to discover any usability issues.

One of the methods of user interface design evaluation is heuristic analysis. Evaluators, usually experts in user interface design, are presented the interface (either a prototype or fully functional application) and asked to comment on it to express the opinion about how good or bad they consider it to be. Evaluation is usually done according to certain rules – heuristics. Heuristic evaluation is often used because it is cheap and can be used in early stages of the development, but it will probably not generate breakthrough design suggestions.

In 1990, Nielsen and Molich conducted a study on heuristic evaluations [49] in which they discovered that although even non-expert evaluators[20] may identify problems experts miss. Overall individual performance ranged only between 20% and 51%. They conclude that aggregated results of at least three to five people should be used to provide good evaluation and additional resources to be spent on alternative evaluation methods.

The most obvious method for testing the user interface, however, is empirical testing with real users of the application. Testing with real users may have various forms. We may observe users in the environment the software will be used, in specialized lab or even remotely. Each of this forms has its own advantages and disadvantages. For example, testing in real environment will be more realistic, but users may be more distracted by other people and we do not have full control over the conditions in the environment, so tests may be hard to repeat [50].

Typically, users are asked to complete a scenario – realistic list of tasks – during which they are being watched by evaluators, who are trying to identify

[20] Evaluators were groups of students and readers of Computerworld magazine, mostly industrial computer professionals

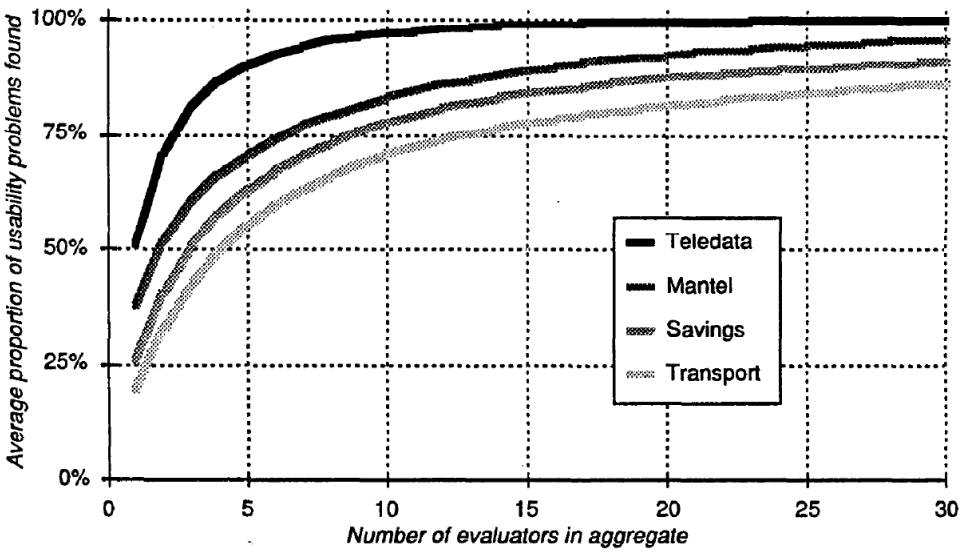


Figure 4.1: Nielsen curve showing aggregate proportion of usability problems found during four original experiments [45]

obstacles or parts of the software which do not support the task completion. After all scenarios are complete, user may be interviewed or asked to fill a questionnaire to get an insight into what he thinks of the interface.

4.1 Heuristic Analysis

Data modeler was heuristically evaluated by the author of the thesis in accordance with ten principles for interaction design defined by Nielsen [45]. These heuristics were based on nine original heuristics Nielsen developed with Molich in [51]. We will go through each rule, list its original definition and discuss how it was handled by our design. Since the evaluator is the same person as the interface designer, we are not expecting to find any major issues. However, the evaluation may still be useful to ensure that no rule was neglected during design.

Visibility of system status

> The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

Data modeler may end up in several states, each as a result of some action taken by the user. Provided feedback depends on the context. Some of the examples are:

- entity changes color when it is selected during creation of ISA hierarchy
- relationship halves are highlighted during creation of XOR relationship
- rectangle with control points is drawn around selected entity
- background color of the canvas changes when switched to marking mode
- icons in context menu may change for items that have on/off state
- success or error message is displayed after import
- menu items may be disabled if user does not have appropriate rights

We have identified only one action that may take longer time to finish – import in some configurations of layout algorithm. However, any feedback we could provide in these cases would be diminished by the fact that the browser or browser tab usually becomes unresponsive each time the performance heavy JavaScript is being run.

Match between system and the real world

> The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

Data modeler has very little text, which can be found mostly in context menus. Menu options use terminology that should be understandable to all users of the modeler and most of the menu options also have icons which should aid in understanding the option. We have tried to match icons to as many options as possible and we believe that selected icons are good representation of the item.

User control and freedom

> Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Although unwanted states (selecting entity, initiation of XOR or ISA creation, text edit etc.) can user easily leave by clicking on the canvas or pressing *Esc* key, there is no support for undo and redo. This functionality was considered, but it was left for possible future extension.

Consistency and standards

> Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Data modeler is minimal web component. Although the usage of context menu is non-standard in classic web pages, it is not unheard of in more complex web applications. Morever, it is a common functionality in all major operating systems with graphical user interface. We consider both design and behaviour of the context menu to be consistent with context menus across various systems and their applications.

Error prevention

> Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

We have tried to design the modeler in a way that would prevent any need to show error messages to the user. Following list explains situation in which the error messages were necessary and also provides examples of situations which requires confirmation from the user:

- Import can fail out of several reasons beyond our control – uploading is turned off, user tries to upload file of wrong file type etc. These cases are handled by showing error message informing user about the cause of the error.
- During creation of XOR relationship, user may try to select relationship half that could not be part of the XOR. The attempt to prevent this is made by changing colors around relationship half: yellow is currently selected entity, red entities can not form XOR and green can form XOR. If user select incorrect relationship half anyway (or relationship half attached to different entity), error message is shown.
- User may try to leave the page, close the browser tab or browser itself, without exporting the diagram. To prevent the loss of the work, user is prompted whether he really wants to leave[21].
- Canvas may be cleared by user via context menu of the canvas. To ensure that the user does not accidentally delete all his work due to misclick, a prompt is shown to confirm the action.

[21] This behavior may be turned off. See table 3.2

Recognition rather than recall

> Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

All possible actions are always easily obtainable via context menu displayed by right click. Since there are no multi page dialogs and everything happens in one window, we argue that user does not need to remember anything for regular usage of the modeler. To obtain more information about advanced features – e.g. shortcuts, simple info view can be always displayed via icon in the upper right corner of the canvas and via *F1* key.

Flexibility and efficiency of use

> Accelerators – unseen by the novice user – may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

Some shortcuts were created to improve efficiency of advanced users. Most notable is cycling through attributes with *TAB* key, which allows both quick rename as well as creation of new attribute, or deletion of attribute by clearing its name. This area could be probably improved in the future based on how users will want to use the modeler.

Aesthetic and minimalist design

> Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Decision to create minimalistic design was made even before the designing process started. All focus is directed to the diagram itself and irrelevant information is never displayed.

