



**MATEMATICKO-FYZIKÁLNÍ
FAKULTA**
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Dennis Pražák

**Konceptuální modelování pomocí
schematických kategorií**

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Martin Svoboda, Ph.D.

Studijní program: Informatika

Studijní obor: Programování a vývoj software

Praha 2023

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 20. 7. 2023

.....

Podpis autora

Děkuji vedoucímu RNDr. Martinu Svobodovi, PhD., za odborné vedení práce, časté konzultace a všechny poskytnuté rady a podněty.

Název práce: Konceptuální modelování pomocí schematických kategorií

Autor: Dennis Pražák

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Martin Svoboda, Ph.D.

Abstrakt: Tato práce se zabývá vývojem grafické aplikace pro konceptuální modelování databázových schémat bez předem známého paradigmatu. Účelem konceptuálního modelu je umožnit popis struktury dat na abstraktní úrovni nezávisle na jejich fyzickém uložení. V dnešní době se však často využívá mnoho různých logických modelů databázových systémů. Vyjadřovací prostředky známých konceptuálních modelů jako ER a UML někdy na popis struktury dat pro cílové logické modely nestačí. Využijeme proto nedávno vzniklého konceptu zvaného schematické kategorie, které jsou obecnější, mají vyšší vyjadřovací sílu a unifikují modelování dat pro různé databázové systémy včetně těch multi-modelových. Schematické kategorie tímto rovněž mažou hranice mezi konceptuální a logickou vrstvou datového modelování. Výsledná grafická webová aplikace umožňuje modelovat schémata ER modelu, který je velmi dobře známý, a jejich automatický převod na schematické kategorie s uživatelsky přívětivou vizualizací.

Klíčová slova: konceptuální modelování, schematická kategorie, databázové systémy

Title: Conceptual modeling using schema categories

Author: Dennis Pražák

Department: Department of Software Engineering

Supervisor: RNDr. Martin Svoboda, Ph.D.

Abstract: This thesis concerns the development of a graphical application for conceptual modeling of database schemas without prior knowledge of the target database paradigm. The purpose of a conceptual model is to allow describing the abstract structure of data independently of their physical storage. These days, a lot of different logical data models are used in database systems. The expressive power of well-known conceptual models like ER or UML is sometimes insufficient for describing the structure of data for target logical models. For this reason, we exploit a recent concept called schema categories, which is more general, has a higher expressive power, and unifies data modeling for different database systems, including the multi-model ones. Furthermore, schema categories erase the border between the conceptual and logical layers of data modeling. The resulting graphical web application allows modeling schemas of the well-known ER model and their automatic conversion into schema categories with a user-friendly visualization.

Keywords: conceptual modeling, schema category, database systems

Obsah

Úvod	3
1 Existující nástroje	5
1.1 Srovnávací kritéria	5
1.2 diagrams.net	6
1.3 drawSQL	10
1.4 ERDPlus	12
1.5 nomnoml	14
1.6 Visual Paradigm Online	16
1.7 Závěr existujících řešení	17
2 Teoretický rámec	20
2.1 Entity-Relationship	20
2.2 Schematická kategorie	23
2.2.1 Motivace	23
2.2.2 Teorie kategorií	24
2.2.3 Schematická kategorie	24
2.3 Vizualizace schematické kategorie	27
2.4 Převod ER na schematickou kategorii	28
3 Specifikace	31
3.1 Požadavky	31
3.1.1 Funkční požadavky	31
3.1.2 Nefunkční požadavky	33
3.2 Entity	33
3.3 Procesy	37
3.4 Koncept	40
3.5 Třídy	41
3.6 Scénáře	47
4 Implementace	51
4.1 Použité technologie	51
4.2 Uživatelská dokumentace	53
4.3 Technická dokumentace	55
4.4 Vývoj	56
4.5 Instalace a nasazení	56
Závěr	58
Seznam použité literatury	59

Seznam obrázků	62
Seznam tabulek	63
A Přílohy	64
A.1 Elektronické přílohy	64

Úvod

Vývoj informačních a databázových systémů se neustále posouvá vpřed. Při jejich návrhu je zásadním krokem konceptuální modelování, které posléze vede na tvorbu databázového schématu. Ke konceptuálnímu modelování existuje několik uznávaných přístupů, jako je Entity-Relationship (ER) [1] a Unified Modeling Language (UML) [2]. Tyto modely však byly navrhovány v dobách, kdy relační databáze vládly softwarovým systémům, poněvadž tabulky byly nejvhodnějším logickým modelem pro potřeby doby.

V posledních letech se v oboru začalo užívat rozličných databázových paradigmat. Tím v praktické rovině myslíme zejména různé modely na logické vrstvě modelování. Každý takový model má svůj účel, své slabé a silné stránky. Zkušený softwarový inženýr vybere pro každý systém tu nejvhodnější databázi s nejvhodnějším logickým modelem. Někdy se dokonce modelů v jednom systému používá více najednou, aby každý mohl spravovat tu část dat systému, pro kterou je vhodný.

Mezi nejznámější logické modely databázových systémů patří relační model (například databázový systém MySQL¹), key-value model (Redis²), wide column model (Apache Cassandra³), dokumentový model (MongoDB⁴) a grafový model (Neo4J⁵). Kromě toho existují ještě např. fulltextové vyhledávací databáze (Elasticsearch⁶) a multi-modelové databáze (FaunaDB⁷), které používají různé relační modely, respektive jich používají více najednou.

Naše stávající prostředky konceptuálního modelování nejsou mnohdy dostačující k modelování pro nerelační databáze. Ukazuje se mimo jiné, že byly navrženy s relačními databázemi na mysli, a jsou tak bohužel svázány s modelem logické vrstvy. Při konceptuálním modelování se chceme věnovat pouze vymezení a představě o části světa, na kterou vymezujeme svůj diskurz, nikoli na logickou strukturu dat v cílové databázi.

Utopií, které se snažíme dosáhnout, je mít jediný unifikovaný model, který má jediné rozhraní, jediný způsob modelování dat a jediný dotazovací jazyk v konceptuální vrstvě. Jde o splynutí konceptuální a logické vrstvy do jedné, ve které se pracuje unifikovaným konceptuálním způsobem. Až ve fyzické vrstvě se zvolí vhodná struktura podle potřeby dat.

Prvním hmatatelným krokem k přiblížení k tomuto cíli je schematická kategorie, kterou navrhli Martin Svoboda, Pavel Čontoš a Irena Holubová [3]. Jedná se o prostředek k modelování struktury dat, který je založený na teorii kategorií [4].

Cílem této práce je vytvořit softwarový nástroj, který umožní návrh konceptuálních schémat pomocí prostředků ER modelu a jeho převod na schematickou kategorii. Přiblíží tak tento nový koncept softwarovým inženýrům a analytikům, kteří jsou zběhlí v modelování pomocí ER. Nástroj dále usnadní výzkumníkům experimentování se schematickými kategoriemi.

¹<https://www.mysql.com/>

²<https://redis.io/>

³<https://cassandra.apache.org/>

⁴<https://www.mongodb.com/>

⁵<https://neo4j.com/>

⁶<https://www.elastic.co/>

⁷<https://fauna.com/>

Práce je rozdělena do několika částí. Nejprve prozkoumáme a zanalyzujeme existující nástroje, které slouží ke konceptuálnímu modelování (Kapitola 1). Dále definujeme a popíšeme upravené varianty ER a schematické kategorie navržené vedoucím práce, společně se způsobem vizualizace schematické kategorie a podrobnější motivací za schematickou kategorií (Kapitola 2). Posléze navrhne specifikaci zamýšleného softwarového nástroje na tvorbu diagramů konceptuálních schémat pomocí ER s konverzí do schematické kategorie a vizualizací schematické kategorie (Kapitola 3). Specifikace bude zahrnovat funkční a nefunkční požadavky, konceptuální model, procesy, koncept řešení, diagram tříd a případy užití. Poté popíšeme implementaci a uvedeme dokumentaci tohoto softwarového nástroje včetně návodu k užití (Kapitola 4). Nakonec zhodnotíme výsledky práce.

1. Existující nástroje

V této kapitole zanalyzujeme některé existující nástroje pro tvorbu diagramů a porovnáme je dle navržených kritérií. Žádné nástroje pro naše pojetí konceptuálního modelování neexistují, takže se zde budeme věnovat jiným, ale podobným nástrojům.

Všechny analyzované nástroje jsou dostupné online ve webovém prohlížeči, jelikož v praktické části bude implementován nástroj pro stejné prostředí. Nástroje, které budeme porovnávat, jsou

- diagrams.net [5] vhodné pro tvorbu libovolných diagramů,
- drawSQL [6] určené pro tvorbu relačních schémat,
- ERDPlus [7] k vytváření zejména ER diagramů [1].
- nomnoml [8] k tvorbě UML diagramů [2] psaním textu,
- Visual Paradigm Online [9] pro tvorbu libovolných diagramů.

Před představením existujících nástrojů určíme srovnávací kritéria, dle kterých budeme nástroje analyzovat.

1.1 Srovnávací kritéria

Prvním kritériem pro porovnání nástrojů je jejich kategorie, která vypovídá o účelu nástroje a cílové skupině zákazníků. Základní kategorie jsou:

- konceptuální vrstva – tyto nástroje jsou většinou určené pro tvorbu ER diagramů, případně jiným způsobem modelují vztahy a atributy entit, na které při datovém modelování vymezujeme svůj diskurz,
- logická vrstva – tyto nástroje umožňují tvorbu diagramů s ohledem na typ struktur, v kterých jsou data uchovávána, např. relační databáze,
- kresba libovolných diagramů – nástroje, které nejsou omezeny téměř žádným standardem či konvencí a umožňují kresbu libovolných diagramů jakožto obrázků,
- kresba omezených diagramů – nástroje, které umožňují kresbu diagramů omezených na existující modely (ER, UML, ...).

Dalším kritériem je typ úložiště. Nástroje mohou ukládat svá data do paměti prohlížeče (lokálně pro uživatele), na své servery nebo používat externí úložiště uživatele, například Google Drive¹. Čím více různých typů úložiště nástroj podporuje, tím lépe, neboť uživatel může flexibilně zvolit jeho účelům vyhovující způsob uchovávání dat. Pro interaktivní spolupráci s týmem je lepší sdílené úložiště a pro lokální práci je vhodnější lokální úložiště.

Zmíněná interaktivní spolupráce je dalším důležitým kritériem. U velkých projektů je modelování schématu urychleno, pokud nástroj spolupráci umožňuje.

Dále budeme porovnávat formát, do kterého nástroj diagram ukládá (pokud k uloženému souboru má uživatel přístup). Může se jednat o serializovaný dokument do

¹<https://www.google.com/drive/>

dobře známého standardního formátu, nebo o vlastní formát, který je často nakonec také založený na nějakém standardu.

Kromě uložení rozpracovaného projektu do vhodného formátu musí nástroj umožnit export do formátu, který uživatelé využijí pro své účely. Formáty pro export lze rozdělit do několika kategorií:

- serializovaný formát – většinou se jedná o vlastní formát aplikace a takový soubor nelze jinou aplikací otevřít, ale lze jej programově zpracovat,
- rastrové formáty, např. Portable Network Graphics (PNG)² – mají nejširší využití a podporu, lze je použít v dokumentech a na webových stránkách,
- vektorové formáty, např. Scalable Vector Graphics (SVG) [10] – nemají tak rozšířenou podporu, nicméně jsou vhodnější v dokumentech po estetické stránce díky libovolnému rozlišení (zvláště při tisku); dále existují vektorové editory, pomocí nichž lze výsledek libovolně upravovat bez potřeby souboru v serializovaném formátu; většina webových prohlížečů formát SVG podporuje a soubor vykreslí; do této kategorie lze zařadit i jiné otevřené strukturované formáty, např. VSDX³,
- zjednodušený export – některé nástroje šetří práci uživatele tím, že diagram rovnou exportují do HTML⁴, PDF⁵ a podobných finálních formátů pro okamžitou aplikaci, přestože uživatel může zvolit jiný formát a finální vytvořit sám,
- schematické formáty, např. SQL⁶ – téměř výhradně u nástrojů logické vrstvy; umožňují rovnou vytvářet schémata pro databáze.

Stejně jako u typu úložiště, čím více různých formátů exportu nástroj podporuje, tím lépe, neboť nástroj je flexibilní.

Posledním, neméně důležitým kritériem, je způsob komercializace. Většina volně dostupných nástrojů je nějakým způsobem zpoplatněna, ať už se jedná o jednorázový nebo pravidelný poplatek. Nejčastějším komerčním modelem je verze zdarma s omezenými funkcemi a dále několik placených plánů různé úrovně s odemčenými pokročilými funkcemi. U tohoto modelu je důležité vyrovnat funkce tak, aby byl nástroj použitelný i v bezplatné verzi, a aby byly placené funkce atraktivní pro uživatele. Při srovnávání budeme věnovat pozornost i tomu, jestli jsou placené funkce esenciální.

1.2 diagrams.net

Nástroj diagrams.net [5], dříve draw.io, je obecný open-source kreslicí nástroj (který však nepřijímá změny od externích vývojářů [11]) vydaný s licencí Apache License 2.0⁷, dostupný jako webová aplikace⁸ nebo jako desktopová aplikace. Desktopová verze aplikace je sestavena stejným způsobem jako webová, pouze je zabalená pomocí platformy Electron [12] do okna Chromium. Je vyvinut v běžných webo-

²Portable Network Graphics – <https://www.w3.org/TR/2003/REC-PNG-20031110/>

³Microsoft Visio XML formát založený na ISO 29500 – <https://interoperability.blob.core.windows.net/files/MS-VSDX/%5bMS-VSDX%5d.pdf>

⁴HyperText Markup Language – <https://w3.org/TR/2021/SPSD-html52-20210128/>

⁵Portable Document Format, ISO 32000 – <https://iso.org/standard/75839.html>

⁶Structured Query Language – <https://iso.org/standard/63555.html>

⁷<https://www.apache.org/licenses/LICENSE-2.0>

⁸na adrese <https://app.diagrams.net>

vých technologiích (JavaScript⁹, CSS¹⁰, HTML). Srovnávací kritéria jsou vyhodnocena v Tabulce 1.1.

kategorie	kresba libovolných diagramů
úložiště	lokální, externí, prohlížeč
export	serializovaný, rastrový, vektorový, zjednodušený
spolupráce	částečně podporována (pomocí externích úložišť)
komercializace	veškeré funkce jsou zdarma a není potřeba uživatelský účet; z jiného pohledu lze počítat cenu externích úložišť, ale ta jsou volitelná

Tabulka 1.1: Vyhodnocení srovnávacích kritérií pro nástroj diagrams.net

Uživatelé jsou v levém postranním panelu k dispozici standardní tvary ER diagramů, UML diagramů [2], flowchart diagramů a další základní tvary pro kresbu diagramů. Tvary lze libovolně kombinovat a spojovat podržením levého tlačítka a tažením myši z a do kotev na krajích objektů. Každý objekt a spojovací čára má vlastnosti, které lze upravovat v pravém postranním panelu. Upravovat lze přímo i vlastnosti formátu SVG.

Uživatelské rozhraní, které je vidět na Obrázku 1.1, je velmi podobné kancelářským aplikacím Google. Je tak přívětivé pro nové uživatele, kteří již s aplikacemi Google dříve pracovali.

Menu **Arrange** **Layout** umožňuje celý diagram uspořádat do zvoleného rozložení (Horizontal Flow, Vertical Flow, Horizontal Tree, Vertical Tree, Radial Tree, Organic, Circle, Org Chart, Parallels). Případně lze zvolit **Apply...**, kde lze aplikovat libovolnou transformaci rozložení¹¹. Tato funkce však není perfektní, protože po změně rozložení se jednotlivé prvky diagramu překrývají a uživatel je musí přesunout manuálně do vhodné pozice. Přenastavení rozložení však alespoň položí prvky do zvolené obecné pozice.

V menu **Extras** **Mathematical Typesetting** lze povolit vykreslování matematické notace pomocí knihovny MathJax [13]. Pokud pak v nějakém textovém objektu uživatel zadá například (x^2) , je vykresleno x^2 . Správné vykreslení matematické notace je pak zachováno ve všech zmíněných exportovaných formátech.

Diagramy lze uložit do serializovaného XML¹² formátu .drawio. V tomto formátu je pro každý diagram XML element diagram, ve kterém se nachází data zakódovaná do Base64¹³. Tato data jsou komprimována pomocí zlib¹⁴ a obsahují další XML dokument (URL-encoded¹⁵, tj. zakódovaný), tentokrát již serializaci vlastního diagramu [14]. Formát tak není bez dekomprese čitelný člověkem. Výhodou je, že lze uložit více diagramů do jednoho souboru a každý pojmenovat. Rozhraní k tomu určené je identické s listy souboru tabulkových procesorů, jako Microsoft Excel¹⁶ a Google She-

⁹Standardizován jako ECMA Script, ISO 16262 – <https://iso.org/standard/55755.html>

¹⁰Cascading Style Sheets – <https://www.w3.org/TR/css>

¹¹Dokumentace dostupných transformací rozložení <https://jgraph.github.io/mxgraph/docs/js-api/files/layout/hierarchical/mxHierarchicalLayout-js.html>

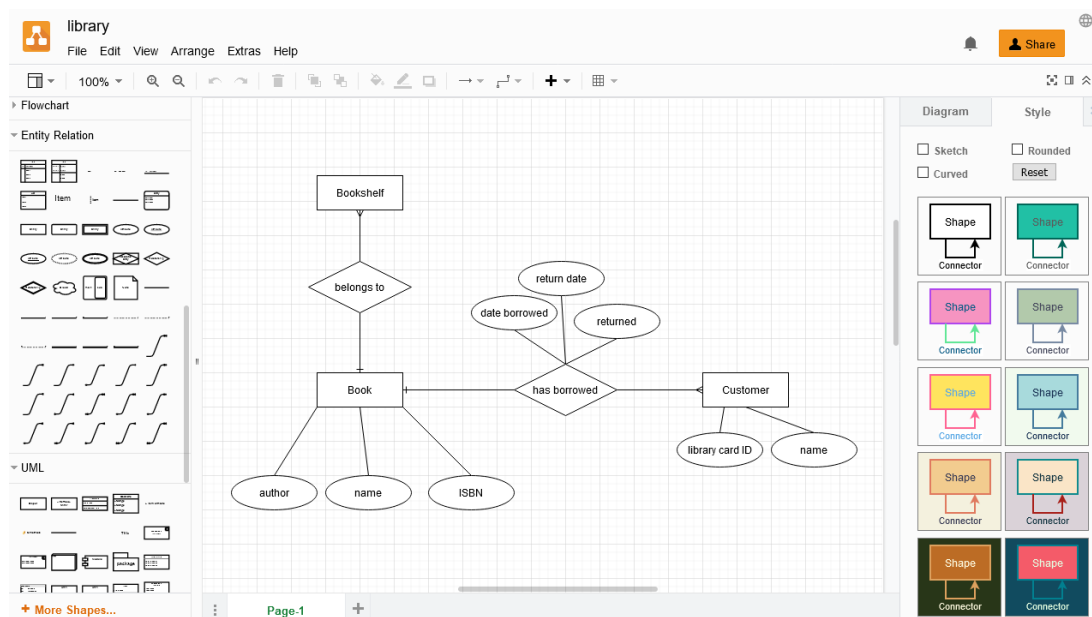
¹²Extensible Markup Language – <https://www.w3.org/TR/xml/>

¹³RFC 2045 §6.8 – <https://datatracker.ietf.org/doc/html/rfc2045#section-6.8>

¹⁴<https://zlib.net>

¹⁵RFC 3986 §2.1 – <https://datatracker.ietf.org/doc/html/rfc3986#section-2.1>

¹⁶<https://aka.ms/excel>



Obrázek 1.1: Tvorba ER diagramu v aplikaci diagrams.net

ets¹⁷.