Help users recognize, diagnose, and recover from errors

> Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Due to the inherent knowledge of the system we are not able to judge how well the error messages are interpreted by users with no or little prior

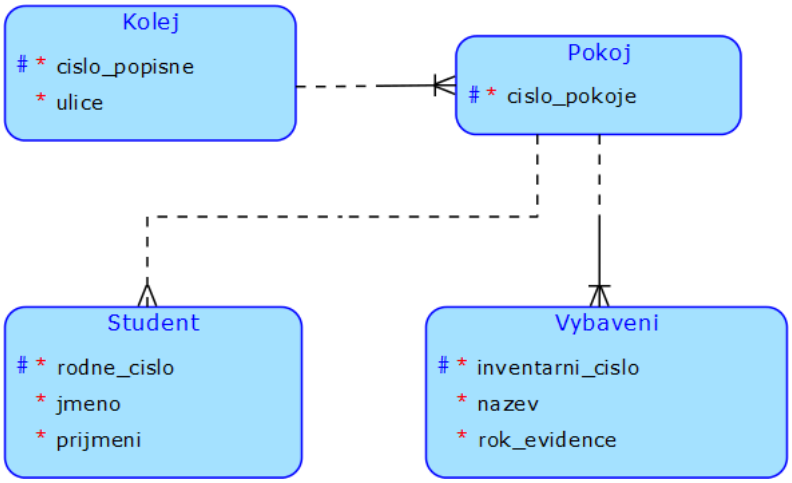


Figure 4.2: Reference solution of the task from user testing

experience with the modeler. However, all error messages are written in plain language and try to explain the cause of the problem. When possible, they also try to suggest a solution.

Help and documentation

> Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Tutorial, implemented to teach users how to use data modeler, was discussed in greater detail in section 3.8.

4.2 Testing with Users

Although theoretical evaluation of user interface may uncover issues of the interface and hint whether the design is good or bad, the software itself will not be used by experts in user interface design, who may have significantly different skillset, knowledge, requirements and point of view of real users of the application. Testing with real users may provide invaluable insight and show actual weaknesses and strengths of the interface.

We have conducted a usability test with 21 students of Database Systems course during one of the tests in semester. Data modeler was deployed into DBS Portal and one of several tasks was to create a logical model for simple problem. To create correct solution, students would need to create several

entities, attributes and relationships. They would also need to set entities names and change properties of attributes and relationships. Correct solution did not contain XOR relationship nor entity participating in ISA hierarchy. Reference solution may be seen in figure 4.2. Students did not have practical experience with new data modeler prior to the test, but some of them may have seen it during seminar.

Students were observed while working on a modeling task without any interaction with them, unless they got stuck and could not find the solution to their problem by themselves. It is important to note here that data modeler has been deployed with tutorial turned off. Once students were finished with the test, they were asked to complete a fifteen question survey.

4.2.1 Observation of Users

All students seemed to grasp overall design philosophy even with tutorial turned off and no prior explanation on how to use the application. They seemed to have no problem with using context menu even in web environment and all users learned that context menu exists by themselves.

The biggest usability issue we have found is that out of 21 students, three were not able to figure out how to create entity. One of them eventually did figure it out, but other two had to be told to "draw" the entity by clicking and dragging the mouse on the canvas. Even though they were not able to figure out the drawing mechanism, they successfully found context menu and information window, which currently contains list of available keyboard shortcuts. An interesting side note is that most of the other users, those that had no problem with starting to work on a diagram, were not observed to notice an icon toggling information window[22]. To solve this issue we will add an option for creating entity of default size to canvas' context menu. We will also add information about click and drag creation to the information window.

Another user was trying to create an attribute by double clicking the entity, which is probably a behaviour of some other data modeler. However, once he saw that this approach does not work, he had no problem adding attribute through context menu. However, this approach seems like a good shorthand and alternative to other methods of creating attributes and thus it will be implemented.

One user was observed to have a difficulty with changing position of the relationship anchor. Instead of dragging anchor itself he tried to drag the entire relationship, which instead creates control points. This probably stems from the learned behavior of SQL Developer Data Modeler, where sometimes dragging control point also changes the position of the anchor itself. However, we find that solution not to be very good, since it is not predictable. We be-

[22]Located in upper right corner of the canvas

lieve that our solution provides better control over the relationship drawing. In current implementation, when user moves mouse cursor over the anchor, the cursor changes to signify that anchor can be dragged. We consider this solution to be adequate.

On the edge between usability issue and bug is the activation of text edit after creation of relationships. Creation of new relationship is finished by clicking on target entity. If user clicks on an editable text (name of the entity or the attribute), the relationship is created and text is selected for edit. Same thing may also happen during ISA hierarchy creation. This behavior is unexpected and may confuse or annoy users and will be fixed.

Once the student was done with the test, teacher offered to take a quick look at his answers. This allowed us to make a small observations of the examinator role as well. Teacher was using mostly a mobile device – tablet – to take a quick look at the diagram and compare it with reference solution. If the two did not match, he tried to find what was the major difference. We were able to see that the layout algorithm works well enough, but in some cases it generated overlapping anchors, which could not have been moved by touch controls. For future work we suggest either adding support for touch controls or creating specialized interface for mobile devices.

Overall, the observations of users during their work with data modeler did not show any fundamental flaws in the design of the user interface. All users were able to use the modeler to create a diagram without any major help and minor issues that were found will be fixed.

4.2.2 Post-test Survey

Post-test survey was meant to provide better understanding of the users' experience with data modeler they just used. In fifteen questions we first tried to establish what experience the user had with other data modelers, then we tried to find out what was their experience with our implementation and in what areas it could be improved.

Question 1: What data modelers do you have experience with? What strengths and weaknesses they have in your opinion?

This question aimed to understand what prior experience with data modeling tools students had. As expected, all users that answered this question listed Oracle SQL Developer Data Modeler, while four users also listed Enterprise Architect, which they probably used in software engineering classes. Only eight students provided any opinion on these tools, generally found them to be counterintuitive and hard to working with, thus confirming that our focus on simplicity and minimalism was correct.

Question 2: What take you the most of the time during model creation? (e.g. creation of entities, addition of attributes, relationship setup, ...)

This question was meant to serve two purposes. First is to figure out what operations should we focus on in further development to help users more easily and more quickly achieve their task, while second purpose is give some insight to the user's opinion on how usable the system is in following questions.

Twenty users in total answered. 13 spend most time with setup of relationships, 4 with attributes and 1 spends equal amount of time on relationships and attributes.

From these answers it is clear the we should pay very close attention to relationship handling in further evaluation of the questionnaire results. Moreover, it looks like prioritizing improvements to relationship handling should be considered to improve user experience of most users.

Question 3: Have you found all functionality where you expected it? If no, what functionality was it?

Only two students answered this question negatively. First student expected to create an entity by single click. Although implementation of this feature is possible, we have found during early tests and development that this could easily lead to creation of entities by accident and worsening the overall user experience. Creation of entity via click can work in a software which offers *tools* that the user may select and use. An example of this implementation can be found in SQL Developer, which has a *New Entity tool* and *Select tool*. User may switch between these tools in a toolbar and thus the accidental creation of entity is unlikely. However, as we have already discussed, the observations showed that creation of entity could be an issue. As a compromise we will add – along with changes discussed in previous section – an option to create entity of standard size by double click.