Jednosouborový program v jazyce Python (viz kód 1.1) přijímá na standardním vstupu base64 řetězec formátu mxfile a na standardní výstup vypíše výslednou dekodovanou XML serializaci. Použití algoritmu Inflate na syrová data je vynuceno podle dokumentace zlib¹⁸. Jedná se o ukázkou postupu dekódování a toho, že formát je otevřený. Organizace JGraph dokonce poskytuje online nástroj¹⁹, pomocí kterého lze dosáhnout téhož výsledku.

Soubor s diagramy lze také uložit do formátu SVG, který je navíc otevřený a podporují ho jiné nástroje. Uživatel má při exportu k dispozici možnost *Include a copy of my diagram*, která do SVG souboru zahrne již zmíněný Base64 řetězec, ve kterém je diagram serializovaný. Ve výsledku to znamená, že takto exportované SVG soubory umí diagrams.net i otevřít a práce na nich může plnohodnotně pokračovat. Toto řešení se nám líbí, protože se jedná o schování vlastního formátu do SVG, který je nejvhodnějším pro přechovávání a zobrazování diagramů.

Dalšími možnostmi exportu a ukládání jsou

- rastrové soubory PNG, JPEG²⁰,
- soubor PDF, do kterého je ve vektorovém formátu diagram vložen,
- soubor HTML, do kterého lze podobně jako v SVG data diagramu uložit v serializované formě, případně pouze vložit veřejný odkaz URL na diagram (pokud je použito odpovídající úložiště); v tomto souboru je pak zahrnut JavaScript od diagrams.net, který diagram vykreslí,
- otevřený formát VSDX, původně vyvinutý pro Microsoft Visio.

Zajímavou vlastností exportu do rastrového formátu PNG je, že po otevření v pro-

¹⁷<https://sheets.google.com>

¹⁸<https://www.zlib.net/manual.html>

¹⁹<https://jgraph.github.io/drawio-tools/tools/convert.html>

²⁰Joint Photographic Experts Group, ISO 19566 – <https://iso.org/standard/65348.html>

```

1 from sys import stdin
2 from base64 import b64decode
3 from zlib import decompress, MAX_WBITS
4 from urllib.parse import unquote
5
6 base64_data = stdin.read()
7 deflated_data = b64decode(base64_data, validate=False)
8 # wbits must be -MAX_WBITS which makes zlib use the raw Inflate algorithm
   without header detection
9 urlencoded_inflated_data = decompress(deflated_data, -MAX_WBITS)
10 urldecoded_data = unquote(urlencoded_inflated_data)
11 print(urldecoded_data)

```

Kód 1.1: Dekódování mxfile

gramu diagrams.net je diagram plnohodnotně upravovatelný. Je toho dosaženo tím, že v tEXt chunku²¹ je pod klíčovým slovem mxfile zahrnuta plnohodnotná serializace diagramu.

Ze stejných souborů lze diagramy také importovat, ovšem editovat je lze, jen pokud je v nich zahrnut formát drawio, čehož je dosaženo u některých formátů popsaných výše.

Jako úložiště si lze vybrat Google Drive, OneDrive²², Dropbox²³, GitHub²⁴, GitLab²⁵, paměť prohlížeče a místní úložiště (disk uživatele). Soubor lze ze stejných úložišť i otevřít a importovat, navíc k tomu i z libovolné dostupné URL.

Interaktivní spolupráce je umožněna, pouze pokud soubor jako úložiště využívá takové, ke kterému mají přístup zápisu (popř. pouze čtení) všichni účastníci se uživatelé (Google Drive, OneDrive, Dropbox, GitHub, GitLab). Tato úložiště je však nutno manuálně vhodně nastavit (přístup ostatním uživatelům). U všech úložišť je rychlost reflektování změn ostatních uživatelů podobná – vcelku pomalá, protože aplikace musí změny aktivně kontrolovat a načítat.

Menu **File** » **Publish** chybně napovídá, že se jedná o funkci interaktivní spolupráce. Ve skutečnosti je uživateli jen zobrazen odkaz na soubor ve vybraném úložišti (ale pouze pro Google Drive a OneDrive, jinak je tato možnost vypnuta). Spolupracující uživatel tak musí tento soubor v daném úložišti uložit k sobě (sdílení), aby mohla spolupráce začít.

Jako další možnost jsme zvažovali desktopovou aplikaci s načteným souborem, který je libovolným externím nástrojem sdílen mezi uživateli. Soubor se nepřenačítá automaticky, ale musí být manuálně synchronizován tlačítkem **File** » **Synchronize**, které je dostupné pouze v desktopové verzi aplikace. Uživatel je při externí změně souboru upozorněn (avšak ne spolehlivě vždy) červeným nápisem. Algoritmus synchronizace funguje správně a tak, jak uživatel očekává.

Nejllepší způsob dosažení interaktivní spolupráce je dle našeho názoru volba systému pro správu Git²⁶ repozitářů (GitLab nebo GitHub), protože

²¹dle PNG formátu sekce 11.3.4.3 <https://www.w3.org/TR/2003/REC-PNG-20031110/>

²²<https://aka.ms/onedrive>

²³<https://dropbox.com>

²⁴<https://github.com>

²⁵<https://gitlab.com>

²⁶Systém pro správu verzí Git – <https://git-scm.com>

1. tato úložiště jsou dostupná jak z webové, tak z desktopové verze aplikace,
2. synchronizace probíhá pomocí systému Git,
3. díky použití systému Git lze jednoduše spravovat verze a body v historii při vývoji diagramu.

K poslednímu bodu je třeba podotknout, že jiná webová úložiště také podporují správu verzí, avšak není tak rozvinutá, jako správa systémem k tomu určeným – Git. Diagrams.net sám o sobě správu verzí neobsahuje, jen obvyklé „Undo, Redo“ pro aktuálního uživatele. Úpravy ostatních uživatelů nelze vrátet postupně, lze se pouze vrátit za bod synchronizace.

Jako výhody určujeme

- univerzálnost a flexibilita – nástroj lze použít pro tvorbu jakýchkoli diagramů,
- množství podporovaných formátů – export pokrývá téměř všechny možné účely,
- cena – všechny funkce jsou zdarma,
- více diagramů v jednom souboru

a nevýhodami jsou

- chybějící možnost pro export do (jednoduše) strojově zpracovatelného formátu, nelze tak bez lidské práce diagram převést do logické vrstvy (to je zapříčiněno obecností nástroje, jeho účelem je kresba, ne abstrakce),
- pomalé zobrazování změn při interaktivní spolupráci, zároveň není zpočátku jasné, jak spolupráce dosáhnout.

Výhodou i nevýhodou může být nutnost použití externího úložiště. Pro velké společnosti se může jednat o bezpečnostní opatření, protože diagrams.net k diagramům nemá přístup. Pro malé týmy se může jednat o nevýhodu, protože je potřeba účet na externím webu, nebo jiný způsob sdílení a správa tohoto úložiště.

1.3 drawSQL

Nástroj drawSQL [6] je modelovací nástroj pro tvorbu relačních schémat. Aplikace je dostupná ve webovém prohlížeči²⁷. Je vyvinuta ve standardních webových technologiích a používá framework Vue.js. Plán zdarma umožňuje tvorbu veřejně přístupných diagramů, které mohou mít maximálně 15 tabulek (entit). Měsíčně placené plány umožňují vytvářet neveřejné diagramy, více (až neomezeně mnoho) tabulek v diagramu, více uživatelů, kteří mohou na diagramu spolupracovat, a přístup k verzovacím nástrojům. K vyzkoušení i používání nástroje je potřeba uživatelský účet. Srovnávací kritéria jsou vyhodnocena v Tabulce 1.2.

Hlavní funkcí drawSQL je export schématu do SQL. Proto si uživatel při vytváření diagramu zvolí cílovou databázi, pro kterou schéma tvoří. Výsledný SQL výraz tak má zaručenou kompatibilitu s cílovou databází. Podporovanými databázemi jsou MySQL²⁸, PostgreSQL²⁹ a SQL Server³⁰.

²⁷na adrese <https://drawsql.app>

²⁸<https://mysql.com>

²⁹<https://postgresql.org>

³⁰Microsoft SQL Server – <https://aka.ms/sqlserver>

kategorie	logická vrstva
úložiště	online, poskytované autory produktu
export	schematický (obecný SQL i platformě specifické formáty), rastrový PNG, serializovaný (JSON [15], v době psaní práce se chystá)
spolupráce	pouze v placené verzi
komercializace	omezená verze navždy zdarma, různé měsíčně placené plány

Tabulka 1.2: Vyhodnocení srovnávacích kritérií pro nástroj drawSQL

Rozhraní, které je vidět na Obrázku 1.2, obsahuje diagram a postranní panel. V postranním panelu lze vytvářet jednotlivé tabulky, definovat jejich sloupce a vlastnosti jednotlivých sloupců – typ sloupce, nullability³¹, zda se jedná o primární klíč, unikátní klíč nebo index. Tyto změny se v reálném čase reflektují v diagramu, ve kterém může uživatel jednotlivé sloupce spojovat, čímž vytváří cizí klíče. Pozici těchto lomených čar lze upravovat pouze posunutím tabulky v diagramu. Pokud je cizích klíčů víc, začne být diagram velmi nepřehledný.

Diagram lze importovat ze souboru SQL stisknutím **File** » **Import**. Stisknutím tlačítka **File** » **Export** se otevře nabídka Export, ve které může uživatel diagram exportovat do SQL své předem zvolené databáze, nebo do rastrového obrázku ve formátu PNG. Vývojáři aplikace plánují implementovat také export diagramu pomocí serializace do formátu JSON. V nabídce Export je navíc možnost nechat si vygenerovat platformě specifický kód jako například migrační třídy pro Laravel³², definice modelů pro Laravel, a migrační schémata pro AdonisJS³³.

Interaktivní spolupráce je k dispozici pouze v placené verzi. Dle našeho názoru je interaktivní spolupráce hlavní funkcí tohoto nástroje oproti konkurenčním relačním modelovacím nástrojům. Některá integrovaná vývojová prostředí (např. Visual Studio³⁴) obsahují nástroj pro relační modelování i generování databázového schématu. Hlavním omezením těchto nástrojů je však absence interaktivní spolupráce, jedná se spíše o spolupráci iterací. Proto považujeme určení interaktivní spolupráce jako placené funkce za negativní rozhodnutí pro využitelnost nástroje v relaci s konkurencí.

Web drawSQL také zveřejňuje šablony modelů³⁵ (jedná se spíše o příklady). Šablony jsou většinou potenciální modely známých produktů (např. WordPress³⁶) a tvoří je autoři drawSQL. Tuto funkci považujeme za výhodu, protože společnosti a individuální vývojáři se mohou inspirovat existujícími a ověřenými řešeními, případně nezačínat se svým modelem od nuly.

Závěrem určíme výhody drawSQL:

- příjemné uživatelské rozhraní (viz Obrázek 1.2),
- možnost určení typu relace, o sémantiku se aplikace stará sama (one-to-one, one-to-many, many-to-many),

³¹ nullability je příznak, který určuje, zda lze sloupec v řádku nastavit na hodnotu NULL

³² Framework pro PHP – <https://laravel.com>

³³ Framework pro Node.js – <https://adonisjs.com>

³⁴ Vývojové prostředí Microsoft Visual Studio – <https://visualstudio.microsoft.com>

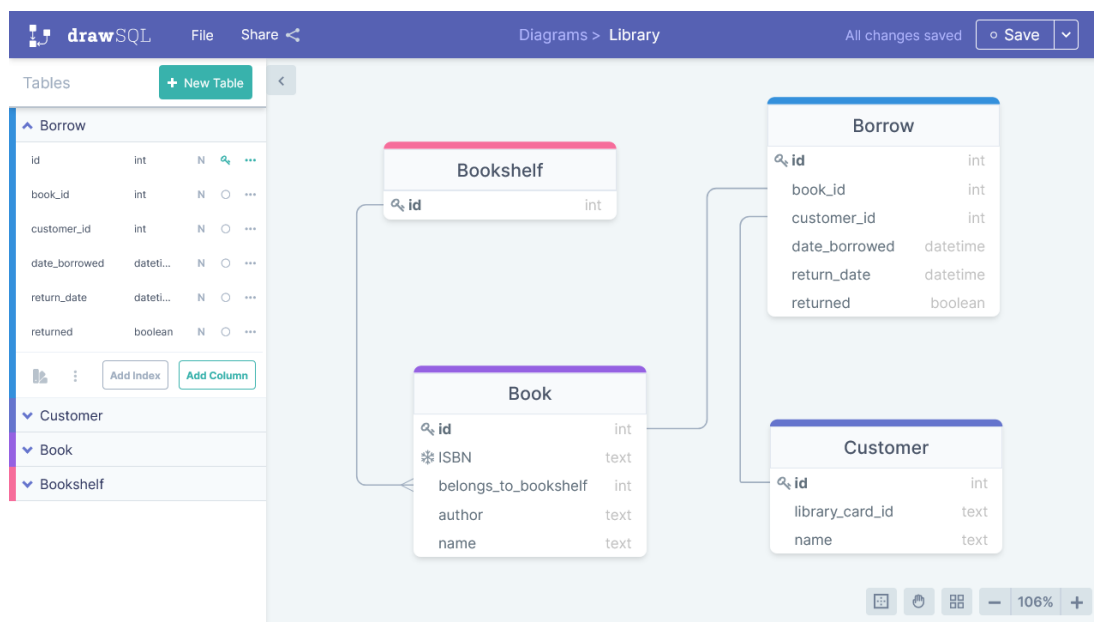
³⁵ na adrese <https://drawsql.app/templates>

³⁶ <https://wordpress.com>

- několik platformě specifických generátorů modelu,
- šablony a příklady existujících modelů

a nevýhody:

- nepřehlednost – nelze upravit ani přesunout lomené čáry spojující cizí klíče, což způsobuje chaos, pokud je v diagramu větší množství entit,
- interaktivní spolupráce pouze v placeném plánu,
- správa verzí pouze v placeném plánu,
- k vyzkoušení nástroje je potřeba uživatelský účet,
- podporuje pouze relační databáze.



Obrázek 1.2: Tvorba diagramu v drawSQL

1.4 ERDPlus

Nástroj ERDPlus [7] je modelovací nástroj pro tvorbu ER diagramů, relačních schém a hvězdicových schém. Aplikace je dostupná ve webovém prohlížeči³⁷. Její uživatelské rozhraní je tedy vyvinuto ve standardních webových technologiích – HTML, CSS a JavaScript. Dále využívá framework React [16] pro tvorbu rozhraní v jazyce JavaScript. Srovnávací kritéria jsou vyhodnocena v Tabulce 1.3

ERDPlus lze používat bez založení uživatelského účtu a vytvořený diagram exportovat do speciálního formátu erdplus, nicméně uživatel tak přijde o možnost využití úložiště diagramů na serveru aplikace. Diagramy (ERDPlus je nazývá *dokumenty*) lze organizovat do složek a podsložek. Služby ERDPlus včetně úložiště nejsou žádným způsobem zpoplatněny.

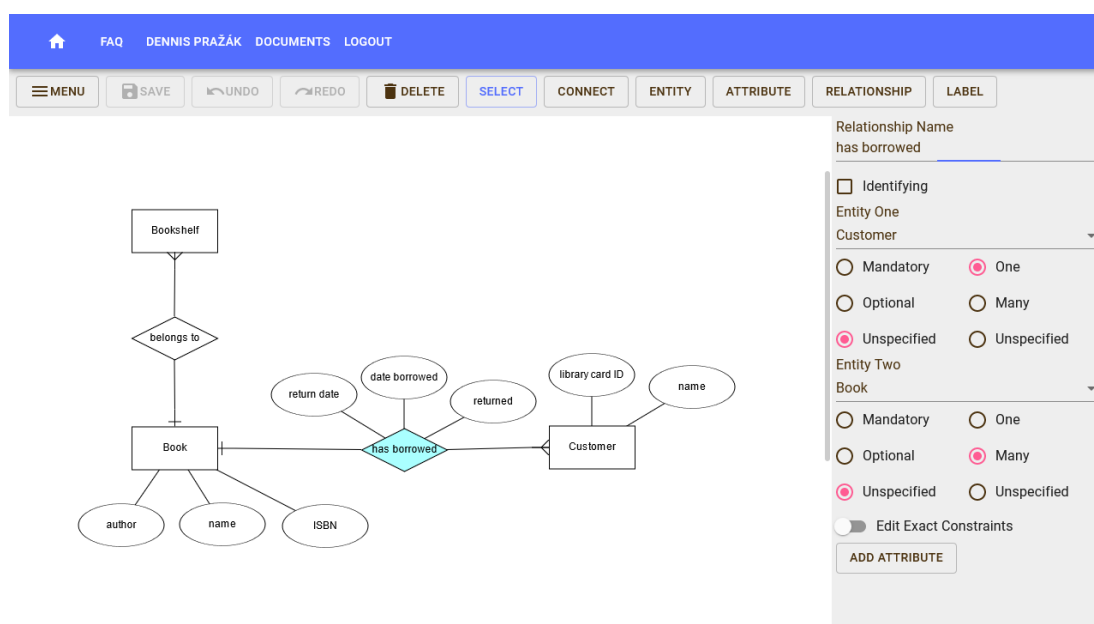
Tvorba ER diagramů je intuitivní s jednoduchým uživatelským rozhraním, které je vidět na Obrázku 1.3. Uživatel má na výběr mezi vytvořením entity, atributu, relace,

³⁷na adrese <https://erdplus.com>

kategorie	logická vrstva
úložiště	online, poskytované autory produkt
export	rastrový PNG
spolupráce	není
komercializace	zdarma

Tabulka 1.3: Vyhodnocení srovnávacích kritérií pro nástroj ERDPlus

spojení mezi těmito objekty a jednoduchého textového popisku. V pravé části rozhraní se nachází panel s vlastnostmi zvoleného objektu. V tomto panelu může uživatel také rychleji tvořit atributy entit a relací. Při zvolení relace lze v panelu zvolit entity, které mají být v relaci, a spojení je pak automaticky vytvořeno. Zároveň lze zvolit jednotlivé kardinality relace.



Obrázek 1.3: Tvorba ER diagramu v ERDplus

Co nám v tomto nástroji rozhodně chybí, jsou složené identifikátory entitních typů. Nenašli jsme způsob, jak je v ERDPlus nastavit. Jednotlivé atributy lze označit za „unikátní“ a nejspíš mají být považované za jednoduché identifikátory. Entitní typy lze označit za „slabé“ a vztahové typy lze určit za „identifikující“. Nelze však nijak určit, kterými kombinacemi vlastních a případně externích atributů lze takový entitní typ identifikovat.