Second student expected an option to rename attribute in context menu. Since in current implementation the attribute can be easily renamed by clicking on its name, we do not think that addition of unnecessary item to the menu would be good idea and it would go against minimalism and simplicity as a principles we set for our design.

Question 4: Rate following operations based on how easy to use you have found them to be, where 1 means very simple to use and 5 means very complicated to use

Answers to this question should reveal what parts of the application students found most difficult to use. Results have been aggregated in table 4.1. Operations were divided into three sections (entity, attribute, relationship), based on

Operation	Ratings count 1	2	3	4	5	Average rating
Creation of entity	18	2	.	.	.	1.10
Entity name edit	19	.	.	.	.	1.00
Change of entity's position	17	3	.	.	.	1.15
Entity resize	21	.	.	.	.	1.00
Average rating for entity operations						**1.06**
Creation of new attribute	13	7	1	.	.	1.43
Attribute name edit	15	.	1	.	.	1.13
Attribute setup	14	4	.	.	.	1.22
Average rating for attribute operations						**1.26**
Creation of new relationship	7	10	4	.	.	1.86
Relationship setup	8	7	3	.	.	1.72
Average rating for relationship operations						**1.79**

Table 4.1: Results of question 4 of post-test survey. Zeros were replaced with dot symbol for better readability

the object they relate to. To be able to compare results more easily, we have computed average rating for each option as well as for each section.

Overall we can see that users did not find any operation to be downright difficult to use. Problems with entity creation were already discussed and with the proposed changes we believe that the average rating 1.06 will further improve.

We notice two outliers in ratings for attribute operations, but it turns out that we have already discussed these in previous question. Student which gave a rating of 3 for creation of new attribute operation expected the attribute to be added by double click on the entity and user who had problems with rename was looking for rename option in context menu.

Creation and setup of relationships are the most complex operations users had to use to solve the task. Ratings of these operations were significantly worse compared to operations on entities and attributes, showing that there is a room for improvement, but they were still positive or neutral.

Question 5 & 6: Did you use any keyboard shortcuts? Why not? Did you expect any keyboard shortcuts? Which ones?

These questions aimed to find out more about how users use data modeler. Vast majority of users did not use any keyboard shortcuts. The reason they stated was that they did not know about them, on the other hand they had not listed any shortcuts they were missing. The fact that they were working in the data modeler for the first time, on a small model and in a stressful

environment may have also contributed to the fact that they were not even trying to use shortcuts.

However, two users stated that they tried to delete entity with *Del* key which did not work. This answer was surprising, because this shortcut was supposed to be supported. Further investigation showed that there is a bug in implementation and the keyboard shortcut was not properly working in Chrome browser.

Questions 7 through 10

These four questions focused on students' overall experience with data modeler. We have asked students how simple to use they consider the modeler to be and whether they could imagine using the modeler outside the Database Systems course. Question 9 asked students whether data modeler provided sufficient feedback.

Students have agreed that the data modeler is easy to use – 16 students stated they found it very easy to use, 5 students somewhat easy to use. Only two students were not sure whether they would or would not use data modeler on other projects, while rest was positive. No user complained about insufficient feedback.

The most useful, however, was the question 10. Users were asked about any unexpected behavior they may have encountered. Thanks to their answers we have realised that the default partiality of the relationship halves was swapped – the identifying half should be required while the other one optional.

Questions 11 through 15

The end of the survey was dedicated to comparison with other data modeling tools. We wanted to know what do they consider to be the biggest strengths and weaknesses and asked them to compare our data modeler to other modelers. We also wanted to know what one thing would they change primarily. Last question was left for any additional comments they might have had.

Since students had not have too much experience with other modelers we did not see any detailed comparisons, although some students stated they found our data modeler easier to use than Oracle Data Modeler. No weaknesses were mentioned, while intuitiveness and simplicity were listed by many students as strengths, which is exactly what the design was striving for.

4.2.3 Summary

Testing with real users of the modeler has proven to be incredibly useful. Despite the fact that we were not able to cover all of the functionality of the data modeler, being able to see how real users interact with the application has allowed us to verify our design, discover some usability issues and figure out new ways for improving the overall user experience.

We are satisfied with the results. All users successfully finished their task without the need for tutorial, user guide or explanation of how to work with the modeler. Post-test survey allowed users to provide their own opinion, which was overwhelmingly positive, although there certainly is room for improvement.

As a result of this test, following changes have been made:

- Entity can now also be created from canvas' context menu or by double click on the canvas
- Instructions for entity creation were added to the info window
- Editing of text is no longer initiated immediately after new relationship or ISA hierarchy is created
- Attribute may be added by double click on the entity
- Default partializ of the relationship has been swapped to better match users' expectations
- Bug where entity would not be deleted by *Del* key in Chrome was fixed

Chapter 5

Future Work

Although data modeler has been successfully released and deployed to a DBS Portal, there is still a room for future development. Source code of the modeler was made available at the faculty's GitLab repository [23], making it possible to continue development by others. We will also release data modeler publicly on the authors GitHub repository[24]. It is possible that future development will branch, since faculty's needs may be different than those of general public.

We believe the right decisions that would ensure future development is possible were made, e.g. using MVC architecture or manipulating SVG DOM directly from JavaScript without any intermediary library, which removes the unnecessary dependency. In this section we would like to suggest a few directions for possible future development that we think might be worthwhile, in no particular order.

The data modeler was finished towards the end of the semester and students were able to use it during final tests and exams, where saving the state of semifinished work is not a common use case. To better support work on larger projects and tasks, e.g. semestral project, an option to save full state of the diagram could be implemented. That means not only storing the underlying model but also visual information. This would have to be implemented in addition to current export, because model needs to be automatically comparable (by comparing string representation of the model). It might also be possible to implement auto save feature.

To improve the usability of the modeling itself, multi select and undo/redo functionality could be considered. Both of these options significantly improve user experience. Multi select would be most valuable to save time when working on larger diagrams since it would allow users to select multiple entities at once and move them to better position. Undo and redo are significant enhancements that lowers the severity of mistakes that user makes, since they can easily get back to previous state.

[23] `https://gitlab.fit.cvut.cz/malecold/DBSDM`
[24] `https://github.com/tfedor/datamodeler`

During the user testing we have learned that examinators may want to mark diagrams from mobile devices. Currently it is possible to view diagrams, but manipulation is tricky to say the least, because user interface of the modeler was built around mouse. To support work from mobile devices, either the support for touch must be added, or entirely new user interface must be created as a different view. The biggest issue would be the handling of mouse movement. Common behaviour of mobile browsers is that short tap works as mouse click, while long tap usually works as right click, which in our case shows context menu. Implementation of drag, however, is not that straightforward. While it would be possible to simulate it via touch events, it may not be desirable, because it would hinder browser's native functionality – page scrolling and zooming, to be more specific. Possible implementation would be to simulate drag by two taps: first tap selects object, second tap sets the target position.

In this thesis we have made some assumptions about the performance, however, the techniques we have used to improve performance were not properly tested and it is not clear whether they helped or not. For future work we suggest measuring the impact of `defs` and `use` elements and whether it is better to use shared elements for, e.g. background of entities or not. However, we have tried to display hundreds of canvas elements with imported models on one page and the only observed performance issue was during import, when we tried to import all models at once.