Nástroj nepodporuje ani složené atributy (atributy, které mají své atributy). Jelikož ER není standardizováno, nelze hodnotit dokončenost ER modelu. Vyjadřovací schopnost nástroje hodnotíme jako velmi základní.

Soubor s diagramem je v úložišti reprezentován vlastním formátem erdplus. Jedná se o textový soubor, jehož obsahem je JSON reprezentace diagramu. Diagram lze exportovat do rastrového formátu PNG.

Zajímavou funkcí je také převod do relačního schématu. Tato funkce je dostupná pouze tehdy, když uživatel ER diagram uloží na server ERDPlus. Poté zvolí možnost *Convert to Relational Schema* a ERDPlus vytvoří nové relační schéma. Z relačních

schémat lze podobně vygenerovat SQL.

Vlastnoruční tvorba relačních diagramů probíhá podobně. Uživatel může tvořit tabulky, přidávat jim sloupce a v tabulkách volit primární klíče v postranním panelu. Pomocí tlačítka `Connect` lze poté přidat cizí klíč, který odkazuje do jiné tabulky tažením myši. ERDPlus do tabulky přidá všechny primární klíče, které cílová tabulka obsahuje, jako nové sloupce. K vytvoření cizího klíče, který odkazuje na stejnou tabulku (tzv. rekurzivní klíč) slouží tlačítko `Recursive Key` ve vlastnostech tabulky. Hvězdicovým schémátům se věnovat nebudeme, protože jsou mimo rozsah této práce.

Výhody:

- převod diagramu z ER do relačního diagramu,
- jednoduchost a intuitivnost procesu kresby diagramu

a nevýhody:

- relační diagram bývá nepřehledný, nelze měnit pořadí jednotlivých definovaných sloupců v tabulce,
- chybí export do vektorového obrázku,
- diagramy nelze stylizovat.

1.5 nomnoml

Nástroj nomnoml je modelovací nástroj pro tvorbu UML diagramů dostupný ve webovém prohlížeči³⁸. Jeho klíčová vlastnost je, že místo interakce myši s webovou aplikací probíhá kresba deklarativně – psaním. Srovnávací kritéria jsou vyhodnocena v Tabulce 1.4.

kategorie	kresba omezených diagramů – UML
úložiště	online
export	rastrový PNG, vektorový SVG
spolupráce	není k dispozici
komercializace	zdarma s otevřeným zdrojovým kódem

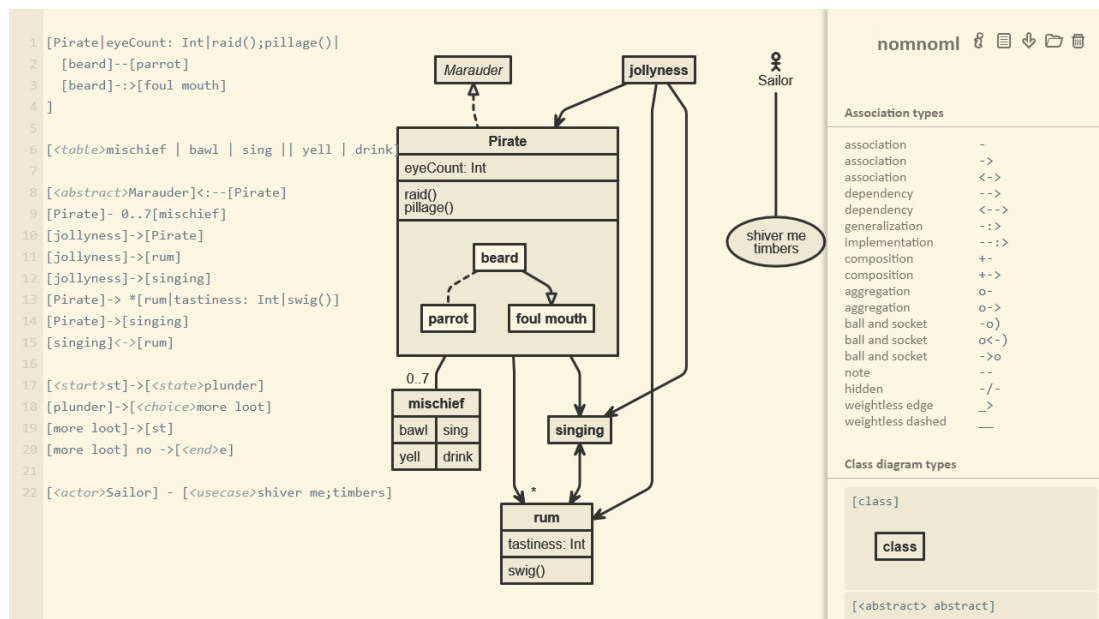
Tabulka 1.4: Vyhodnocení srovnávacích kritérií pro nástroj nomnoml

Uživatelské rozhraní (viz Obrázek 1.4) se skládá z oblasti pro textový vstup, nad kterou je zároveň (v reálném čase) vykreslován výsledný diagram. V pravé horní části se nachází několik tlačítek, při kliknutí na některé z nich se vždy otevře pravý postranní panel s odpovídajícími informacemi a funkcemi. První tlačítko ukazuje rychlý přehled jazyka, ve kterém se má diagram definovat. Druhé tlačítko odhalí kompletní referenci k tomuto jazyku. Dále lze najít tlačítka pro export, sdílení a uložení do místního úložiště uživatele.

Jazyk diagramů je velmi jednoduchý. Skládá se z definic entit a jejich relací. Uživatel může vyjít z úvodního diagramu, který se zobrazí při navštívení hlavní stránky nástroje. Pro ukázkou, definice entity vypadá následovně

```
[<abstract> Entita|soukromaSlozka; soukromaSlozka2|verejnyAtribut].
```

³⁸na adrese <https://nomnoml.com>



Obrázek 1.4: Tvorba UML diagramu v nomnoml

Svislá čára odděluje kategorie atributů, může jich být neomezené množství. Entita je definována jako abstraktní, což také ovlivní její výsledný styl vzhledu v diagramu.

Dále se v jazyce definují vztahy mezi entitami [Entita] -> [Entita2]. Různé šipky mají různé významy. Vztah -> je *asociace*, dále o-> je *agregace* apod.

V jazyce lze také deklarovat direktivy, začínající znakem #. Těmi může uživatel upravit vzhled, vytvořit nové styly a nastavit algoritmy, kterými bude zvoleno rozložení entit v diagramu. Algoritmy lze nastavit direktivou #ranker a na výběr je ze tří možností: *network-simplex*, *tight-tree*, *longest-path*. Nástroj nomnoml používá k vykreslování diagramu knihovnu *dagre*³⁹ pro JavaScript, jejíž vývoj byl však ukončen. Z dokumentace této knihovny vyplývá, že možnost *ranker* mění algoritmus, který vrcholům v grafu (diagramu) přiřazuje důležitost, která se pak odráží v pořadí zobrazení entit. Definitivní význam této možnosti není z dokumentace zřejmý, u nástroje nomnoml lze však pozorovat změnu v pořadí a vzdálenostech entit (délce spojovacích čar). Uživatel tak může vyzkoušet různá rozložení a použít to, které je vizuálně nejpřehlednější.

Diagram lze exportovat do formátu PNG a dále podobně jako v Sekci 1.2, nomnoml také umožňuje export do SVG se zakomponovaným zdrojovým kódem. Uživatel tak může diagram distribuovat v tomto formátu a zároveň tento formát v nástroji nomnoml i otevřít a plnohodnotně pokračovat v práci. Podobně je možné sdílet odkaz přímo na vytvořený diagram ve službě nomnoml. Odkaz je vytvořen tak, že do URL je jako parametr *source* zapsán přímo zdrojový kód diagramu. Výsledná adresa URL je tak velice dlouhá, ale služba nomnoml nemusí diagramy ukládat, stačí je vykreslit z dat v odkazu. Přestože se nejedná o opravdové online úložiště, kategorizovali jsme ho tímto způsobem. Rozdělaný diagram lze také uložit do paměti prohlížeče. Díky těmto mechanismům stačí službě nomnoml staticky poskytovat pouze klientskou stranu své aplikace, a přesto umožňuje ukládání a sdílení práce.

Nástroj nomnoml je tedy inovativní svým přístupem ke kresbě diagramů. U slo-

³⁹<https://github.com/dagrejs/dagre>

žitých diagramů se však nutně ve zdrojovém kódu uživatel ztrácí, protože nomnoml nenabízí žádné dělení či kompozici tohoto kódu. S vizuální reprezentací grafu nelze přímo (např. myší) pracovat, veškeré rozložení a orientaci diagramu tak musí uživatel nechat na algoritmu nástroje. Nástroj dále nenabízí ani možnost pracovat s několika diagramy najednou. Z těchto důvodů určujeme produkt jako vhodný pouze pro jednotlivce a tvorbu méně rozsáhlých UML diagramů.

1.6 Visual Paradigm Online

Nástroj Visual Paradigm Online⁴⁰ je proprietární produkt od firmy Visual Paradigm. Tato firma je známá svou stejnojmennou desktopovou aplikací, která má stejný účel. Zde se zaměříme na její online verzi. Srovnávací kritéria jsou vyhodnocena v Tabulce 1.5.

kategorie	kresba libovolných diagramů
úložiště	online, lokální, externí, prohlížeč
export	serializovaný, rastrový, vektorový, zjednodušený
spolupráce	podporována
komercializace	verze zdarma, měsíčně/ročně placené plány několika úrovní

Tabulka 1.5: Vyhodnocení srovnávacích kritérií pro nástroj Visual Paradigm Online

Visual Paradigm Online je nástroj pro tvorbu různých typů diagramů. Mimo těch je určen i pro úpravu fotografií, tvorbu koláží, design infografiky, příspěvků na sociální sítě, uživatelských rozhraní, plakátů, dárkových poukazů apod. Funkcionalita kromě tvorby diagramů je mimo rozsah této práce.

Nástroj poskytuje předlohy tvarů pro diagramy tříd, use case diagramy, sekvenční diagramy, diagramy aktivit, diagramy nasazení (deployment), ER diagramy a velmi mnoho dalších. Uživatel si před tvorbou diagramu musí vybrat jednu z těchto kategorií, čímž vymezí tvary, které jsou v uživatelském rozhraní dostupné k použití v levém postranním panelu. Ve stejném panelu lze ale posléze přidat libovolnou další kategorii tvarů dle výběru uživatele.

Nástroj je svým rozhraním (pozorovat ho lze Obrázku 1.5), designem a chováním nápadně podobný produktu diagrams.net ze Sekce 1.2. Stejně tak spojování tvarů čarami má velmi podobné chování. Jeden rozdíl je ten, že kresba spojení z jednoho tvaru do toho samého (tvorba „cyklu“) je velmi nepředvídatelná. Čára se při tom samovolně přemisťuje do všech možných stran, až nakonec často zůstane na jedné ze stran tvaru.

Diagramy lze uložit do formátu vpd, který při otevření v textovém editoru připomíná base64 kód. Vypadá velice podobně jako serializace pro diagrams.net ze Sekce 1.2. Nejedná se však o validní base64 kód, protože ten kóduje 3 bajty do 4 znaků, jinak používá výplň 0-2 znaky =. Nicméně vpd obsahuje i sekvence jako je M=Q8, které tomu neodpovídají. Jedná se tak nejspíš o nezdokumentovaný proprietární formát.

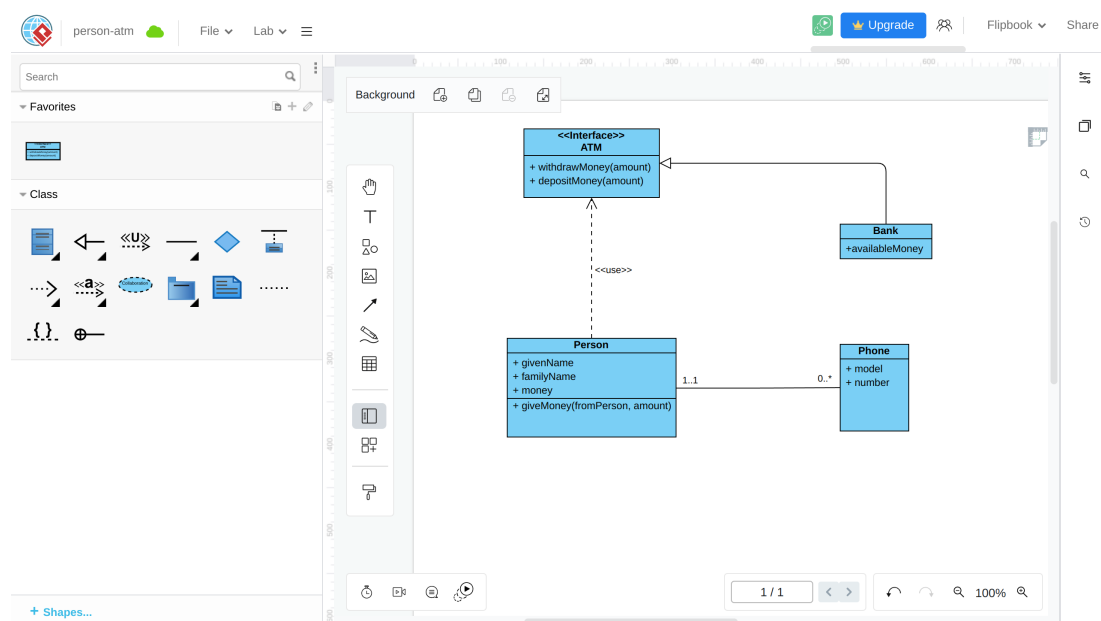
Diagramy lze exportovat do formátu SVG a PDF, do kterých lze volitelně uložit i serializaci diagramu ve formátu vpd a případně tak plnohodnotně přenést editovatelný diagram i s jeho vektorovou reprezentací. Rastrový export je možný ve formátu

⁴⁰<https://online.visual-paradigm.com>

JPEG nebo PNG. Stejně jako u diagrams.net ze Sekce 1.2 je do PNG volitelně přidána serializace diagramu v `text` chunku formátu PNG. Dokonce je použito stejné klíčové slovo identifikující tuto serializaci – `mxfile`. To vzbuzuje podezření, že se jedná o stejný formát, jako používá diagrams.net ze Sekce 1.2, tedy formát serializace modelu diagramu knihovny `mxGraph`⁴¹. Tuto domněnku potvrdila i podpora diagrams.net v odpovědi na náš dotaz. Konkrétně, Visual Paradigm Online postavili celý svůj editor na open-source editoru diagrams.net.

Jak už bylo zmíněno, soubor `vpd` nemá úplně stejnou strukturu jako `mxfile`⁴². Nejspíše se jedná o nějaké proprietární zakódování `mxfile` souboru, aby šel otevřít pouze v closed-source editoru Visual Paradigm Online.

Komericializace Visual Paradigm Online je rozdělena do několika platebních úrovní. Každá úroveň má určenou cenu za uživatele za měsíc. Úrovně jsou dle ceny a úplnosti funkcí vzestupně Free (zdarma), Starter, Advance, Combo (všechny funkce). Úroveň Free obsahuje i prémiové šablony, ale je v nich zobrazen vodoznak. Ten je v placených plánech odstraněn. Placené plány navíc obsahují více typů diagramů a více tvarů k použití, rastrový export vyššího rozlišení, historii verzí diagramu (při spolupráci) a další.



Obrázek 1.5: Tvorba UML diagramu ve Visual Paradigm Online

1.7 Závěr existujících řešení

V této kapitole jsme zanalyzovali některé existující nástroje na tvorbu diagramů. Zjistili jsme, že jich existuje více druhů a každý je vhodný pro jiný účel, ať už jimi jsou konceptuální či logická vrstva, spolupráce v týmu, nebo dokonce vývoj schématu pro relační databáze s exportem přímo do SQL.

Přehled analýzy zmíněných existujících řešení je vidět v Tabulce 1.6. Tabulka pro

⁴¹<https://jgraph.github.io/mxgraph/>

⁴²<https://drawio-app.com/extracting-the-xml-from-mxfiles/>

každý nástroj uvádí evaluaci jeho hlavních srovnávacích kritérií a u některých z nich uvádí dodatečné vysvětlivky.

název	kategorie	úložiště	interaktivní spolupráce	formát projektu	export	komercializace
draw.io	libovolné	lokální, externí, prohlížeč	částečně ^a	mxfile	serializovaný, rastrový, vektorový, zjednodušený	zdarma ^{b,c}
drawSQL	logická	online	placená	JSON ^d	schematický, rastrový	zdarma, předplacené plány ⁱ
ERDPlus	konceptuální	online	ne	JSON	rastrový	zdarma
nomnoml	omezené	ne ^e	ne	nomnoml ^f	rastrový, vektorový	zdarma ^c
Visual Paradigm Online	libovolné	online, lokální, externí, prohlížeč	ano ^g	vpd ^h	serializovaný, rastrový, vektorový, zjednodušený	zdarma, předplacené plány ^j

- a. využívá úložiště třetích stran
- b. lze počítat navíc cenu externích úložišť
- c. open-source projekt
- d. plánovaná funkce v době psaní práce
- e. diagram může být ale součástí URL, pomocí které ho lze sdílet
- f. jedná se o vlastní jazyk pro tvorbu diagramů
- g. 3 účastníci interaktivní spolupráce zdarma, jinak placené
- h. nejspíše se jedná o zašifrovaný mxfile
- i. zdarma 1 uživatel a 15 projektů, plány 19-179 \$/měsíc
- j. zdarma základní diagramy, plány 4-15 \$/uživatel/měsíc

Tabulka 1.6: Srovnání existujících řešení

Porovnání exportovaných formátů, které mají nástroje k dispozici, jsou vidět v Tabulce 1.7. Formáty XML a JSON jsme vyhodnotili jako splněné, pokud soubor jejich projektu tento formát používá pro serializaci, případně pokud nástroj přímo nabízí export do daného formátu.

Srovnání podporovaných úložišť je uvedeno v Tabulce 1.8. Pouze nástroj draw.io podporuje více externích úložišť. Visual Studio Paradigm podporuje pouze Google Drive. Webová aplikace nomnoml přímo nepodporuje externí úložiště, ale díky poskytnutému lokálnímu nástroji si uživatel může nastavit pro tento účel všechna daná úložiště sám.

název	PNG	JPEG	SVG	PDF	HTML	SQL	XML	JSON
draw.io	ano	ano	ano	ano	ano	ne	ano	ne
drawSQL	ano	ne	ne	ne	ne	ano	ne	ne ^a
ERDPlus	ano	ne	ne	ne	ne	ano ^b	ne	ano ^c
nomnoml	ano	ne	ano	ne	ne	ne	ne	ne
Visual Paradigm Online	ano	ano	ano	ano	ne	ne	ne	ne

- a. plánovaná funkce v době psaní práce
b. převod ER → relační schéma → SQL
c. soubor má příponu erdplus, ale jedná se o textový soubor s obsahem ve formátu JSON

Tabulka 1.7: Exportované formáty existujících řešení

název	online	paměť prohlížeče	Google Drive	OneDrive	GitHub	GitLab	Dropbox
draw.io	ne	ano	ano	ano	ano	ano	ano
drawSQL	ano	ne	ne	ne	ne	ne	ne
ERDPlus	ano	ne	ne	ne	ne	ne	ne
nomnoml	ne	ano	ne ^a	ne ^a	ne ^a	ne ^a	ne ^a
Visual Paradigm Online	ano	ano	ano	ne	ne	ne	ne

- a. nomnoml kromě webové aplikace poskytuje i lokální program pro převod diagramů z nomnoml jazyka, takže lze nomnoml soubor uložit do těchto úložišť a spravovat jeho verze

Tabulka 1.8: Úložiště existujících řešení

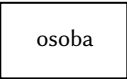


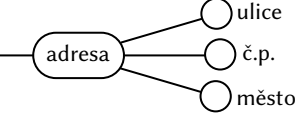
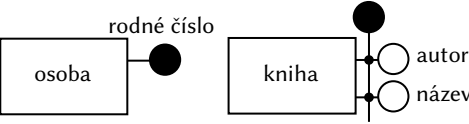
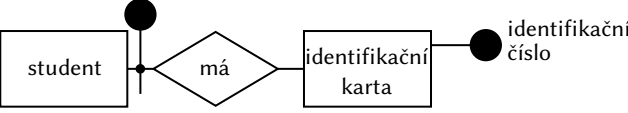
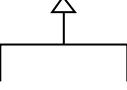
2. Teoretický rámec

V této kapitole představíme potřebné teoretické koncepty, které bude využívat výsledná aplikace. Mezi tyto koncepty patří Entity-Relationship (ER) model a nově vyvíjený konceptuální model v podobě schematické kategorie [3] v rozšířené verzi navržené vedoucím práce.

2.1 Entity-Relationship

Datový model Entity-Relationship (ER) poprvé představil Peter Pin-Shan Chen už v roce 1976 [1]. Od té doby se však ER vyvíjel, jak se potřeby datového modelování rozšiřovaly. ER není standardizováno, ale jednu moderní verzi představili Atzeni, Ceri, Paraboschi a Torlone [17, s. 163-179]. Na jejich ER modelu založíme ten náš, který zde popíšeme.

V Tabulce 2.1 jsou vyobrazeny jednotlivé konstrukty ER modelu.

Konstrukt	Vizuální reprezentace
Entitní typ	
Vztahový typ	
Atribut	
Složený atribut	
Interní identifikátor	
Externí identifikátor	
Zobecnění	

Tabulka 2.1: Grafická reprezentace konstruktů ER modelu, upraveno a přeloženo [17, s. 164, Obr. 5.4]

Zde blíže popíšeme jejich sémantiku:

- Entitní typ (Entity Type) reprezentuje předpis pro instance entit reálného světa. Každý entitní typ má jméno, které je unikátní v daném schématu.
- Vztahový typ (Relationship Type) reprezentuje vztah mezi dvěma nebo více (ne nutně různými) entitními typy. Každý vztahový typ má jméno.

- Atribut (Attribute) reprezentuje vlastnost entitních nebo vztahových typů. Každý atribut má jednoznačné jméno.
- Složený atribut (Composite Attribute) je atribut, který má sám atributy. Zakazujeme však další větvení, tedy atributy složeného atributu už samy nemohou být složené. Každý složený atribut má sám jméno, podobně jako jeho vlastní atributy.
- Kardinalita (Cardinality) je dvojice $(a, b) \in \{0, 1\} \times \{1, *\}$, kde a nazýváme minimální kardinalita (spodní hranice) a b maximální kardinalita (horní hranice). Kardinalitu musí mít každý atribut a každý účastník vztahového typu. Výchozí kardinalita je $(1, 1)$ a ve schématu se většinou neuvádí. Spodní hranice 0 znamená, že účast je volitelná; hranice 1 znamená, že účast je povinná. Horní hranice 1 znamená, že účast je nejvýše jedna; hranice $*$ znamená, že účastí je libovolný počet.
 - Hranice kardinalit pro jednotlivé účastníky vztahových typů vyjadřují minimální a resp. maximální počet výskytů jednotlivých instancí účastníků v tomto vztahu.
 - Hranice kardinalit u atributů vyjadřují minimální a resp. maximální počet hodnot atributu, které se vztahují k dané instanci entity/vztahu.
- Identifikátor (Identifier) umožňuje jednoznačně rozlišit (identifikovat) instance entitních typů. Pro každý entitní typ je povinný alespoň jeden identifikátor, ale může jich být více. Každý identifikátor je tvořen buď
 - jedním nebo více atributy daného entitního typu; takový identifikátor nazýváme *interní*, nebo
 - jedním, nebo více vztahovými typy, jichž se daný entitní typ účastní, a žádným či libovolným množstvím atributů daného entitního typu; takový identifikátor nazýváme *externí*.
- Zobecnění (generalization), nebo také ISA hierarchie¹ (ISA Hierarchy), vyjadřuje vztah podobný dědičnosti v objektově orientovaném programování. Jde o vztah mezi entitním typem E zvaným *rodič* a jedním nebo více *děťmi* E_1, \dots, E_n . Všechny vlastnosti rodiče (atributy, identifikátory, spojené vztahové typy a další ISA hierarchie) jsou i vlastnosti každého z dětí. Každá instance dítěte je také instancí rodiče.

Entitní typy, které nemají ani jeden interní identifikátor (musí mít tedy externí), nazýváme *slabé* entitní typy (weak entity types). Pokud mají interní identifikátor, nazýváme je *silné* entitní typy (strong entity types).

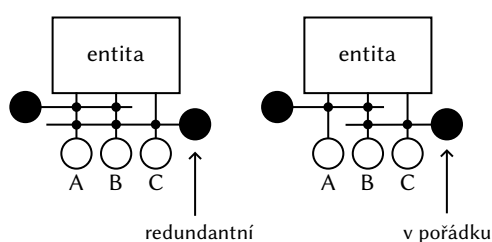
Vztahový typ se externí identifikace může účastnit nejvýše od jednoho účastníka. Jinak řečeno, pokud je vztahový typ součástí slabého identifikátoru nějakého entitního typu, daný vztahový typ už nemůže být součástí slabého identifikátoru jiného entitního typu. Jeden entitní typ však může mít více externích identifikátorů, který každý zahrnuje daný vztahový typ. Nicméně, externí identifikace se může řetězit, jako na Obrázku 2.2. Nesmí ovšem vzniknout orientovaný cyklus, a to ani v kombinaci s ISA hierarchiemi. Formálněji – pokud vytvoříme orientovaný graf $G = (V, E)$ takový, že

¹ISA z anglického „is a“, analogicky ke vztahu „has a“

- vrcholy V jsou entitní typy a
- hrany E jsou
 - pro každý externí identifikátor od identifikovaného entitního typu a k identifikujícímu entitnímu typu b orientovaná hrana (a, b) ,
 - pro každou ISA hierarchii pro každý vztah rodič-dítě, kde a je dítě a b je rodič, orientovaná hrana (a, b) ,

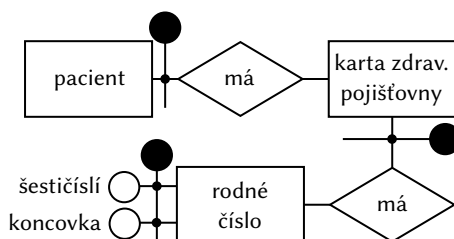
pak graf G musí být acyklický.

V ER se nesmí vyskytovat identifikátory, které jsou redundantní. Pro každé dva identifikátory jednoho entitního typu I, J , pokud $J \subset I$, pak I je *redundantní*, protože k identifikaci entitního typu stačí J . Všechny atributy a vztahové typy v $I \setminus J$ nejsou potřeba k identifikaci. Ukázka redundantního identifikátoru je na Obrázku 2.1.



Obrázek 2.1: Ukázka redundantního a neredundantního identifikátoru

Entitní typ, který má externí identifikátor, musí být účastněn vztahového typu (jímž je identifikován) s kardinalitou $(1, 1)$. Teoreticky by se mohl účastnit i s jinou kardinalitou, ale pro naše účely tyto situace modelovat nebudeme.



Obrázek 2.2: Zřetězení externích identifikátorů

U kardinality poznamenejme, že se v ER modelu často dovoluje použít jako hranice libovolná nezáporná celá čísla, tedy $(a, b) \in \mathbb{N}_0 \times (\mathbb{N}_0 \cup \{*\})$, tž. $a \leq b$ (dodefinujeme $\forall a \in \mathbb{N}_0 : a < *$). Dají se tak vyjádřit přesnější omezení, např. že jeden uživatel může mít maximálně 5 bankovních účtů. Ovšem námi definované hranice kardinality vyjadřují volitelnost/povinnost pro spodní hranici a jednočetnost/mnohočetnost pro horní hranici. Pokryjeme jimi z teoretického pohledu a s ohledem na povahu konstruktů v nejrůznějších logických modelech všechny strukturálně odlišné situace, které by mohly nastat.

Dále upozorníme, že místo $*$ se v ER modelu může použít symbol n nebo N pro vyjádření „libovolného počtu“. Důležitá je ale konzistentnost, aby se v jednom modelu nevyskytovaly dva různé symboly, což by mohlo zmást čtenáře. V této práci budeme používat pouze symbol $*$.

2.2 Schematická kategorie

V této sekci popíšeme mechanismus pro konceptuální modelování s názvem schematická kategorie společně s teorií kategorií, na které je založena. Nejdříve ale představíme motivaci za uvedením nového způsobu konceptuálního modelování.

2.2.1 Motivace

Při začátku vývoje databázových systémů bylo popsáno několik databázových modelů dat. Už v roce 1975 je ANSI rozdělila do tří vrstev [18].

- Konceptuální vrstva popisuje část světa, na kterou vymezujeme svůj diskurz.
- Logická vrstva popisuje logickou strukturu dat (např. graf, tabulka, ...).
- Fyzická vrstva popisuje, jak jsou data fyzicky uložena.

Přestože databázových modelů bylo navrženo několik, časem se ukázalo, že nejužitečnější je ten relační. To proto, že data byla často tabulkové povahy. Pokud výjimečně nebyla takové povahy, musela se relačnímu modelu přizpůsobit.

S příchodem potřeby zpracování velkých dat (Big Data) [19] se ukázalo, že relační model není dostačující. Ve velkém množství dat, která spolu nutně nesouvisí, není totiž vždy vhodné hledat tabulkovou strukturu. Ukazuje se, že většina dat reálného světa není relačního charakteru. V různých situacích jsou tedy vhodné různé databázové systémy s různými logickými modely. Namísto přizpůsobování dat logickému modelu se postupně začal více přizpůsobovat logický model datům. Dokonce se začalo používat více databázových systémů najednou v rámci jednoho informačního systému. Například pro cache lze použít key/value store, pro data různorodé povahy lze použít dokumentové databáze a pro silně strukturovaná data starší relační databáze.

Naše vize do budoucna je mít jediný databázový systém, který má jediné rozhraní, jediný způsob modelování dat a jediný dotazovací jazyk, ale zároveň není závislý na fyzické vrstvě. V současných systémech je často tvorba a iterace databázových schémat časově náročná a obtěžující. Chceme proto sloučit konceptuální a logickou vrstvu a předejít tak problémům a lidskému rozhodování při převodu mezi nimi. Vznikne tak pouze jedna vrstva, ve které se pracuje unifikovaným, konceptuálním způsobem.

Prvním krokem v této vizi by mohly být schematické kategorie, které uživateli umožní popsat strukturu dat, se kterými chce v databázi pracovat. Jejich koncept popisují Martin Svoboda, Pavel Čontoš a Irena Holubová [3].

Prostředky ER jsou vhodné ke konceptuálnímu modelování, nicméně mají několik nevýhod. Některé z nich zde identifikujeme.

- V ER lze často modelovat jeden případ mnoha způsoby a není předem jasné, který z nich je nejlepší. Schematické kategorie většinu takových rozdílů smažou, zejména rozdíl mezi entitními typy, vztahovými typy a atributy. Často totiž tyto rozdílů nejsou důležité.
- ER zavádí několik omezení, např. vztahové typy nemohou mít interní identifikátor a složené atributy se nemohou libovolně větvit. Tato omezení vznikají, protože se historicky ER používalo zejména pro konceptuální modelování pro relační databáze. Libovolně větvené atributy se těžko převedou do relačního modelu.

Unified Modeling Language (UML) [2] také umožňuje vytvářet konceptuální schémata. Při vývoji software, zvláště při práci v týmech, je většinou zvoleno UML oproti ER. Je to dáno existencí rozličných nástrojů na vytváření UML diagramů a nástrojů na automatizaci převodu do logické vrstvy. Vyjadřovací schopnost UML je nicméně menší, než u ER. Postupným vývojem UML se expresivnost dodává. Nicméně dosahuje se toho přes konstrukty (např. stereotypy), u kterých lze poznat, že nesouhlasí s původní myšlenkou UML.

2.2.2 Teorie kategorií

Nejdříve popíšeme obecný pojem kategorie z teorie kategorií, na níž je schematická kategorie založena.

Kategorie je matematická struktura, která zobecňuje mnoho jiných matematických struktur. Umožňuje tak mimo jiné studovat vztahy mezi nimi. Poprvé byla představena Eilenbergem a MacLanem v roce 1945 [4].

Kategorie $C = (\mathcal{O}, \mathcal{M}, \circ)$ se skládá z

- množiny objektů \mathcal{O} ,
- množiny morfismů \mathcal{M} ; každý morfismus $f \in \mathcal{M}$ má zdrojový objekt $A \in \mathcal{O}$ (budeme nazývat také *doména*), cílový objekt $B \in \mathcal{O}$ (také *kodoména*), ne nutně různý, a zapisujeme $f : A \rightarrow B$ (f je morfismus z A do B),
- operace skládání $\circ : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$; pro každé dva morfismy $f, g \in \mathcal{M}$, tž. $f : A \rightarrow B, g : B \rightarrow C$, musí $g \circ f \in \mathcal{M}$ (tranzitivita); pro tuto operaci navíc platí vlastnosti
 - asociativita – pro morfismy $f, g, h \in \mathcal{M}$ takové, že $f : A \rightarrow B, g : B \rightarrow C, h : C \rightarrow D$, platí $h \circ (g \circ f) = (h \circ g) \circ f$,
 - identitní morfismy – pro každý morfismus $f \in \mathcal{M}, f : A \rightarrow B$ a jeho objekty A , resp. $B \in \mathcal{O}$ existují morfismy 1_A , resp. $1_B \in \mathcal{M}$, tž. $f \circ 1_A = f = 1_B \circ f$; morfismy 1_A , resp. 1_B nazýváme *identitní morfismy*.

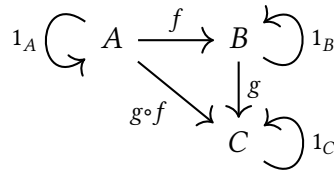
Objekty a morfismy lze definovat i obecněji s použitím tříd místo množin, ale pro naše účely budou stačit množiny.

Jako jednoduchý příklad kategorie uvedeme reálná čísla s neostrou nerovností. Objekty této kategorie jsou reálná čísla $\mathcal{O} = \mathbb{R}$. Pro každá reálná čísla $A, B \in \mathbb{R}$ přidáme morfismus $f : A \rightarrow B$ právě tehdy, když $A \leq B$. Pro všechny morfismy $f, g \in \mathcal{M}$ a objekty $A, B, C \in \mathcal{O}$ takové, že $f : A \rightarrow B$ a $g : B \rightarrow C$ definujeme $g \circ f = h$, kde $h : A \rightarrow C$ a $h \in \mathcal{M}$.

Kategorie je možné vizuálně reprezentovat orientovaným multigrafem, kde vrcholy jsou objekty a orientované hrany morfismy. Příklad této vizualizace je na Obrázku 2.3. Jedná se o kategorii se třemi objekty A, B, C . Všimněme si, že každý objekt má svůj identitní morfismus.

2.2.3 Schematická kategorie

Schematická kategorie je mechanismus na popis konceptuálního schématu dat založený na teorii kategorií. Oproti ER má schematická kategorie větší vyjadřovací sílu. Její koncept společně s algoritmem převodu z ER schématu do schematické kategorie



Obrázek 2.3: Příklad kategorie

uvádí [3]. My však použijeme upravenou definici schematické kategorie navrženou vedoucím práce.

Nejprve zavedeme pomocný pojem *signatura*. Jedá se o řetězec nad abecedou symbolů \mathbb{N} . Prázdný řetězec značíme ε . Operaci zřetězení značíme symbolem \cdot tečky. Příklady signatur: ε , 13, $13 \cdot 7$.

Formálně je schematická kategorie instance kategorie $(\mathcal{O}, \mathcal{M}, \circ)$ taková, že

- každý objekt této kategorie má strukturu trojice: (identita, název, množina identifikátorů), kde
 - identita je libovolný symbol z \mathbb{N} , který umožňuje rozlišit a unikátně identifikovat každý objekt; tedy speciálně i pro případ, kdy by všechny ostatní složky měl totožné s jiným objektem,
 - název reprezentuje textovým řetězcem uživatelské jméno daného objektu, může být i prázdný, pak ho značíme \perp ,
 - množina identifikátorů obsahuje identifikátory; každý jednotlivý identifikátor je množina identit a vyjadřuje, čím lze daný objekt konceptuálně identifikovat (podobně jako identifikátory z ER),
- každý morfismus má strukturu osmice: (signatura, doména, kodoména, název, kardinalita, duplicity, uspořádání),
 - signatura byla již popsána jako pomocný pojem; její účel je umožnit (společně s doménou a kodoménou) rozlišit a unikátně identifikovat každý morfismus v daném schématu; jedná se o řetězec vyjadřující orientovanou cestu, složený ze signatur bazových morfismů (zřetězuje ve stejném pořadí jako se zapisuje skládání morfismů v kategorii, viz Obrázek 2.4),
 - doména a kodoména odpovídají zdrojovému a cílovému objektu tohoto morfismu,
 - směr je buď 0 (tam) nebo 1 (zpět) s výchozí hodnotou 0; hodnota 1 vyjadřuje, že tento morfismus je pouze inverze k jinému, dodaná pro úplnost modelu,
 - název je uživatelské jméno tohoto morfismu, může být i prázdné \perp ,
 - kardinalita je dvojice (min, max), která odpovídá kardinalitě z ER v Sekci 2.1,
 - duplicity a uspořádání jsou booleovské hodnoty (true/false), které konceptuálně modelují, zda jsou pro vztahované instance (modelovány kodoménou), která jsou morfismem spojena k vztahované instanci (modelována doménou), povoleny duplicity, resp. jestli mají být uspořádány; tyto hod-

noty mají význam, pouze pokud je horní hranice kardinality tohoto morfismu $*$; výchozí hodnota obou složek je proto $false$.

Morfismy schematické kategorie rozdělíme na několik vzájemně disjunktních druhů. Příklad každého druhu lze pozorovat na Obrázku 2.4.

- *Bázové* (base) morfismy jsou ty, které vyjadřují konceptuální spojení dvou objektů ze schématu (tedy odpovídají jednotlivým spojením z ER); jejich signatura je jeden unikátní symbol z abecedy.
- *Identitní* (identity) morfismy jsou ty, které vznikly jen kvůli splnění stejnojmenného axiomu z definice kategorie; jejich signatura je ε .
- *Odvozené* (derived) morfismy jsou ty, které vznikly kvůli tranzitivitě (tedy aby byly morfismy uzavřené na operaci skládání); jejich signatura je opravdová cesta – zřetěžené signatury morfismů, ze kterých byl tento morfismus vytvořen.