Furthermore, we suggest improvements to both anchor placement and layout algorithms. It might be possible to improve anchor placement algorithm to reduce both overlapping and crossing relationships by taking into consideration the placement of the other entity, or by resizing the entity itself. Layout algorithm might try to force grid-like placing of entities more thoroughly, however this stems from personal preferences, and not from proper analysis of what makes diagram visually pleasurable.

Regarding the user interface, we were not able to test all functionality of the data modeler. More tests should be conducted to cover more advanced functionality. More tests on new users should be done as well, to discover the real need for the tutorial, since as we have discussed in chapter 4, students were able to comfortably use data modeler even without it.

Relational models are a vital part of database development and design. Logical models currently supported by data modeler are preceding them in the database design process and some tools are able to convert between them. In section 3.2 we have already explained how the flexibility of the MVC architecture allows the expansion of the modeler to support other type of models, which might be a good long term goal for future development. With good support of relational modeling and conversion between logical and relational model, students of Database Systems course might be able to work on their semestral project fully in the DBS Portal, without external software.

It might also be interesting to explore the possibility to use data modeler as

standalone application. Since data modeler is fully client based, it is already possible to work offline, but from the browser. It might be possible to use solutions like Electron[25] or AppJS[26] to create standalone application relatively easily, which would provide new possibilities for entire modeler. Due to access to file system, it might be possible to add support for whole projects consisting of multiple models, for example.

[25] http://electron.atom.io/
[26] http://appjs.com/

Conclusion

This thesis has discussed the development of web component for drawing ER diagrams – data modeler. The main purpose of the modeler is to support the education process of the Database Systems course at Faculty of Information Technology of Czech Technical University. During this mandatory course, students are among other things tasked with creation of logical models of smaller databases, both for their semestral project as well as during tests, which check their comprehension of the subject. Data modeler will thus be used in different contexts mainly by two distinct groups of users: students and examinators.

Currently, students work mostly in Oracle SQL Developer Data Modeler, a full-featured database IDE to finish their semestral work. It is possible that data modeler, as part of a web portal for education support of this course (DBS Portal) will replace this software in the future. When working on the project, students would be allowed to use almost all features of the modeler. On the other hand, the functionality is restricted during tests or exams (e.g. students may not export or import models), where the modeler currently brings the biggest value.

In the context of tests and exams we must also consider a second group of users, examinators. Examinators create reference solution of a given task and check students' solutions after. The data modeler supports their role by various means. Some of them are: allowing multiple diagrams to be displayed on a single page, automatic comparison of saved models as well as providing marking mode, in which examinators may mark incorrect parts of students' solution.

The development of data modeler started by the analysis, during which we have assessed requirements of the data modeler. Analysis of the current state of data modeler of the DBS portal followed, we have found out that the current state is unfit in regards of functionality, user interface and future extensibility. The decision was made to develop data modeler anew.

To be able to design good user interface we have analysed SQL Developer

Data Modeler, as a primary software used during the course, and other web-based modelers, both universal diagramming tools and specific solutions for database modeling. The final user interface was designed according to minimalism, simplicity and efficiency principles.

The data modeler was developed in JavaScript and built on HTML, CSS and SVG. In the Implementation chapter we have explained how the concepts of Object-oriented programming, namespacing, classes and encapsulation, were achieved in prototype-based scripting language. Further, we have discussed Model-View-Controller architecture and how it should provide good basis for future development. At last, this chapter went into further implementation details and solutions to some of the encountered issues.

After implementation, the data modeler was deployed to DBS Portal where it replaced old, unfit implementation. The interface was heuristically evaluated to verify it does not contain major issues, after which proper user test was conducted. Twenty one students worked in the new version of data modeler during the test at the end of the semester. Their observation and post-test survey discovered some of the usability issues, most of which were fixed since, but none of them were major. Students were able to use data modeler without any prior explanation and their feedback was overwhelmingly positive.

Although the goals set were accomplished and new data modeler, which is now also publicly available, is helping students and teachers during the Database Systems course, there is still room for improvement. This thesis finishes by outlining several possible extensions, that we believe would further improve the usability and utility of the modeler.

In conclusion, we have developed a new data modeler, which was deployed to the education support portal of the Database Systems course and is currently being used during exams. The user interface design as well as the modeler's functionality was verified by user test with students of this course. Based on the results of this test we may conclude that the new modeler is easy to use and does not have any major usability issues or missing functionality.

Bibliography

[1] Course description - BI-DBS. [cit. 2016-12-20]. Available from: http://bk.fit.cvut.cz/en/predmety/00/00/00/00/00/00/01/12/24/p1122406.html

[2] DB-Engines Ranking. [cit. 2016-12-20]. Available from: http://db-engines.com/en/ranking

[3] DB-Engines Ranking per database model category. [cit. 2016-12-20]. Available from: http://db-engines.com/en/ranking_categories

[4] Chen, P. P.-S. The Entity-relationship Model–Toward a Unified View of Data. *ACM Trans. Database Syst.*, volume 1, no. 1, Mar. 1976: pp. 9–36, ISSN 0362-5915, doi:10.1145/320434.320440. Available from: http://doi.acm.org/10.1145/320434.320440

[5] Halpin, T. Entity Relationshop modelling from an ORM perspective: Part 1. *Journal of Conceptual Modeling*, , no. 11, 1999, [cit. 2016-12-19]. Available from: http://www.orm.net/pdf/JCM11.pdf

[6] Slavotínek, J. *Webová komponenta na kreslení ER diagramů*. Bachelor's thesis, Faculty of Information Technology of Czech Technical University in Prague, 2016.

[7] Hunka, J. Hodnocení vedoucího závěrečné práce, supervisor's evaluation of Jiří Slavotinek's bachelor's thesis "Webová komponenta na kreslení ER diagramu".

[8] Halaška, I. Posudek oponenta závěrečné práce, reviewer's report of Jiří Slavotinek's bachelor's thesis "Webová komponenta na kreslení ER diagramu".

[9] Oracle SQL Developer Overview. [cit. 2016-12-23]. Available from: http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index-097090.html

[10] Creately. [accessed 2016-12-24]. Available from: https://creately.com/

[11] Vertabelo. [accessed 2016-12-24. Available from: http://www.vertabelo.com

[12] Vertabelo Features. [accessed 2016-12-24. Available from: http://www.vertabelo.com/features

[13] Draw.io. [accessed 2016-12-24]. Available from: https://www.draw.io/

[14] TinyModeler. [accessed 2016-12-24]. Available from: http://tinymodeler.com/

[15] Raskin, J. *The Humane Interface: New Directions for Designing Interactive Systems*. ACM Press Series, Addison-Wesley, 2000, ISBN 9780201379372. Available from: https://books.google.sk/books?id=D39vjmLfO3kC

[16] Saternos, C. *Client-Server Web Apps with JavaScript and Java: Rich, Scalable, and RESTful*. O'Reilly Media, Inc., 3 2014, ISBN 9781449369293.