Ve schematické kategorii bez újmy na vyjadřovací schopnosti zakážeme bázové smyčky (tj. bázový morfismus, jehož doména a kodoména je totožná). Pokud chceme vyjádřit rekursivní vztah, můžeme bázovou smyčku nahradit objektem, který tento vztah reprezentuje. Navíc dodáme dva bázové morfismy, které spojí objekt vztahu s objektem, jehož se vztah týká.

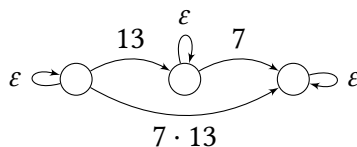
Operaci skládání morfismů \circ ve schematické kategorii lze definovat následovně. Pro dva morfismy $f, g \in \mathcal{M}$ a objekty $A, B, C \in \mathcal{O}$ takové, že $f : A \rightarrow B, g : B \rightarrow C$ a

- f se skládá z $(sig_1, dom_1, mid, name_1, (min_1, max_1), dup_1, ord_1)$,
- g se skládá z $(sig_2, mid, cod_2, name_2, (min_2, max_2), dup_2, ord_2)$,

je jejich složením $g \circ f$ morfismus $h \in \mathcal{M}, h : A \rightarrow C$ skládající se z

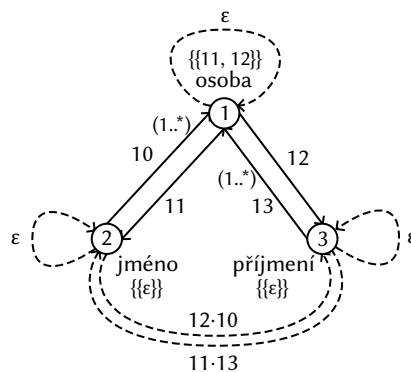
$$(sig_2 \cdot sig_1, dom_1, cod_2, \perp, (\min(min_1, min_2), \max(max_1, max_2)), dup_1 \vee dup_2, ord_1 \wedge ord_2).$$

Protože f a g na sebe navazují, musí být kodoména f totožná s doménou g , označili jsme ji mid . Dále, protože dom_1, mid a cod_2 odpovídají identitám objektů A, B , resp. C , složený morfismus má doménu dom_1 a kodoménu cod_2 . Když u jednoho z morfismů záleží na duplicitách bude záležet i u složeného na duplicitách. Aby záleželo u složeného morfismu na uspořádání, musí však záležet na uspořádání u obou skládaných morfismů.



Obrázek 2.4: Signatury morfismů, \cdot je operace konkatence (zřetězování) symbolů

Na Obrázku 2.5 je příklad schematické kategorie se třemi objekty. Objekt „osoba“ je identifikován dohromady dvojicí objektů „jméno“ a „příjmení“. Objekty „jméno“ a „příjmení“ jsou každý identifikován sám sebou, proto jim náleží jediný identifikátor vždy v podobě $\{\varepsilon\}$. Na obrázku jsou dále bázové morfismy (plné čáry) se svými signaturami a kardinalitami, přičemž výchozí $(1, 1)$ neuvádíme. Identitní a dva vybrané odvozené morfismy jsou vyznačeny čárkovanými křivkami. Jejich signatury jsou složené řetězce, resp. prázdné řetězce ε .



Obrázek 2.5: Příklad schematické kategorie

2.3 Vizualizace schematické kategorie

Schematická kategorie je formálně zadaný model. Pro potenciálního uživatele schematické kategorie je její jednoduitnost a informativnost příliš nepřehledná. Navrháme proto způsob vizualizace, který přinese odlišení a zvýraznění některých důležitých konstruktů, včetně vyobrazení některých složek objektů a morfismů, ale zachovává vyjadřovací sílu představených schematických kategorií. Pojmenujme ho Vizualizace schematické kategorie (VSK).

Objekt schematické kategorie, který má jediný identifikátor $\{\epsilon\}$ nazveme *self-identifikovaný* objekt. Instance takových objektů jsou identifikovány svými hodnotami. Jsou tedy podobné atributům z ER, a proto je budeme značit kružnicí.

Objekty, které nejsou self-identifikované, jsou určité identifikované jinými objekty. Jsou významově analogické entitním nebo vztahovým typům z ER. Značit je budeme obdélníkem.

Dvojice duálních morfismů budeme vždy značit jedinou neorientovanou hranou vedoucí mezi příslušnými objekty. Nebudeme vizualizovat identitní ani odvozené morfismy. Neukážeme ani signatury morfismů, nejsou totiž potřeba.

Kardinality morfismů budeme vykreslovat blízko domény patřičného morfismu. Výchozí kardinality zobrazovat nebudeme.

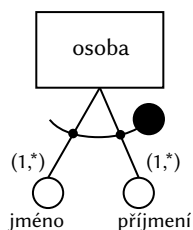
Identifikátory objektů budeme značit stejně jako v ER přeškrtnutím patřičných morfismů, a i v případě jednoduchých identifikátorů.

Složky morfismu *uspořádání* a *duplicita* budou pro výchozí hodnoty (`false` a `false`) neviditelné. Jinak v cílovém objektu na konci spojovací čáry morfismu vyznačíme hodnotu `true` uspořádání symbolem \leq , resp. $+$ pro duplicitu.

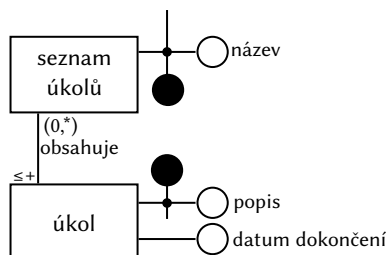
Uživatelské názvy objektů i morfismů budeme vykreslovat, podobně jako v ER, u self-identifikovaných objektů v blízkosti objektu, jinak uvnitř obdélníku.

Na Obrázku 2.6 vidíme diagram odpovídající schematické kategorii z Obrázku 2.5. Sémantika kardinalit $(1, *)$ u obou obrázků taková, že konceptuálně může jedno jméno, resp. příjmení patřit více osobám.

Na Obrázku 2.7 lze vidět vizualizaci schematické kategorie, která má u jednoho z morfismů aktivní uspořádání a duplicitu. Význam tohoto schématu je takový, že seznam úkolů může mít libovolný počet úkolů, a to dokonce se stejným popisem (duplicita), a že je důležité udržovat informaci o pořadí, ve kterém byly jednotlivé úkoly přidávány (uspořádání).



Obrázek 2.6: Příklad vizualizace schematické kategorie



Obrázek 2.7: Vizualizace schematické kategorie s využitými složkami *uspořádání* a *duplicity*

2.4 Převod ER na schematickou kategorii

Algoritmus převodu z ER na schematickou kategorii už popsali Svoboda a kol. [3, s. 192-196]. Pro naši upravenou verzi schematické kategorie však musíme upravit i algoritmus převodu.

Uvědomme si nejdříve, že v ER diagramu, který dostaneme, nemusí mít některý entitní typ ani jeden vlastní atribut. Přestože každý entitní typ musí mít alespoň jeden identifikátor, tento identifikátor nemusí být tvořen vlastním atributem. To ze dvou důvodů, které se vzájemně nevylučují:

1. entitní typ může být dítě v hierarchii, kdy dědí identifikátor od rodiče (případně i nepřímo od předků v hierarchii s více úrovněmi),
2. nebo se může jednat o slabý entitní typ, který má pouze externí identifikátory, a tedy ani jeden interní identifikátor.

Kvůli definici schematické kategorie budeme vybírat unikátní identifikátory z abecedy \mathbb{N} .

Mějme tedy validní ER diagram. Převedeme ho na schematickou kategorii.

Nejdříve definujeme správné pořadí, ve kterém převádět entitní typy. Vezměme závislostní graf, který jsme definovali kvůli zakázání cyklů v Sekci 2.1. Tento graf je z definice orientovaný a pro korektní ER je acyklický. Vezměme jeho libovolné topologické uspořádání. Entitní typy budeme převádět od *největšího* prvku tohoto uspořádání k nejmenšímu. To proto, abychom vždy nejdříve přivedli entitní typ, pro který už jsou vyřešeny všechny ty, na kterých je identifikačně závislý. Jinak by náš převod slabého entitního typu nebo potomka v hierarchii mohl nekorektně využívat ještě nepřevedené závislosti. S tímto pořadím budeme počítat po celý algoritmus.

Kvůli délce algoritmu zde nejdříve naznačíme hrubý postup, než popíšeme jednotlivé kroky. Začneme s prázdnou schematickou kategorií a postupně budeme převádět konstrukty ER do konstruktů schematické kategorie:

- Nejdříve vytvoříme objekt pro každý jednoduchý nebo složený atribut a spojíme je morfismy (složené atributy s jejich atributy).
- Následně vytvoříme objekt pro každý entitní typ, ale ještě mu nepřivedeme identifikátory, protože nemáme v tuto chvíli všechny potřebné morfismy.
- Proto se následně budeme věnovat vztahovým typům – vytvoříme pro každý z nich objekt (zatím zase bez identifikátorů) a spojíme ho morfismy s jeho atributy a účastníky se entitními typy.
- Poté se budeme věnovat identifikátorům objektů entitních typů a vztahových typů.
- Nakonec převedeme ISA hierarchie, včetně zděděných identifikátorů.

Pro každý (jednoduchý nebo složený) atribut z ER vytvoříme odpovídající objekt schematické kategorie (s odpovídajícím názvem a unikátní identitou). Množina identifikátorů objektu bude vždy obsahovat pouze jediný identifikátor $\{\varepsilon\}$, půjde tedy o self-identifikovaný objekt.

Pro každý složený atribut C z ER diagramu a pro každý jeho atribut A zvolíme unikátní symbol $n \in \mathbb{N}$ a vytvoříme dva duální bazové morfismy mezi odpovídajícími objekty O_C a O_A .

$$(n, \text{identita } O_C, \text{identita } O_A, \perp, (1, 1), \text{false}, \text{false}),$$

$$(n, \text{identita } O_A, \text{identita } O_C, \perp, (1, *), \text{false}, \text{false}).$$

Signatury těchto morfismů jsou stejné, abychom poznali, že jsou k sobě duální. Jednotlivé morfismy jsou i přesto v množině rozlišitelné.

Pro každý entitní typ E z ER diagramu vytvoříme jemu odpovídající objekt O_E , ale zatím necháme množinu identifikátorů prázdnou. Pro atributy E vezmeme každý odpovídající objekt O_A a vytvoříme bazový morfismus mezi O_E a O_A (i k němu duální) podobně jako výše. Kardinalitu směrem k atributu nastavíme na kardinalitu A vzhledem k E a kardinalitu směrem zpět na

- $(1, 1)$ pokud je A jednoduchý identifikátor E ,
- jinak $(1, *)$.

Pro každý vztahový typ R z ER diagramu vytvoříme odpovídající objekt O_R a necháme množinu identifikátorů prázdnou. Všechny atributy a složené atributy R spojíme s O_R stejně jako tomu bylo u entitních typů a jejich atributů. Mezi R a každým účastníkem tohoto vztahového typu, označme E , (resp. mezi jejich odpovídajícími objekty) vytvoříme bazový morfismus s kardinalitou odpovídající kardinalitě E vzhledem k R . Duální morfismus k tomu právě vytvořenému bude mít kardinalitu podle povahy celého původního vztahového typu:

- pokud existuje účastník vztahu s horní mezí 1 a E se účastní s horní mezí 1, pak nastavíme kardinalitu duálního morfismu na $(1, 1)$,
- pokud (stejně jako výše) existuje účastník vztahu s horní mezí 1, ale E se neúčastní s horní mezí 1, pak nastavíme $(1, *)$,
- jinak (každý účastník má horní mez $*$) nastavíme kardinalitu na $(1, 1)$.

Nyní se můžeme věnovat identifikátorům entitních typů. Ty převádíme také v pořadí, které jsme pro entitní typy určili. Pro každý identifikátor I entitního typu E

tedy vytvoříme nový identifikátor pro odpovídající objekt schematické kategorie O_E . Převod každého prvku identifikátoru I proběhne následovně.

- Pokud se jedná o vlastní atribut E , převedeme ho jednoduše na signaturu morfismu, který spojuje O_E s odpovídajícím objektem daného atributu.
- Pokud se jedná o vztahový typ, najdeme orientovanou cestu přes odpovídající morfismy z E do každého E_i entitního typu, do kterého vede externí identifikátor. Nechť c je zřetězení signatur morfismů, přes které tato cesta vede. Dále, nechť O_i je objekt odpovídající E_i . Pro každý identifikátor I_k z množiny identifikátorů objektu O_i vytvoříme $S := \{c_k \cdot c \mid c_k \in I_k\}$.

Sjednocením všech těchto množin S a množiny signatur odpovídajících vlastním atributům získáme množinu signatur, která tvoří jeden identifikátor objektu O_E , který odpovídá ER identifikátoru I . Zřetězili jsme totiž signatury identifikátorů „vzdálených entit, které nás identifikují“ se signaturami cest do těchto entit.

Dále nastavíme identifikátory každému objektu O_R , který odpovídá nějakému vztahovému typu R . Odlišíme tři případy:

- Pokud se R účastní externí identifikace, nastavíme identifikátor na množinu S definovanou stejně jako výše (akorát c bude cesta mezi R a E_i).
- Jinak pokud existuje alespoň jeden účastník R , který se vztahu účastní s maximální kardinalitou 1, pak bude identifikátor tvořen signaturami, které vzniknou zřetězením signatury morfismu mezi účastníkem a R s identifikátory účastníka.
- Jinak bude identifikátor tvořen všemi účastníky (zřetězením jako u předchozího případu).

Nakonec převedeme ISA hierarchie. Pro každý vztah rodič-dítě (P, C) z každé ISA hierarchie vezmeme odpovídající objekty O_P, O_C a vytvoříme mezi nimi báze morfismus (z O_C do O_P) s kardinalitou $(1, 1)$ a k němu duální morfismus s kardinalitou rovněž $(1, 1)$. Správně „dořetěžené“ identifikátory (zřetězené přes právě vytvořený morfismus) z O_P vložíme do O_C . Tím O_C „zdědil“ identifikátory od O_P .

Aby schematická kategorie splňovala vlastnosti kategorie, dodáme do ní nejprve všechny identitní morfismy pro každý objekt se signaturou ε a následně také pomocí operace skládání morfismů \circ všechny odvozené morfismy chybějící do tranzitivního uzávěru. Ve skutečnosti tyto morfismy v implementaci neukládáme, mimo jiné protože odvozených morfismů může být nekonečně mnoho. Pouze předpokládáme, že ve schematické kategorii jsou.

Tím je algoritmus dokončen a máme korektní schematickou kategorii odpovídající původnímu ER diagramu.

3. Specifikace

V této kapitole navrhne software, který plánujeme implementovat, pomocí požadavků, konceptuálního datového modelu, procesů, tříd a scénářů. Kromě toho také uvedeme koncept řešení, který poskytne rychlý přehled toho, jak bude systém technologicky navržen. Řídíme se tak běžným postupem softwarového inženýrství [20].

3.1 Požadavky

V této sekci představíme funkční a nefunkční požadavky na systém [20, s. 83].

3.1.1 Funkční požadavky

Vzhledem k většímu množství funkčních požadavků je rozdělíme do několika kategorií, které budou seskupovat požadavky týkající se podobné části systému.

Projekt

- Součástí projektu budou tři typy diagramů – ER, schematická kategorie a vizualizace schematické kategorie.
- Projekt bude obsahovat data, z kterých bude možné obnovit všechny tři diagramy, na kterých uživatel pracuje během jednoho sezení.
- V systému bude možné vytvořit nový projekt, uložit ho a načíst.
- Projekt bude možné pojmenovat pro odlišení od ostatních projektů.

Export

- Jednotlivé diagramy bude možné exportovat do rastrového i vektorového formátu.
- Do těchto exportovaných formátů bude volitelně možné vložit projekt, který z nich pak bude možné načíst. Tímto bude projekt možné otevřít jak v prohlížeči obrázků (a zobrazit rastrově nebo vektorově diagram), tak v našem systému a pokračovat v práci.
- Při exportu do PNG bude možný výběr mezi průhledným a celobarevným pozadím.

Diagramy

- Zobrazení diagramu bude možné posouvat myší.
- Zobrazení diagramu bude možné přibližovat a oddalovat kolečkem myši.
- Posunutí a přiblížení bude volitelně možné synchronizovat mezi všemi diagramy.
- Systém bude kontrolovat validitu uživatelem vytvořených konstruktů.

- Elementy bude možné posunovat držením levého tlačítka myši a tažením.
- Související elementy napříč diagramy se volitelně budou posouvat společně.
- Vlastnosti všech objektů bude možné měnit (popisky, typ, pozice). Tyto změny budou reflektovány v ostatních diagramech.
- Při držení klávesy **Ctrl** bude možné zvolit více objektů najednou postupným klikáním levého tlačítka myši.
- Veškeré elementy bude možné z diagramu mazat alespoň klávesou **Delete**.
- Všechny elementy bude možné zvolit levým tlačítkem myši.

ER Diagram

- Do diagramu bude možné přidat entitní typ (silné i slabé), vztahový typ, atribut (včetně složeného atributu),
- K entitním typům bude možné přidat identifikátory, včetně externích.
- Bude možné vytvořit ISA hierarchii mezi entitními typy.
- Mezi jednotlivými elementy diagramu bude možné přidat spojovací čáru.
- U spojení bude možné specifikovat a zobrazit kardinalitu. Dolní mez bude buď 0 nebo 1, horní mez 1 nebo n . Výchozí kardinality (1, 1) nebudou zobrazeny.
- Uživatel bude moci využít předpřipravené konstrukty, které bude možné vložit do diagramu. Například se může jednat o ISA hierarchie s předpřipravenými entitami.

Schematická kategorie

- Objekty a morfismy schematické kategorie bude možné přidávat a mazat, přičemž tato změna se odrazí v ostatních diagramech.
- V diagramu se zobrazí data o objektech a morfismech včetně kardinalit a popisů. Výchozí kardinality (1, 1) se zobrazovat nebudou.
- Při zvolení objektu se zvýrazní všechny objekty v okolí, které ho identifikují (např. barevně, vzdálené identifikátory jinou barvou).

Vizualizace schematické kategorie

- Vizualizace schematické kategorie ze schematické kategorie zjistí sémantiku a vizualizuje ji různými tvary.
- Objekty a morfismy bude možné přidávat a mazat i ve vizualizaci schematické kategorie.
- Při zvolení objektu se barevně zvýrazní bezprostřední a vzdálené identifikátory různými barvami.

3.1.2 Nefunkční požadavky

- Aplikaci bude možné používat na všech běžných desktopových operačních systémech.
- Nesmí dojít ke ztrátě práce při náhlém ukončení aplikace kvůli interním či externím vlivům. To může být zařízeno např. průběžným ukládáním práce.
- Aplikaci bude možné používat i při výpadku internetového připojení.
- V rámci bezpečnosti žádná data týkající se práce na projektu neopustí zařízení klienta, pokud tak klient explicitně neučiní (například export a přesun souboru).
- Aplikace bude navržena tak, aby bylo možné bez větších komplikací rozšířit její funkcionalitu (např. přidat podporu UML).
- Aplikace bude nenáročná na provoz a investice provozovatele.