[17] Rajan, S. Introduction to Functional JavaScript. [accessed 2016-12-29]. Available from: https://medium.com/functional-javascript/introduction-to-functional-javascript-45a9dca6c64a

[18] Ecma International. *ECMAScript® 2016 Language Specification*. [accessed 2016-12-28]. Available from: https://tc39.github.io/ecma262/

[19] ECMAScript 5 compatibility table. [accessed 2016-12-28]. Available from: http://kangax.github.io/compat-table/es5/

[20] Mozilla Developer Network. *Javascript Guide – Introduction*. [accessed 2016-12-28]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction

[21] Node.js. [accessed 2016-12-28]. Available from: https://nodejs.org/en/

[22] Definition of namespace in English – Oxford Dictionary. [accessed 2016-12-29]. Available from: https://en.oxforddictionaries.com/definition/namespace

[23] Mozilla Developer Network. *Introduction to Object-Oriented JavaScript*. [accessed 2016-12-29]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

[24] Rauschmayer, A. *Exploring ES6 – Upgrade to the next version of JavaScript*. [accessed 2016-12-29]. Available from: http://exploringjs.com/es6/

[25] Mozilla Developer Network. *JavaScript Reference – new operator*. [accessed 2016-12-29]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new

[26] Crockford, D. Private Members in JavaScript. [accessed 2016-12-29]. Available from: http://www.crockford.com/javascript/private.html

[27] Osmani, A. *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide*. O'Reilly Media, July 2012, ISBN 9781449331818. Available from: https://addyosmani.com/resources/essentialjsdesignpatterns/book/

[28] Miraglia, E. A JavaScript Module Pattern. [accessed 2016-12-29]. Available from: http://yuiblog.com/blog/2007/06/12/module-pattern/

[29] jQuery Learning Center – Code Organization Techniques. [accessed 2016-12-29]. Available from: https://learn.jquery.com/code-organization/concepts/

[30] Spencer, E. JavaScript Module pattern – overused, dangerous and bloody annoying. 2009, [accessed 2016-12-29]. Available from: https://edspencer.net/2009/10/05/javascript-module-pattern-overused-dangerous-and-bloody-annoying/

[31] 9. Classes – Python 3.6.0 documentation. [accessed 2016-12-29]. Available from: https://docs.python.org/3/tutorial/classes.html

[32] World Wide Web Consortium. *HTML5 Specification – The canvas element*. [accessed 2016-12-29]. Available from: https://www.w3.org/TR/2014/REC-html5-20141028/scripting-1.html#the-canvas-element

[33] Mozilla Developer Network. *Canvas API*. [accessed 2016-12-29]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

[34] World Wide Web Consortium. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. [accessed 2016-12-29]. Available from: https://www.w3.org/TR/SVG11/Overview.html

[35] World Wide Web Consortium. *SVG Current Status*. [accessed 2016-12-29]. Available from: https://www.w3.org/standards/techs/svg#w3c_all

[36] World Wide Web Consortium. *HTML Current Status*. [accessed 2016-12-29]. Available from: https://www.w3.org/standards/techs/html#w3c_all

[37] World Wide Web Consortium. *Coordinate Systems, Transformations and Units – SVG 1.1 (Second Edition)*. [accessed 2016-12-30]. Available from: https://www.w3.org/TR/SVG11/coords.html#EstablishingANewViewport

[38] Apple Inc. *Model-View-Controller*. [accessed 2016-12-30]. Available from: https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

[39] Microsoft Corporation. *Model-View-Controller*. [accessed 2016-12-30]. Available from: https://msdn.microsoft.com/en-us/library/ff649643.aspx

[40] Google. *MVC Architecture – Google Chrome*. [accessed 2016-12-30]. Available from: https://developer.chrome.com/apps/app_frameworks

[41] Hu, Y. Efficient and High Quality Force-Directed Graph Drawing. *The Mathematical Journal*, volume 10, 2006: pp. 37–71.

[42] Kobourov, S. Force-Directed Drawing Algorithms. In *Handbook of Graph Drawing and Visualization*, edited by R. Tamassia, chapter 12, CRC Press, 2013.

[43] Introducing JSON. [accessed 2017-01-01]. Available from: http://www.json.org/

[44] JSZip. [accessed 2017-01-01]. Available from: https://stuk.github.io/jszip/

[45] Nielsen, J. *10 Usability Heuristics for User Interface Design*. Nielsen Norman Group, 1 1995. Available from: https://www.nngroup.com/articles/ten-usability-heuristics/

[46] Shneiderman, B. The Eight Golden Rules of Interface Design. 5 2016, [accessed 2017-01-02]. Available from: https://www.cs.umd.edu/users/ben/goldenrules.html

[47] Font Awesome. [accessed 2017-01-02]. Available from: http://fontawesome.io/

[48] saveSvgAsPng. [accessed 2017-01-02]. Available from: https://github.com/exupero/saveSvgAsPng

[49] Nielsen, J.; Molich, R. Heuristic Evaluation of User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, New York, NY, USA: ACM, 1990, ISBN 0-201-50932-6, pp. 249–256, doi:10.1145/97243.97281. Available from: `http://doi.acm.org/10.1145/97243.97281`

[50] Žikovský, P. Usability. Special Testing. Personas, unpublished Lecture in User Interface Design.

[51] Molich, R.; Nielsen, J. Improving a Human-computer Dialogue. *Commun. ACM*, volume 33, no. 3, Mar. 1990: pp. 338–348, ISSN 0001-0782, doi:10.1145/77481.77486. Available from: `http://doi.acm.org/10.1145/77481.77486`

Appendix A

Acronyms

CSS Cascading Style Sheets.

DOM Document Object Model.

ER Model Entity-Relationship Model.

HTML HyperText Markup Language.

IDE Integrated development environment.

IIFE Immediately-invoked function expressions.

JSON JavaScript Object Notation.

MVC Model-View-Controller.

OOP Object-oriented programming.

SQL Structured Query Language.

SVG Scalable Vector Graphics.

XML Extensible Markup Language.

XOR Exclusive OR.

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `DP_Fedor_Tomas_2017.pdf` ..... the thesis text in PDF format
- `impl` ..... the directory of data modeler source
  - `dist` ..... the directory of JavaScript codes for distribution
  - `libs` ..... the directory of 3rd party libraries and codes
  - `src` ..... the directory of JavaScript source codes
  - `styles` ..... the directory of CSS files
- `text` ..... the directory of thesis source code in LaTeX format
  - `imgs` ..... images used in this thesis

Appendix C

Examples of Layout Algorithm

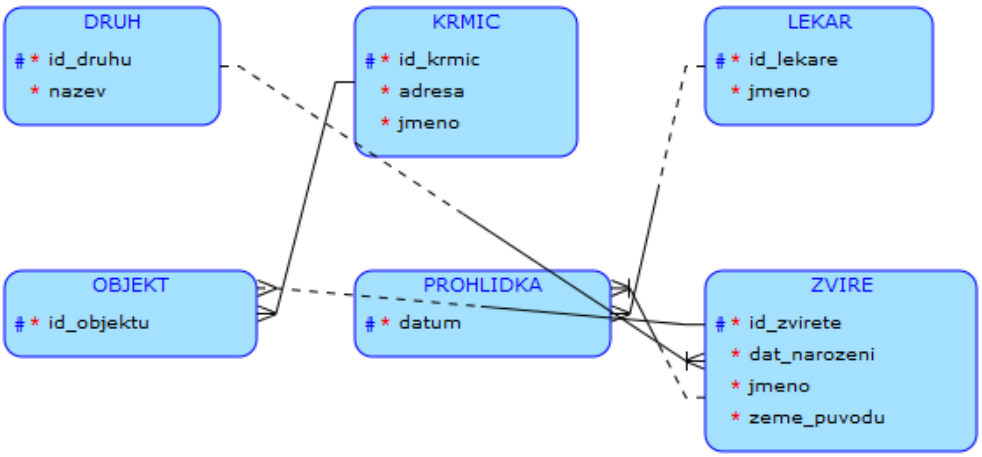


Figure C.1: Initial configuration of diagram 1

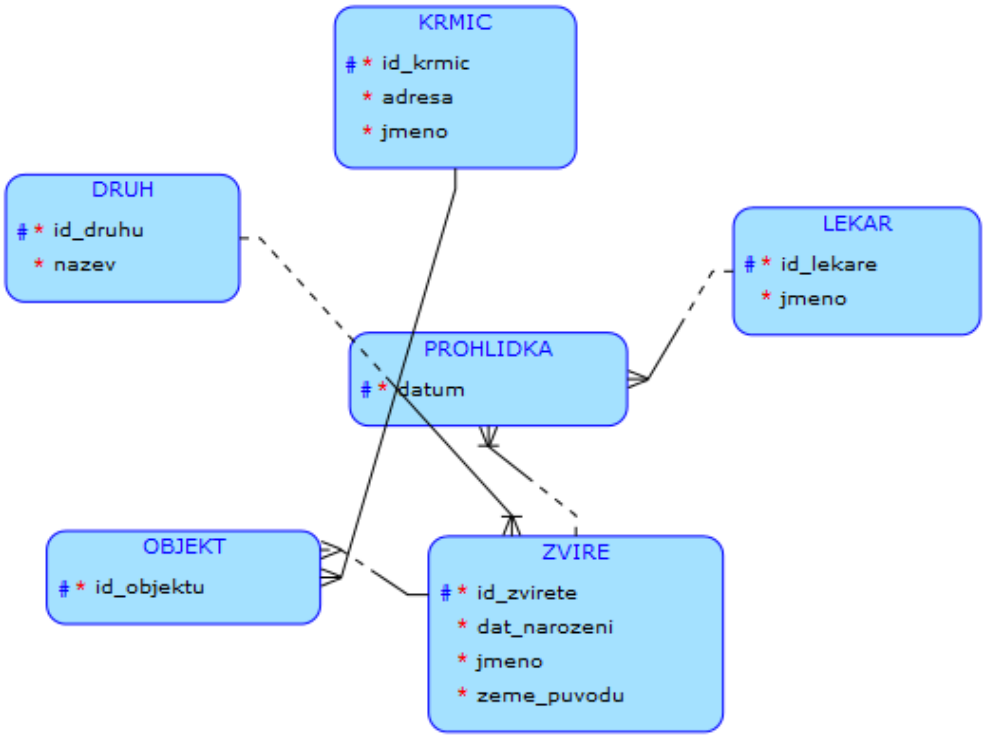


Figure C.2: Diagram 1 after 25 iterations

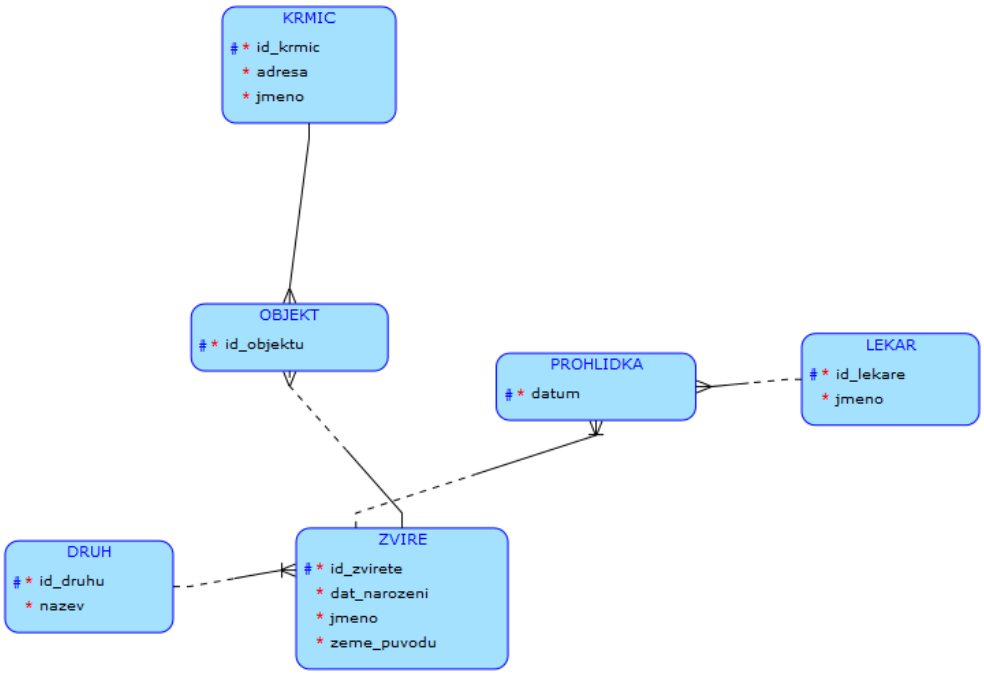


Figure C.3: Diagram 1 after 100 iterations

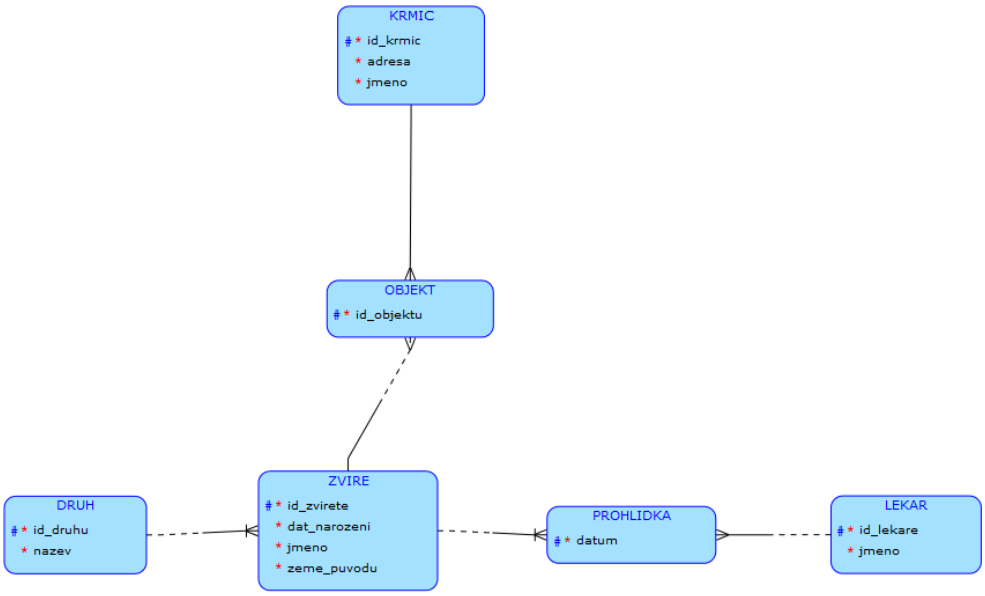


Figure C.4: Diagram 1 after 500 iterations