3.2 Entity

V této kapitole představíme konceptuální datový model aplikace. Využijeme k tomu prostředky Unified Modeling Language (UML) [2].

Na Obrázku 3.1 je konceptuální schéma modelující ER diagram. Obsahuje třídy odpovídající všem konstruktům popsaných v Sekci 2.1 – entitní typ, vztahový typ, atribut, složený atribut, ISA hierarchie, kardinalita a identifikátor. Entitní typ, vztahový typ i atribut mají složku vyjadřující jejich uživatelské jméno. Navíc jsme uvedli tři třídy asociace k asociacím mezi těmito objekty, které mají všechny svou kardinalitu:

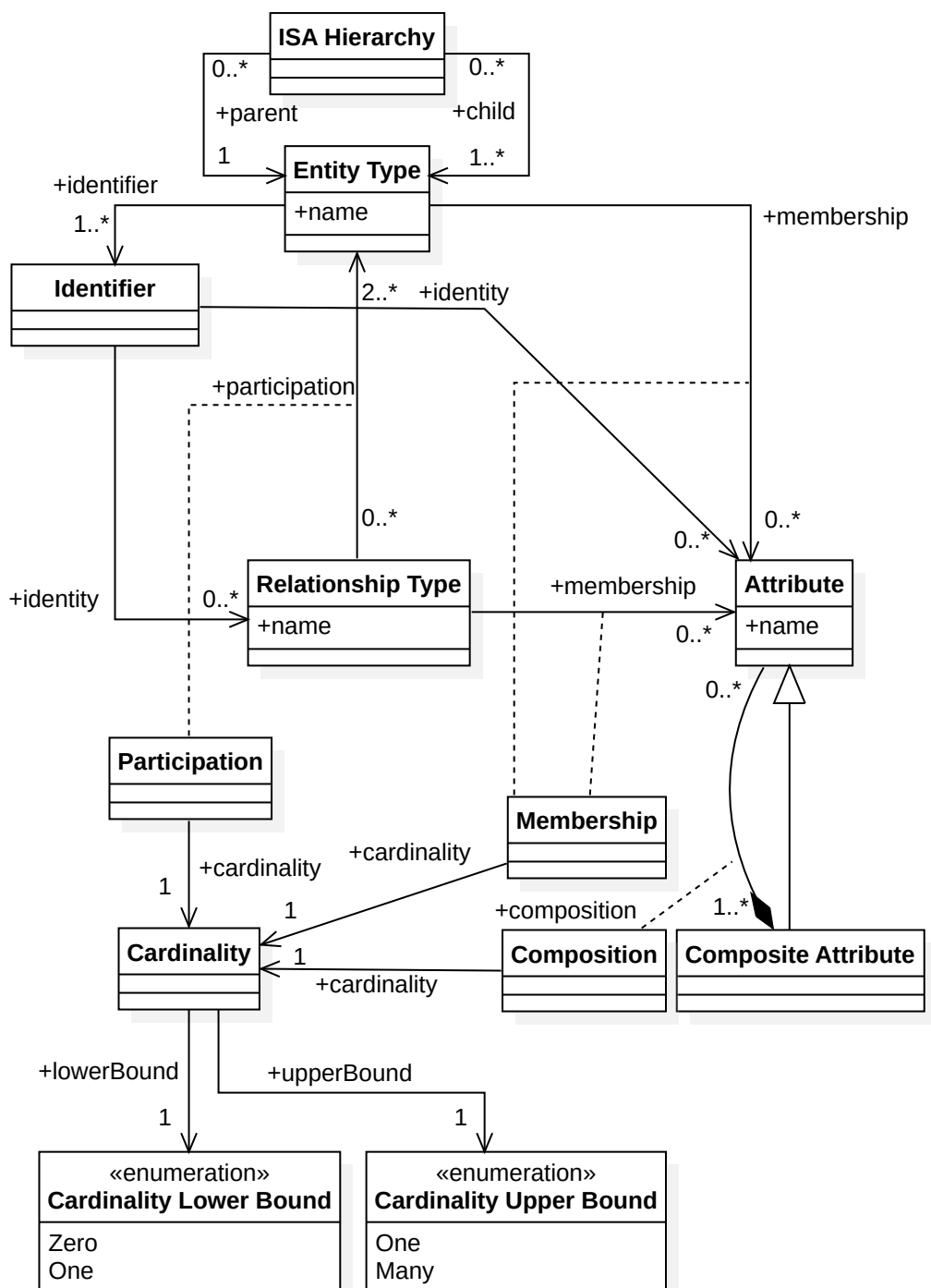
- *Membership* (členství) vyjadřuje vztah mezi entitním typem nebo vztahovým typem a atributem. Toto jméno vyjadřuje, že atributy považujeme za členy (members) entitních a vztahových typů. Entitní i vztahové typy mají buď žádný atribut, nebo libovolné množství atributů.
- *Composition* (složení) vyjadřuje vztah mezi složeným atributem a atributy, z kterých se skládá. Složený atribut musí mít alespoň jeden atribut, jinak ho nepovažujeme za složený.
- *Participation* (účast) vyjadřuje vztah mezi vztahovým typem a entitními typy, které se daného vztahového typu účastní.

Názvy pro naše tři třídy asociace byly inspirovány ER diagramem, který modeluje ER od Atzeniho [17, Obr. 5.22]

Tyto tři třídy asociace jsme mohli sjednotit do jedné, protože všechny obsahují pouze jednu instanci kardinality. Tím by však zanikl jejich rozlišný význam, který může být v určitém kontextu důležitý. Mohli bychom jim také uvést společného předka, ale to považujeme za zbytečné a konceptuálně nemnoho vypovídající.

Kardinalita se skládá ze dvou složek, jejichž možné hodnoty jsou dány enumeracemi. To vyjadřuje naše omezení, která jsme na kardinalitu položili.

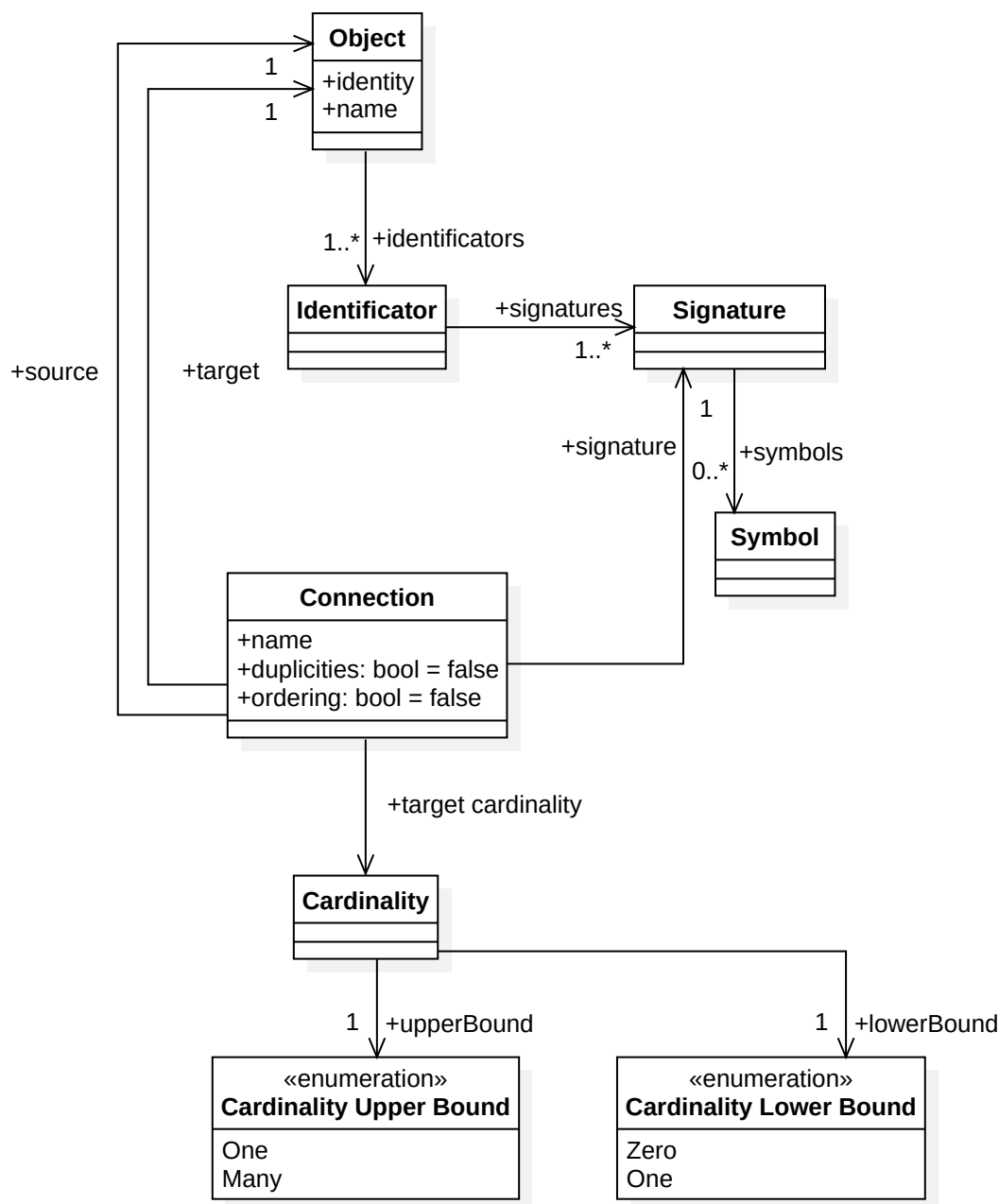
ISA hierarchie se přesně podle definice skládá z jednoho entitního typu rodiče a jednoho či více entitních typů dětí. Mohli jsme místo toho modelovat vztah jednoho rodiče s jedním dítětem, ale v ER je důležité nad těmito vztahy uvažovat po n -ticích.



Obrázek 3.1: Konceptuální schéma – ER diagram

Identifikátory jsme namodelovali bez rozlišení významu externích a interních identifikátorů, protože to je vlastnost, která pro každý identifikátor vyplyne až v instanci modelu. Každý entitní typ musí mít z definice našeho ER modelu alespoň jeden identifikátor. Ten se může skládat z libovolného množství atributů a vztahových typů. V instanci modelu bychom označili identifikátor mající alespoň jeden vztahový typ jako externí, jinak interní. Přestože model technicky povoluje i identifikátory bez členů, nejsou takové identifikátory validní.

Na Obrázku 3.2 je konceptuální model schematické kategorie. Stěžejními třídami v modelu jsou objekt, morfismus a signatura.



Obrázek 3.2: Konceptuální schéma – schematická kategorie

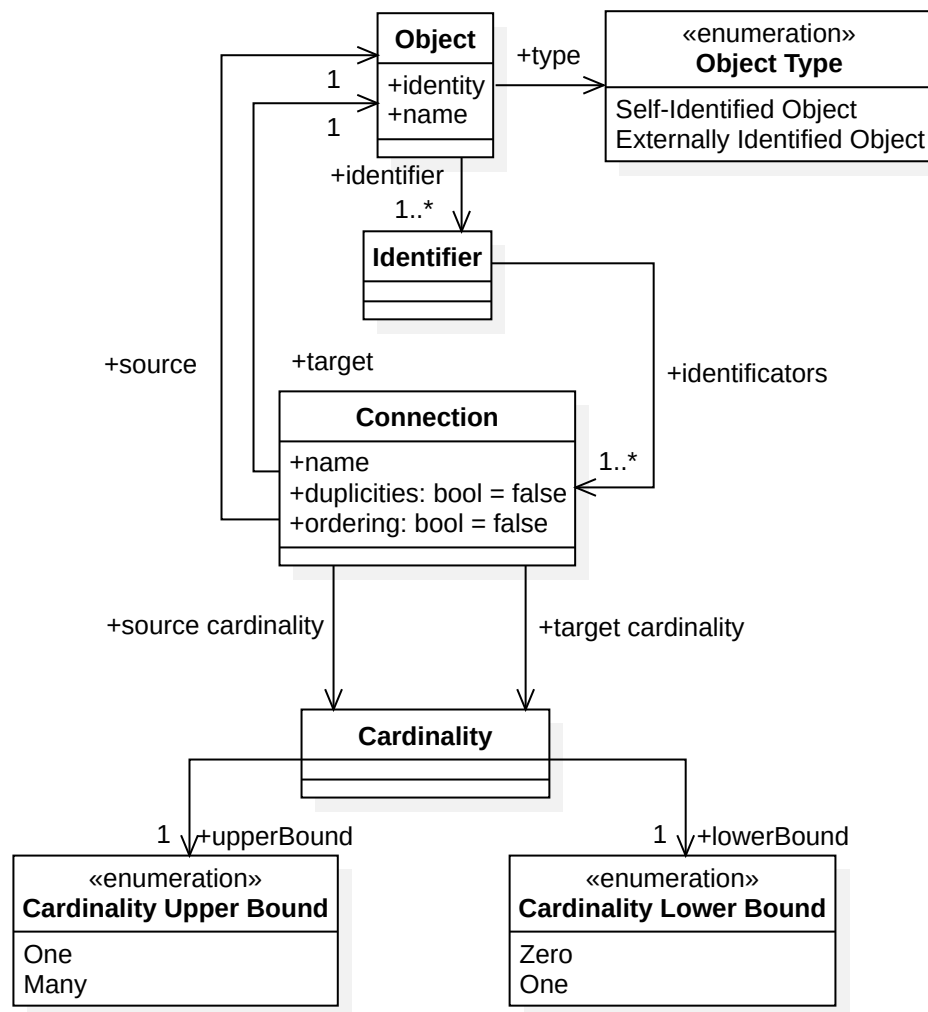
Signatura se skládá z jednotlivých symbolů, kterých může být 0 (prázdný řetězec ε) a více. Jedná se totiž z definice o řetězec. Jednotlivými symboly budou přirozená čísla. Signatury půjde navíc řetězit za sebe. Pokud označíme danou instanci signatury $tato$, zřetězení by mělo probíhat v pořadí $tato \cdot další$, stejně jako je to běžné u jazyků. V implementaci by se mělo dbát na správné pořadí při skládání morfismů, které je opačné a odpovídá pořadí skládání funkcí.

V konceptuálním modelu vyjadřujeme, že každý objekt musí mít alespoň jeden identifikátor, který se skládá ze signatur. Speciálně tímto identifikátorem může být i $\{\varepsilon\}$, protože signatura může být prázdný řetězec. Jak jednotlivé identifikátory, tak celá jejich sada u daného objektu, by měla být množina, a proto se nemusí dbát na absenci pořadí a nemožnost duplicit.

Pro morfismy je vyjádřeno, že by měl mít každý svou signaturu. Mohli bychom

morfismy vydělit na námi definované druhy specializováním třídy morfismu na bázo-
vý, odvozený a identitní. U báзовých by bylo vyjádřeno, že jejich signaturou je pouze
jediný symbol. U odvozených je jejich signaturou již namodelovaná třída signatury.
A u identitních je jejich signaturou prázdný řetězec, tedy konstanta. Toto rozdělení
nám však pro konceptuální model přijde zbytečné, protože se jedná o vlastnost mor-
fismu, která vyplývá z toho, z kolika symbolů se jeho signatura skládá.

Další prvky konceptuálního modelu schematické kategorie jsou zřejmé a vychá-
zejí přímo z definice schematické kategorie ze Sekce 2.2.



Obrázek 3.3: Konceptuální schéma – Vizualizace schematické kategorie

Konceptuální model vizualizace schematické kategorie, který je na Obrázku 3.3, je podobný konceptuálnímu modelu schematické kategorie, s několika málo rozdíly. Du-
ální báзовé morfismy splývají do jedné neorientované hrany Connection. Doména
a kodoména původních duálních morfismů se stávají zdrojovým a cílovým objektem
daného spojení. Kvůli tomuto splynutí má také každá výsledná hrana dvě kardinality
– jednu pro každý morfismus, respektive pro každý objekt, který hrana spojuje. Kon-
cept signatury ve vizualizaci zaniká. Místo toho jsou identifikátory objektů tvořeny
přímo navazujícími morfismy. Objekty a morfismy ztrácí svoji identitu, resp. signatu-
ru. Ta konceptuálně ve vizualizaci schematické kategorie není.

V konceptuálním modelu vizualizace schematické kategorie nově rozlišujeme dva
typy objektů – self-identifikovaný objekt (definovaný v Sekci 2.3) a naproti tomu ex-

terně identifikovaný objekt. Rozlišujeme je kvůli tomu, že ve vizualizaci schematické kategorie jsou tyto odlišné typy objektů zobrazeny různými tvary.

Další koncepty jsou totožné s konceptuálním modelem schematické kategorie.

3.3 Procesy

Součástí specifikace softwarového systému jsou i procesy. Ty uvedeme v této sekci.

Modelování entitního typu v ER diagramu

Uživatel mezi existující elementy diagramu vloží nový entitní typ.

Kroky zahrnují:

1. Přidání nového prvku do diagramu.
2. Zvolení druhu prvku – entitní typ.
3. Zadání popisku (názu entitního typu).
4. Případná změna pozice entitního typu v diagramu.
5. Případné přidání atributů k entitnímu typu a volba jejich kardinality.

Modelování vztahového typu v ER diagramu

Uživatel bude dále chtít do ER diagramu přidat vztahový typ. Vztahový typ musí mít alespoň dva (ne nutně různé) účastníky, jimiž mohou být pouze entitní typy.

Kroky zahrnují:

1. Přidání nového prvku do diagramu.
2. Zvolení druhu prvku – vztahový typ.
3. Zadání popisku (názu vztahového typu).
4. Zvolení pozice vztahového typu v diagramu.
5. Zvolení účastníků vztahového typu.
6. Zvolení kardinalit účastníků vztahového typu.

Modelování atributu ER diagramu

Modelování samostatného atributu může probíhat podobně jako v předchozích scénářích, ale protože se jedná o jeden z nejčastějších procesů, nabídneme alternativně zkratku, kterou zde popíšeme.

Kroky zahrnují:

1. Zvolení entitního nebo vztahového typu, ke kterému přidat atribut, nebo případně zvolení atributu, který se má tímto stát složeným.
2. Přidání atributu a jeho spojení se zvoleným typem.
3. Zadání názvu atributu.
4. Případná změna pozice atributu v diagramu.

Editace existujících prvků ER diagramu

Úprava existujících prvků probíhá pro všechny prvky podobně. Při modelování může uživatel přehodnotit model a upravit vlastnosti některých prvků.

Kroky zahrnují:

1. Zvolení prvku v diagramu k editaci.
2. Zvolení vlastnosti prvku, která bude upravena.
3. Zadání/změna hodnoty vlastnosti prvku.

Mazání prvků v ER diagramu

Při modelování může uživatel přehodnotit model a odstranit některé prvky, načež je případně nahradit jinými.