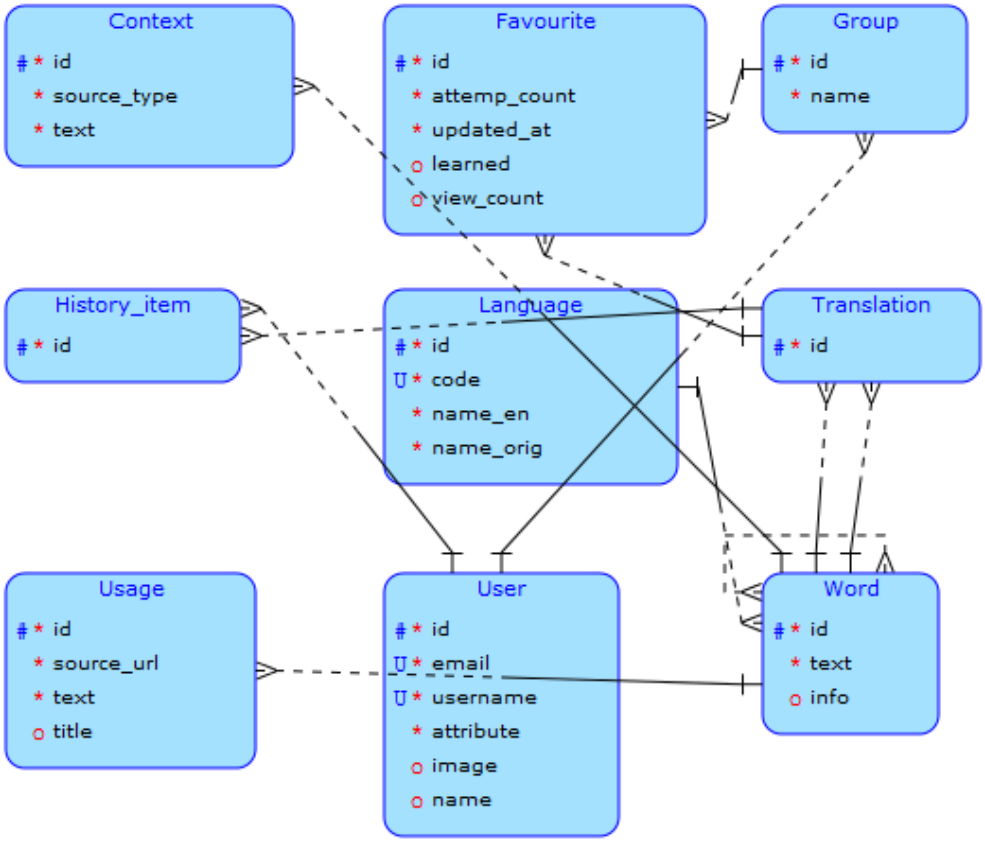


Figure C.5: Initial configuration of diagram 2

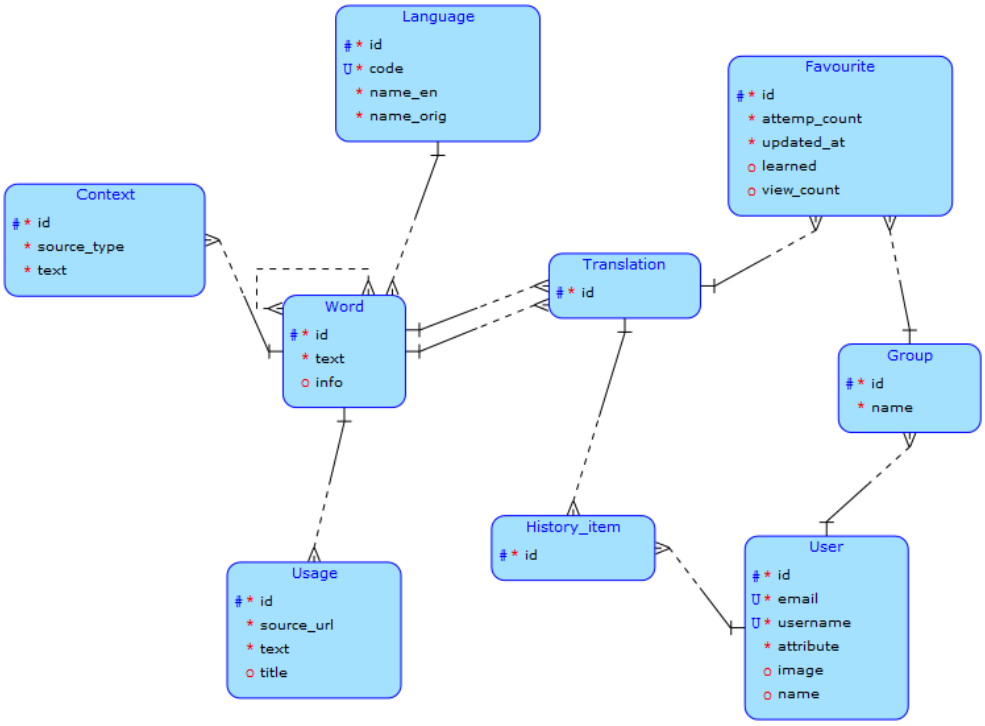


Figure C.6: Diagram 2 after 100 iterations

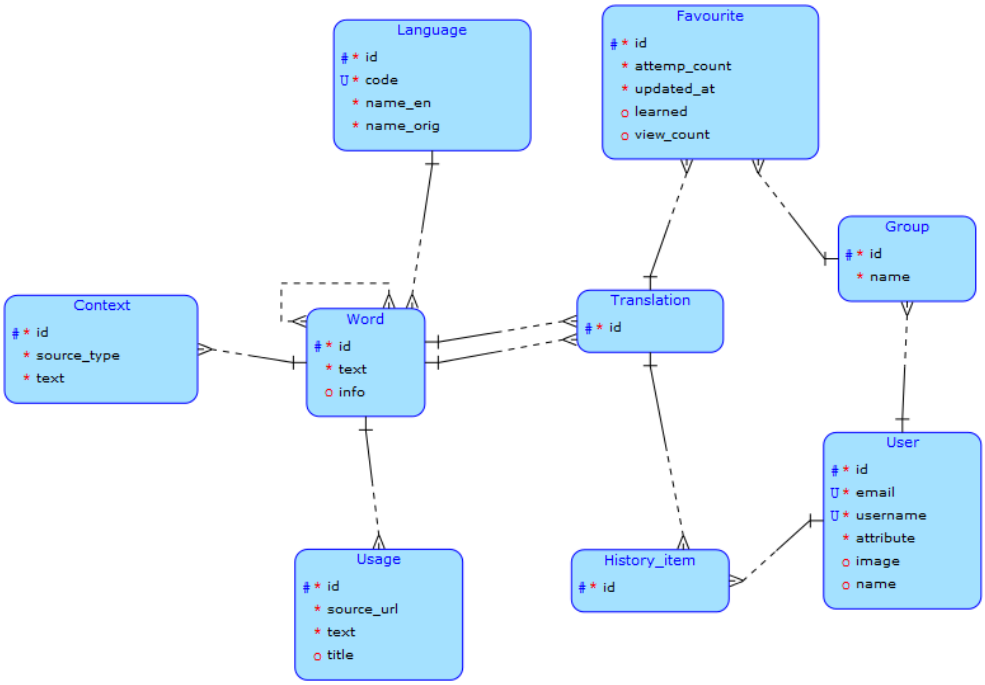


Figure C.7: Diagram 2 after 500 iterations

Appendix D

Screenshots

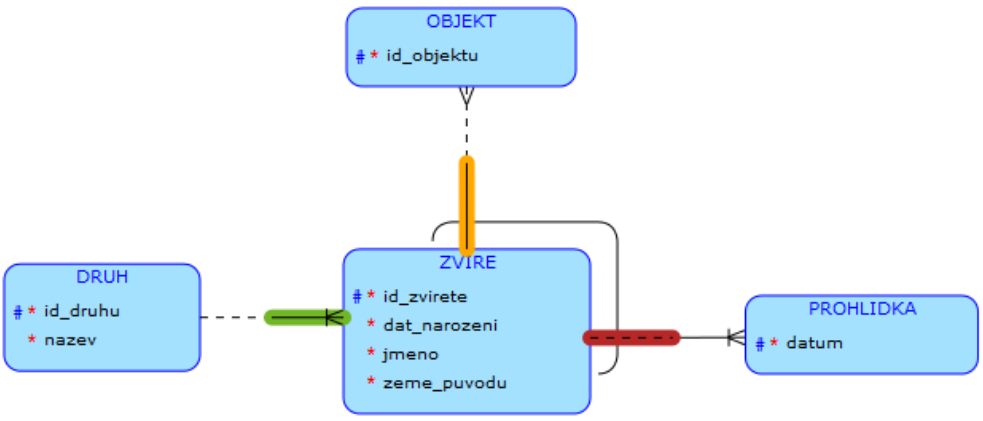


Figure D.1: Screenshot of diagram during creation of XOR relationship. User