Kroky zahrnují:

1. Zvolení prvku či prvků v diagramu, které chceme smazat.
2. Odstranění prvků z diagramu.
3. Odstranění souvisejících spojení s jinými prvky diagramu.
4. U entitních typů odstranění případných identifikátorů z diagramu.

Volba identifikátoru entitního typu v ER diagramu

Při modelování entitního typu bude uživatel muset přidat jeho identifikátory.

Kroky zahrnují:

1. Zvolení jednoho a více atributů či vztahových typů, které mají tvořit nový identifikátor.
2. Přidání identifikátoru do diagramu.
3. Kontrola validity ER, protože mohl vzniknout redundantní identifikátor.

Změna typu elementu ER diagramu

Uživatel potřebuje měnit typ elementu diagramu, například když se rozhodne, že je v daném případě lepší modelovat entitní typ jako složený atribut.

Kroky zahrnují:

1. Specifikace elementu diagramu, který bude uživatel měnit (entitní typ, vztahový typ, nebo atribut).
2. Volba typu elementu, na který se má element změnit.
3. Kontrola validity diagramu. Úpravou mohl vzniknout nevalidní ER diagram. Uživatel je na chyby upozorněn a je na něm, aby diagram upravil do validního stavu.

Kontrola validity ER diagramu

Při každé zásadní změně systém musí kontrolovat, zda je diagram stále validní a případně nalezení chyb uživatele upozornit.

Kroky zahrnují:

1. Kontrola toho, že každý entitní typ má alespoň jeden identifikátor, včetně identifikátorů z dědičnosti z ISA hierarchií.
2. Kontrola toho, že pokud je entitní typ identifikován vztahovým typem, pak je v něm účastněn s kardinalitou (1, 1).
3. Kontrola atributů a jejich spojení – složené atributy musí být spojeny s právě jedním entitním nebo vztahovým typem a jedním nebo více jednoduchými atributy a s ničím víc. Jednoduché atributy jsou spojeny buď s právě jedním entitním nebo vztahovým typem, nebo s právě jedním složeným atributem.
4. Kontrola vztahových typů – účastní se jich alespoň dva (ne nutně různé) entitní typy a libovolný počet atributů a nic víc.
5. Kontrola existence referovaných prvků v diagramu – pokud byl jeden z účastníků spojení nebo identifikace odstraněn z datového modelu, pak musí být odstraněno i dané spojení/identifikátor.
6. Kontrola účastníků ISA hierarchií – zda existují a zda se jedná o entitní typy.
7. Kontrola neexistence redundantních identifikátorů, které byly popsány v Sekci 2.1.
8. Validace neexistence cyklů slabých entitních typů a hierarchií, jak byla popsána v Sekci 2.1.
9. Zobrazení všech chyb, pokud byly nějaké nalezeny.

Změna zobrazení

Při modelování se uživatel bude chtít zaměřit na jednu část diagramu přesunem plátna a přiblížením.

Kroky zahrnují

1. Zvolit diagram, jehož plátno se bude přesouvat a přibližovat.
2. Pokud je zvolena synchronizace pláten, pak stejné přiblížení a přesun nastavit i v ostatních plátnech.
3. Omezit přiblížení na nějaké minimální a maximální, aby nedošlo k dezorientování uživatele v plátně.
4. Nastavit přiblížení a přesun plátna na uživatelem žádané.

Export libovolného diagramu

Uživatel bude chtít diagram exportovat, pro přenos na jiné médium, případně vložení do externího dokumentu.

Kroky zahrnují:

1. Zvolení akce exportování diagramu.
2. Zvolení formátu výsledného souboru.
3. Zvolení diagramu, který bude exportován. Výchozím zvoleným diagramem bude ten, na kterém uživatel naposledy pracoval.

4. Pokud je pro formu exportu zvolen obrázek, je nabídnuta volba vložení dat projektu do obrázku, aby šel později v systému upravit.
5. Pokud je formátem rastrový obrázek, je nabídnuto vložení pozadí s volitelnou barvou.
6. Diagram je exportován do zvoleného formátu.

Další procesy uvádět nebudeme, protože jsou analogické těm popsaným. Zaměřili jsme se jen na ty nejdůležitější reprezentanty.

Převod na schematickou kategorii

Změny ER diagramu se projeví ve schematické kategorii.

1. Uživatel upraví ER diagram.
2. Změny se projeví v diagramu schematické kategorie.
3. Změny ve schematické kategorii se projeví ve vizualizaci schematické kategorie.

Vizualizace identifikátorů ve schematické kategorii

V diagramu schematické kategorie nejsou nijak vyznačené identifikátory objektů, protože jinak by diagram byl moc nepřehledný. K účelu komunikace informace o identifikátorech slouží tento proces, který zobrazí pouze identifikátory jednoho zvoleného objektu.

Kroky zahrnují:

1. Volba objektu, jehož identifikátory mají být zobrazeny.
2. Zvýraznění každého identifikátoru (např. barevně).
3. Volitelně zvýraznění i identifikátorů objektů, které jsou součástí identifikátorů z předchozího kroku. Je tak viditelné vícero vrstev identifikace.

Další procesy již popisovat nebudeme, neboť jsou analogické k již popsaným. Zaměřili jsme se pouze na ty nejdůležitější reprezentanty.

3.4 Koncept

Zejména kvůli nefunkčnímu požadavku na možnost použití aplikace na všech běžných desktopových operačních systémech volíme za řešení webovou aplikaci. Pro nenáročnost na provoz a bezpečnost dat se bude jednat konkrétně o statickou webovou aplikaci. To si můžeme dovolit díky tomu, že aplikace nepotřebuje databázi ani složitý back-end.

Aby se jednalo o statickou webovou aplikaci, server musí umět pouze odesílat webové stránky uživateli. Výsledný formát aplikace musí být tedy soubory webových technologií, které se budou bez úprav odesílat do prohlížeče uživatele. Docílí se toho překladem z moderních webových technologií vyššího řádu do starších webových technologií nižšího řádu (HTML, JavaScript, CSS).

Zvoleným programovacím jazykem bude TypeScript, který byl vytvořen společností Microsoft [21]. Tento jazyk je nadstavbou jazyka JavaScript, která přidává mj.

statické typové kontroly. Statické typové kontroly umožňují udržitelnost větších projektů, protože vytváří kontrakty mezi částmi aplikace (např. typy parametrů funkce), na rozdíl od jazyka JavaScript, který je dynamicky typovaný. Kontroly se uskutečňují při překladač do jazyka JavaScript.

Jako framework zvolíme React od společnosti Meta Open Source [16]. Jedná se o framework pro tvorbu webových aplikací a uživatelských rozhraní, který používá vývojové paradigma tzv. komponent. Idea je taková, že vývojář vytváří malé komponenty, které skládá do větších komponent, z kterých nakonec složí webovou aplikaci. Pod komponentou si lze představit nějaký ovládací prvek uživatelského rozhraní, např. tlačítko, nebo např. entitní typ z ER. React podporuje TypeScript, což odpovídá našemu zvolenému programovacímu jazyku. Tento framework byl v roce 2022 nejpopulárnější front-end webový framework [22]. Tím máme zaručenu velkou komunitu vývojářů a velký výběr knihoven.

Diagramy budeme vykreslovat pomocí Scalable Vector Graphics (SVG) [10]. To je přímo podporované v HTML většinou moderních webových prohlížečů.

Hlavní komponentou aplikace bude SVG plátno, ve kterém budou položeny komponenty jednotlivých konstruktů diagramů, které budou složeny ze SVG tvarů. Kromě pláten budeme potřebovat modul s uživatelskými prvky, jako jsou tlačítka pro menu a ovládací prvky pro změnu vlastností konstruktů v diagramech. Posledním hlavním modulem, bude modul s modelem aplikace, obsahující nejrozličnější třídy a rozhraní, které budou odpovídat jednotlivým konstruktům a diagramům.

3.5 Třídy

V této sekci definujeme třídy datového modelu našeho systému. Rozšíříme a konkretizujeme tím konceptuální model ze Sekce 3.2.

Kvůli zvoleným knihovnám, které představíme později při implementaci systému, jsou na datový model kladeny určité restriktce.

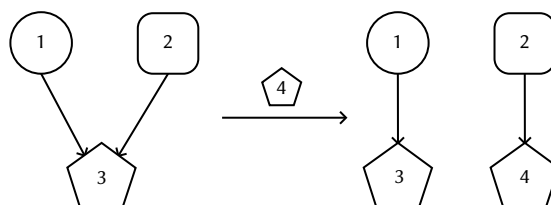
Každá třída musí mít bezparametrický konstrukt. Při deserializaci objektu z JSON v jazyce JavaScript, resp. TypeScript, dostaneme prostý objekt (plain object, tj. objekt bez prototypového řetězku, angl. prototype chain). Tento objekt navíc nebude mít ani funkce, ty totiž do JSON serializovat nelze. Budeme tedy muset převést prostý objekt na instanci třídy a k tomu budeme potřebovat zkonstruovat „výchozí“ instanci. Do té posléze přiřadíme data z prostého objektu. Každá třída tedy musí mít konstrukt bez parametrů, příp. musí mít všechny parametry nastavenou výchozí hodnotu.

Všechny datové složky každé třídy musí mít atributy specifikující jejich typ. Podobně jako předchozí restriktce má i tato jako důvod deserializaci. Stejně jako celé objekty musí být převedeny na instance, tak i jejich hluboce zanořené objekty. Při převádění musí být známo, jaká třída patří danému objektu. Protože TypeScript typové anotace jsou při kompilaci do JavaScriptu odstraněny, musí tohoto být dosaženo atributy, které jsou k dispozici za běhu.

Každá třída modelu musí mít datovou složku `[immerable] = true`. Symbol `immerable` je definován knihovnou Immer [23], kterou popíšeme později při implementaci. Zjednodušeně řečeno, tato datová složka říká knihovně, že instance dané třídy může považovat za použitelné pro svou funkcionalitu.

Celý datový model musí tvořit orientovaný les (tj. orientovaný graf, kde mezi každý-

mi dvěma vrcholy vede nejvýše jedna orientovaná cesta a všechny jeho stromy jsou zakořeněné). Myslíme tím zejména to, že v modelu nejsou vnitřní reference a na každou instanci třídy může držet referenci nejvýše jedna jiná instance (její majitel). Úplná absence cyklů v datovém modelu je nutná k tomu, aby nevznikaly cyklické závislosti při úpravě instance modelu. Tato restrikce vzniká kvůli frameworku React [16]. Ten totiž reaguje na změny v datovém modelu porovnáním referencí, a proto nemůžeme nikdy instance tříd přímo měnit, vždy musíme vytvořit novou instanci. Jinak řečeno, instance tříd jsou immutable. Pokud by potom držely dvě různé instance referenci na nějakou jinou instanci, mohli bychom zapomenout přenastavit obě tyto reference. Navíc tento proces nedělá vývojář manuálně, ale používá knihovnu, která kvůli zjevným výkonnostním důvodům nemůže udržovat všechny reference v instanci modelu tímto způsobem. Vizualizace tohoto problému je vidět na Obrázku 3.4, kde dvě instance (1, 2) referují na jednu instanci (3). Upravíme instanci 3, tedy musíme vytvořit novou instanci (4). Zapomeneme ale aktualizovat referenci z instance 1 (například, protože k ní zrovna nemáme přístup).



Obrázek 3.4: Ukázka problému immutability a referencí

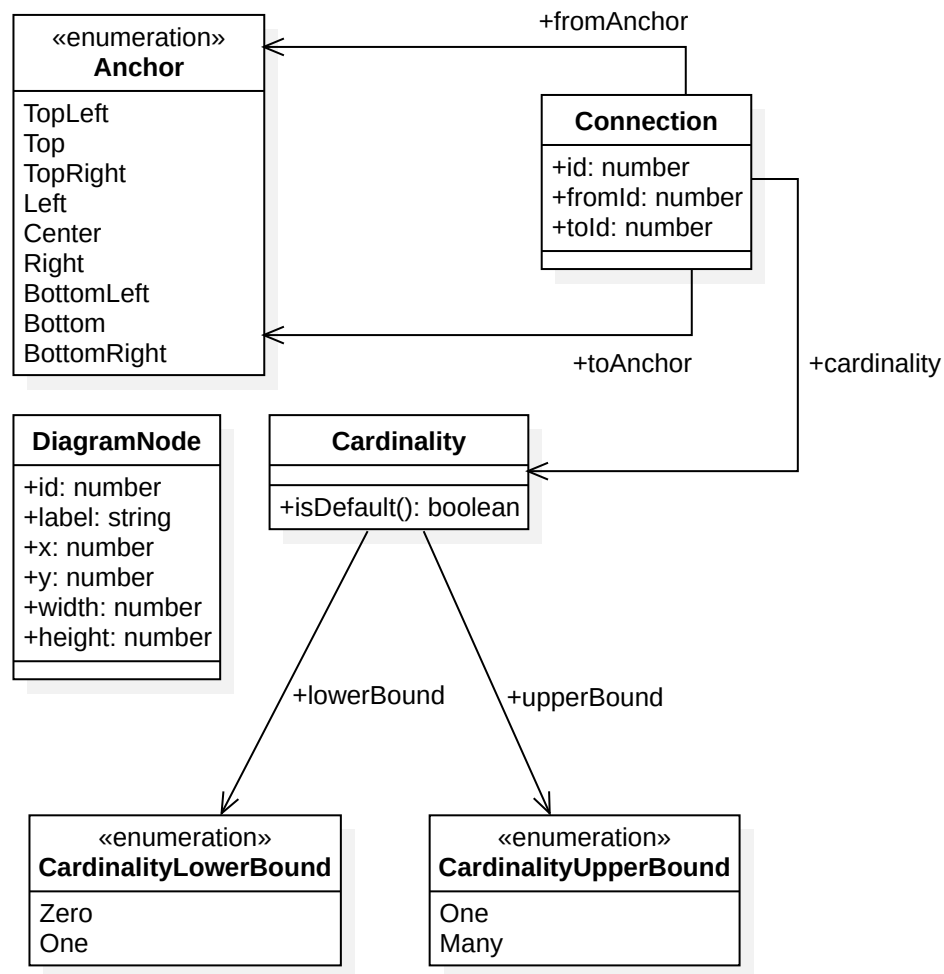
Problém s referencemi lze vyřešit tím, že každá odkazovaná třída v modelu bude mít datovou složku `id`, která bude instance unikátně identifikovat. Potom instance jiných tříd budou referovat pomocí těchto složek. Podobně je tomu například v relačních databázích. Nevýhodou je, že může dojít ke dvěma nevalidním stavům: instance s daným `id` už neexistuje (např. byla smazána), nebo jich existuje více (došlo chybně k duplikaci objektu bez změny `id`). Systém by se měl postarat o to, aby k tomu nedošlo. Bohužel jsme se tímto připravili také o některé výhody, které v JavaScriptu poskytuje garbage collection (GC). Mezi objekty totiž referujeme způsobem, o kterém GC nemůže vědět.

Když teď známe všechny restrikce na náš model, můžeme ho začít navrhovat. Stejně jako pro konceptuální model k tomu použijeme UML třídy s komentářem v textu. Některé výše zmíněné povinné datové složky a atributy budeme v modelu považovat za implicitní.

Nebudeme popisovat úplně celý datový model kvůli jeho rozsahu. Některé záležitosti, které se týkají pouze zobrazování nebo nejsou důležité ve více částech aplikace, vynecháme.

Na Obrázku 3.5 je diagram tříd popisující základní obecné konstrukty, které se týkají všech diagramů v systému. Třída `Cardinality` obsahuje mj. operaci `isDefault`, která říká, zda se jedná o výchozí kardinalitu (námi definovanou jako (1, 1)). V jazyce TypeScript, resp. JavaScript, nelze přetěžovat operátory, všechny operace musí být metody v dané třídě. Dále v nich neexistuje koncept hodnotového porovnání instancí tříd, resp. porovnání objektů. Pokud se porovnávají instance operátorem `==`, porovnávají se reference.

Z těchto důvodů nelze kardinalitu porovnat s jinou instancí bez přítomnosti od-



Obrázek 3.5: Diagram tříd – diagram

povídající metody. V našem případě je pouze potřeba zjistit, zda se jedná o výchozí kardinalitu, a proto definujeme odpovídající operaci.

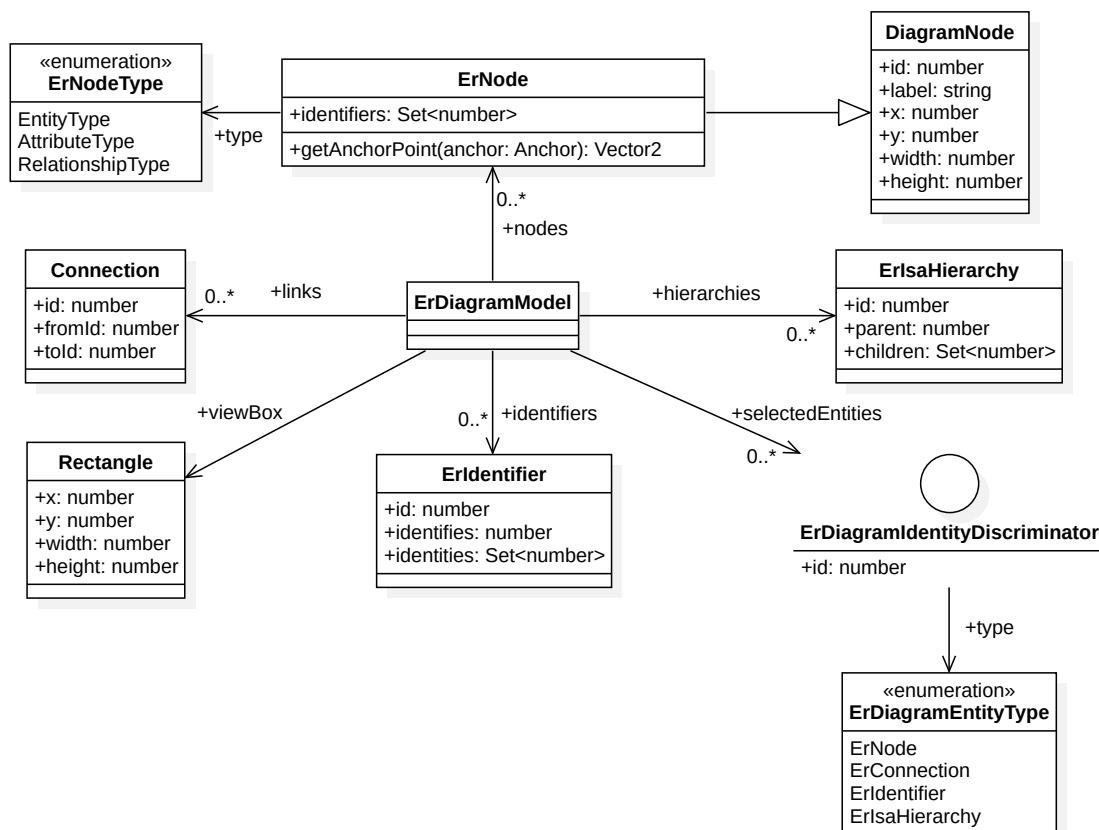
Třída **Anchor** (kotva) se týká pouze zobrazení. Říká, z a do které oblasti elementu diagramu vede odpovídající spojení. Každý element diagramu si sám definuje a vypočítá těchto 9 kotev, které by se měly nacházet po jeho obvodu s výjimkou center. Tyto kotvy jsou relativní k rozměrům a umístění elementu, a proto je musí vždy přepočítat.