has initiated XOR creation on yellow relationship half. Green relationship half may be added to XOR, while red relationship half can not, because it is already included in it.

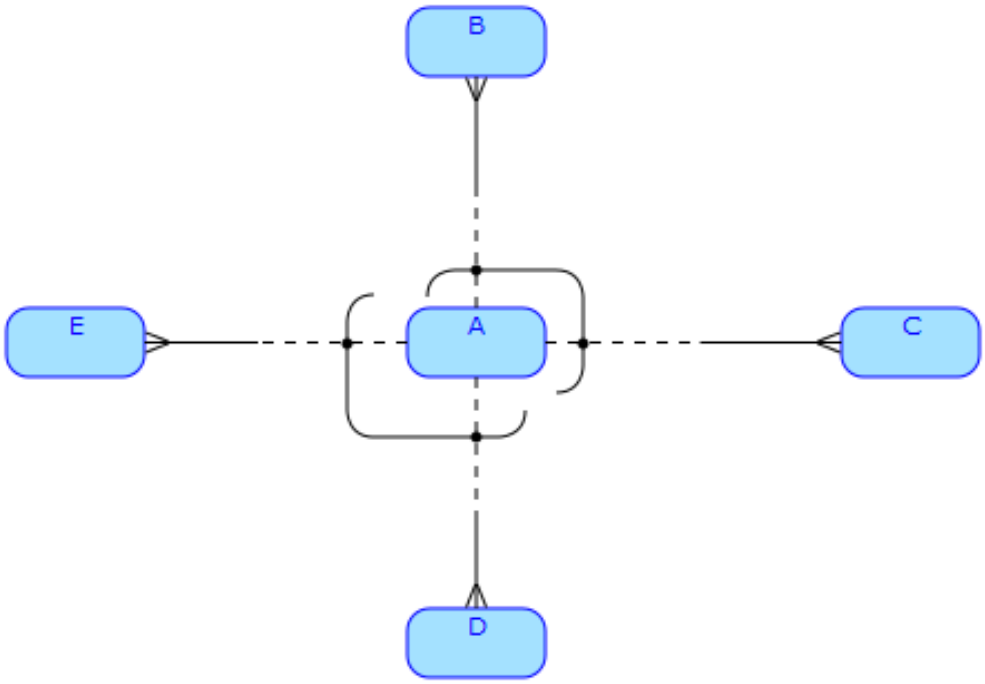


Figure D.2: Example of entity with two XOR relationship

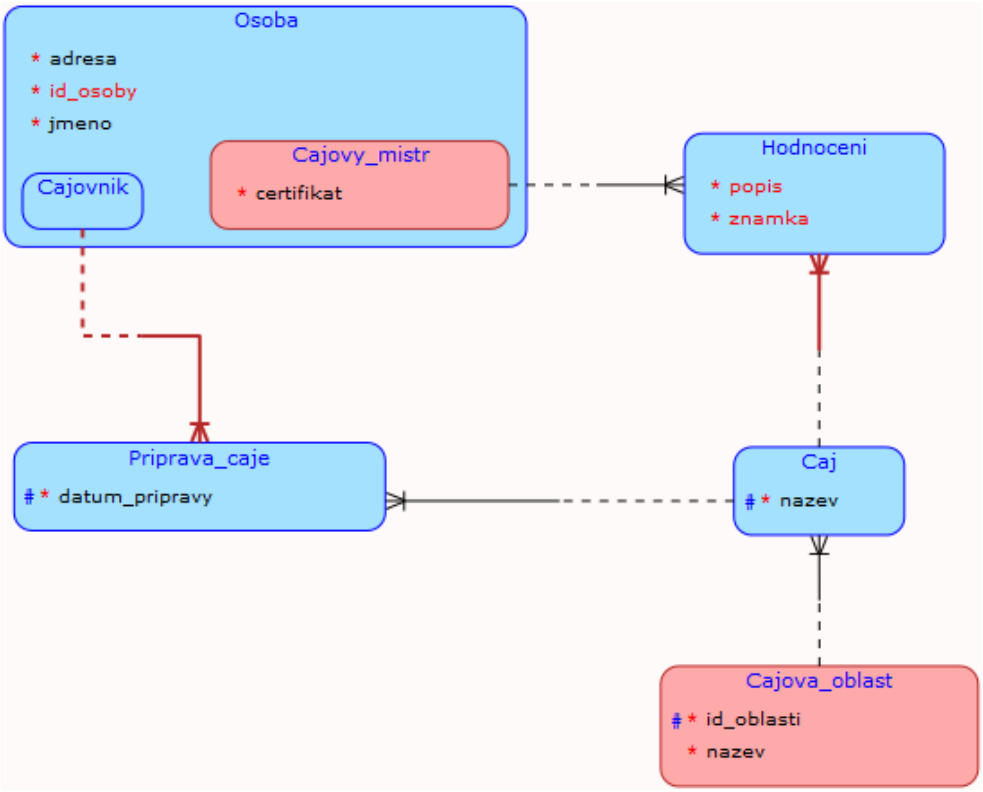


Figure D.3: Diagram in marking mode, with parts of the diagram marked as incorrect