Kvůli restrikci o více referencích má třída **DiagramNode** své `id`. Toto bude běžné u většiny tříd, které reprezentují elementy diagramů.

Na Obrázku 3.6 je diagram tříd, které se týkají ER diagramu.

Hlavní třída **ErDiagramModel** je majitel svých **ErNode**, **Connection**, **ErIdentifier** a **ErIsaHierarchy**. Pro zobrazování a uživatelskou interakci drží také rozhraní **ErDiagramIdentityDiscriminator**, které obaluje `id` uživatelem právě vybraných elementů diagramu. Tento obal navíc obsahuje typový diskriminátor `type`, jenž vyjadřuje, o jaký druh elementu se jedná. Tato typová diskriminace je důležitá, aby systém věděl, kde má daný element s daným `id` hledat, zdali mezi `nodes`, `links`, `identifiers`, nebo `hierarchies`.

Podobně je určen typ **ErNode**, který pro element diagramu říká, jak se má zobrazovat. V ER jsme definovali tři hlavní elementy – entitní typ, vztahový typ a atribut. Mohli jsme tyto elementy modelovat jako jednotlivé třídy, ale tento přístup je vhod-



Obrázek 3.6: Diagram tříd – ER

nější, protože rozdíl mezi nimi je pouze ve vizualizaci. Také se tímto způsobem v jazyce TypeScript jednodušeji pozná, o jaký typ má aktuálně držená instance `ErNode`. V reakci na to lze jednodušeji implementovat v systému rozličnou logiku týkající se pouze jednotlivých typů elementů.

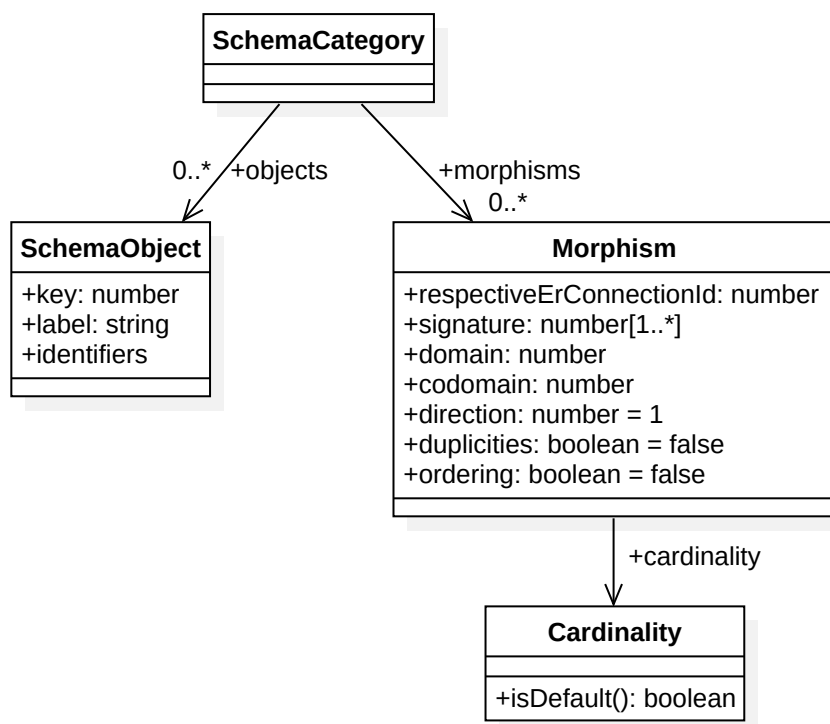
V diagramu používáme množinový typ `Set`, který je v JavaScriptu zabudovaný. Jednodušeji se pak pracuje s odpovídajícími datovými složkami, kde nezáleží na pořadí a duplicitách. Atributy `children` a `identities`, které tento typ mají, mají teoreticky kardinalitu $(1, *)$. V ISA hierarchii musí být alespoň jedno dítě a v identifikátoru alespoň jedna identita.

U třídy `ErIdentifier` bylo nutné názvem rozlišit jednotlivé datové složky. Pojmenování může být matoucí, takže ho zde popíšeme. Identifikátor (identifier) se skládá z jednoho entitního typu, jehož tímto identifikuje (`identifies`). Součástí identifikátoru jsou pak jednotlivé elementy, ze kterých se identifikátor skládá. Říkáme jim identity (`identities`).

Pro správné zobrazení diagramu má třída `ErDiagramModel` také datovou složku `viewBox`. Ta vyjadřuje, na jakou část diagramu se má plátno zaměřit. Jedná se o obdélník, který svou pozicí vyjadřuje přesun plátna a svou šířkou a výškou vyjadřuje přiblížení. Třída `Rectangle` je obecná, a používá se proto i v jiných částech systému, kde je potřeba pracovat s obdélníky.

Diagram tříd, které se týkají schematické kategorie, je na Obrázku 3.7. Význam většiny obsahu tohoto diagramu je evidentní. Upozorníme pouze na atribut `respectivelyErConnectionId`.

Tento atribut vyjadřuje `id` ER spojení, z kterého daný morfismus vznikl. Tato



Obrázek 3.7: Diagram tříd – schematická kategorie

informace je užitečná pro aktualizaci morfismu v návaznosti na úpravu ER diagramu. Také lze při úpravě schematické kategorie některé změny naopak převést zpět do ER.

Objekty schematické kategorie, které vznikly převodem z ER budou mít stejnou hodnotu key jako id odpovídajícího objektu. Tím zaručíme stejnou svázanost diagramů jako tomu je u morfismů.

Dále představíme vybrané třídy, které nesouvisí přímo s diagramy. Pro práci s 2D geometrií budeme potřebovat třídy `Vector2` a `Angle`.

Třída `Vector2` z Obrázku 3.8 reprezentuje dvourozměrný vektor s vhodnými operacemi včetně sčítání a odčítání s jinými vektory, násobení skalárem a negace (unární minus). Další operace jsou vidět na zmíněném obrázku a jejich význam by měl být evidentní. Třída poskytuje také běžně používané vektory jako statické atributy. Operace, které pracují s úhly, používají k jejich reprezentaci třídu `Angle`.

Třída `Angle` z Obrázku 3.9 reprezentuje úhly nezávisle na jejich jednotkách. Je to vhodnější, než používat pouze pomocné metody, které by převáděly stupně na radiány a naopak. Interně je úhel reprezentován ve stupních, protože při počítání s radiány by kvůli menším číslům mohlo docházet ke ztrátě přesnosti.

Metoda `normalized` převede úhel do rozsahu $[0^\circ, 360^\circ)$, resp. $[0, 2\pi)$. To jak pro negativní, tak pro pozitivní úhly mimo tyto rozsahy.

Na Obrázku 3.10 jsou znázorněny dva typy kartézské soustavy souřadnic – pravotočivá a levotočivá. V matematice a geometrii je nejběžnější pravotočivá soustava souřadnic, nicméně SVG používá levotočivou (angl. left-handed), jak je tomu běžné u počítačové grafiky. Pro převod úhlu z běžně uvažovaného pravotočivého do levotočivého SVG úhlu slouží právě metoda `toLeftHandedSystem`. Ta přijde vhod při rotování vektorů, či celých SVG konstruktů.

Třída `GlobalIdGenerator` z Obrázku 3.11 slouží jako generátor identifikátorů

Vector2
+x: number +y: number <u>+zero: Vector2</u> <u>+left: Vector2</u> <u>+up: Vector2</u> <u>+right: Vector2</u> <u>+down: Vector2</u>
+add(other: Vector2): Vector2 +multiply(other: Vector2): Vector2 +subtract(other: Vector2): Vector2 +negate(): Vector2 +distanceTo(other: Vector2): number +normalize(): Vector2 +length(): number +crossProductMagnitude(other: Vector2): number +equals(other: Vector2): boolean +angle(): Angle +toString(): string +rotate(angle: Angle): Vector2 <u>+fromLengthAndAngle(length: number, angle: Angle): Vector2</u>

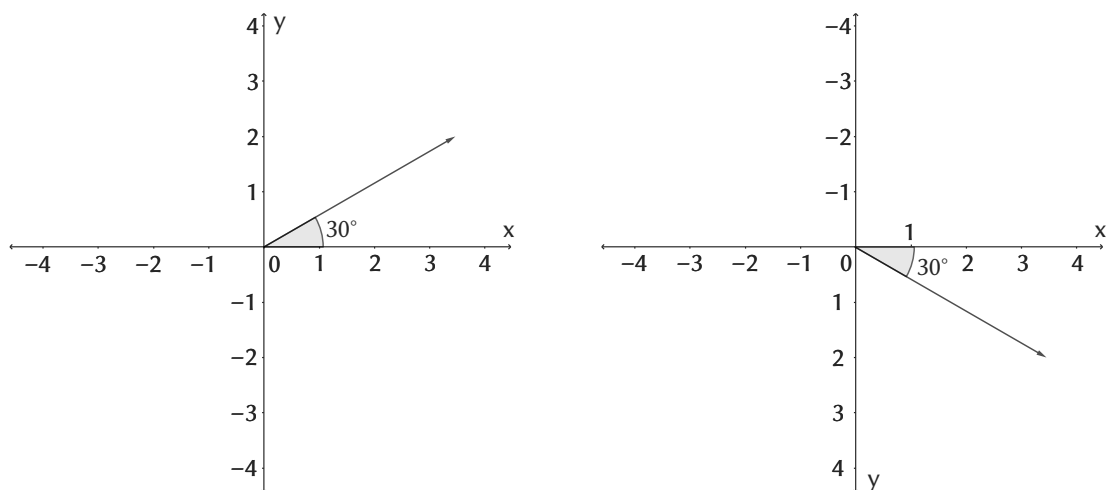
Obrázek 3.8: Diagram tříd – Vector2

Angle
-degrees: number <u>+zero: Angle</u> <u>+fullCircle: Angle</u> <u>+rightAngle: Angle</u>
<u>+fromDeg(degrees: number): Angle</u> <u>+fromRad(radians: number): Angle</u> +deg(): number +rad(): number +normalized(): Angle +negate(): Angle +toLeftHandedSystem(): Angle +isInRangeInclusive(rangeFrom: Angle, rangeTo: Angle): boolean

Obrázek 3.9: Diagram tříd – Angle

a klíčů pro naše objekty. Zajišťuje, že vždy vrátí unikátní číslo jako nový identifikátor. Jednoduchá implementace je začít s číslem 0 a každé další zvětšit o 1. Tato třída je singleton, jeden z objektově orientovaných designových vzorů uvedený Gammou a kol. (také známých jako „Gang of Four“) [24, s. 144]. Tím může existovat nejvýše jedna její instance v běžícím systému.

Třída SvgPathStringBuilder z Obrázku 3.12 konstruuje SVG path data atribut [10, § 9.3]. Obsahuje mnoho dalších metod. Kromě těch, které lze vidět na zmíněném obrázku, například pro každý příkaz lze pozice (location) specifikovat i relativně k předchozí pozici (místo absolutně). Třída odpovídá vzoru builder [24, s. 110]. Užívání této třídy odpovídá vypisování „příkazů“ pro path data atribut. Užitečnost této třídy spočívá v přehlednosti – je lepší používat její metody s deskriptivním názvem místo path data příkazů, které mají formu jednoho písmene a argumentů.



Obrázek 3.10: Pravotočivá (vlevo) a levotočivá (vpravo) kartézská soustava souřadnic

GlobalIdGenerator
-id: number = 0 -instance: GlobalIdGenerator
-constructor() +nextId(): number +setId(value: number) +getInstance(): GlobalIdGenerator

Obrázek 3.11: Diagram tříd – GlobalIdGenerator

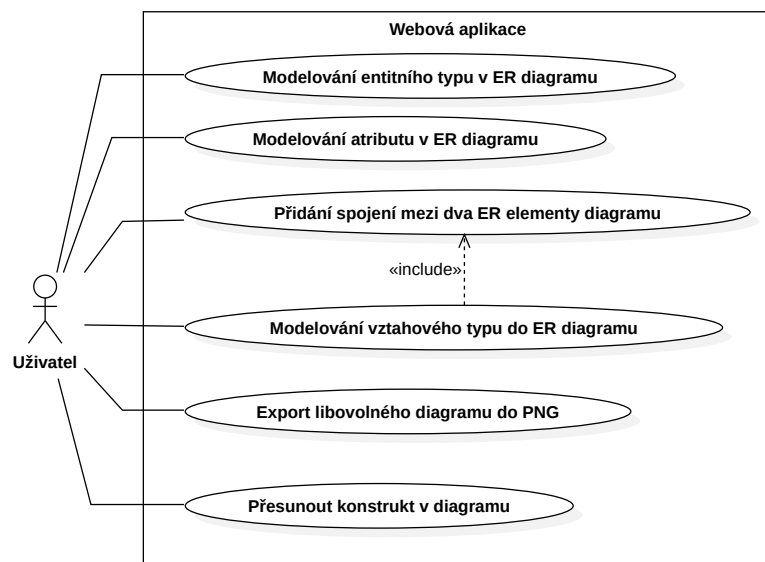
SvgPathStringBuilder
-commands: string[0..*]
+start(location: Vector2) +move(location: Vector2) +lineTo(location: Vector2) +close() +cubicBezier(controlPoint1: Vector2, controlPoint2: Vector2, end: Vector2) +quadraticBezier(controlPoint: Vector2, end: Vector2) +arc(radiusX: number, radiusY: number, xAxisRotation: number, largeArc: boolean, sweep: boolean, location: Vector2) +isEmpty(): boolean +toString(): string -constructor(degrees: number)

Obrázek 3.12: Diagram tříd – SvgPathStringBuilder

3.6 Scénáře

V této sekci uvedeme scénáře pro případy užití [25, s. 65], kterými rozšíříme procesy.

Případy užití nebudeme popisovat všechny, kvůli stručnosti. Navíc některé případy užití jsou si velmi podobné, například export do SVG je téměř stejný jako export do PNG, akorát se u SVG nevolí barva pozadí. Zvolíme tedy pouze případy užití, které jsou nějak modelové nebo zásadní.



Obrázek 3.13: Diagram případů užití

Modelování entitního typu v ER diagramu

Počáteční stav: V systému je rozdělaný projekt. ER diagram lze editovat.

Běžný průběh:

1. Uživatel přetáhne myší konstrukt „entitní typ“ z panelu konstruktů do ER diagramu.
2. V diagramu je vytvořen nový entitní typ s výchozím názvem.
3. Uživatel zvolí právě vytvořený entitní typ myší.
4. V panelu „Control Panel“ upraví vlastnost entitního typu „label“.
5. Název entitního typu se změní na uživatelem definovaný.

Stav systému po dokončení: V diagramu je nový entitní typ, který má uživatelem definovaný název. Změna se projeví i ve schematické kategorii, respektive v její vizualizaci.

Modelování vztahového typu v ER diagramu

Počáteční stav: V diagramu je entitní typ nebo více entitních typů, které se budou účastnit vztahového typu. Případně je lze vytvořit i v průběhu toho případu užití. Změna se projeví i ve schematické kategorii, resp. v její vizualizaci.

Běžný průběh:

1. Uživatel z panelu s konstrukty myší přetáhne vztahový typ do ER diagramu.
2. V ER diagramu se objeví nový vztahový typ.
3. Uživatel v ovládacím panelu změní název vztahového typu.
4. Uživatel spojí entitní typ se vztahovým typem podle už popsaného případu užití spojování elementů. Tím přidá účastníka vztahového typu.
5. To zopakuje s dalšími účastníky (ne nutně jinými).

Stav systému po dokončení: V systému je nyní nový vztahový typ s účastníky. To se projeví i ve schematické kategorii, resp. v její vizualizaci.

Modelování atributu v ER diagramu

Počáteční stav: V ER diagramu je entitní typ. Diagram lze editovat.

Běžný průběh:

1. Uživatel pravým tlačítkem myši klikne na entitní typ nebo vztahový typ.
2. Otevře se kontextové menu.
3. Uživatel zvolí položku „Add attribute“.
4. Do ER diagramu je přidán nový atribut, který je spojen s entitním typem.
5. Uživatel zvolí právě vytvořený atribut myší.
6. V ovládacím panelu upraví vlastnost „label“.
7. Název atributu je nastaven na právě uživatelem definovaný.
8. Uživatel případně zvolí spojení mezi entitním typem a atributem myší.
9. V ovládacím panelu zvolí kardinalitu ze čtyř předdefinovaných možností.
10. Tato kardinalita je pro spojení nastavena.

Stav systému po dokončení: V modelu systému je uložen nový atribut entitního typu, který má daný název a kardinalitu. Změna se projeví i ve schematické kategorii, resp. v její vizualizaci.

Přidání spojení mezi dva ER elementy diagramu

Počáteční stav: V ER diagramu dva existující elementy.

Běžný průběh:

1. Uživatel zvolí první element levým tlačítkem myši.
2. Uživatel klikne na druhý element pravým tlačítkem myši.
3. Zobrazí se kontextové menu.
4. Uživatel zvolí položku „New connection“.
5. Mezi elementy se vloží spojení.

Možné chyby:

- Elementy mohou být nekompatibilní – např. dva vztahové typy, dva entitní typy. V takovém případě se spojení nevytvoří a zobrazí se chybová hláška.

Stav systému po dokončení: V modelu i v zobrazeném diagramu bude mezi elementy nové spojení.

Export libovolného diagramu do PNG

Počáteční stav: V systému je otevřený projekt.

Běžný průběh:

1. Uživatel zvolí položku **File** » **Export as** » **PNG** v hlavním menu aplikace.
2. Otevře se dialog s možnostmi exportu s výběrem diagramu k exportu. Výchozí zvolený diagram bude ten, ve kterém naposledy uživatel pracoval.
3. V dialogu dále uživatel zvolí, jestli chce do PNG souboru zahrnout i serializovanou verzi projektu, aby šel později PNG soubor otevřít v aplikaci.
4. Uživatel v dialogu dále zvolí, jestli má PNG mít barvu pozadí, případně jakou. Výchozí nastavení je PNG bez pozadí, tedy transparentní.
5. Uživatel potvrdí dialog.
6. V prohlížeči uživatele se „stáhne“ exportovaný PNG soubor, který má název odpovídající názvu projektu.

Možné chyby:

- V zařízení uživatele není dostatek persistentní paměti na uložení obrázku. O zachycení a obstarání této chyby se stará webový prohlížeč uživatele.
- Soubor se stejným názvem již existuje. O zachycení a obstarání se rovněž stará webový prohlížeč uživatele.
- Uživatel zruší dialog místo potvrzení. V takovém případě se nic dalšího nestane (nedojde k exportu).

Stav systému po dokončení: Ve složce nastavené ve webovém prohlížeči uživatele na uložení stahovaných souborů bude uložen PNG soubor s rasterizovaným diagramem.

4. Implementace

V této kapitole popíšeme postup implementace aplikace. Nejdříve představíme technologie a knihovny, které jsme k vývoji použili. Dále uvedeme uživatelskou dokumentaci, kde popíšeme fungování a způsob použití aplikace z pohledu uživatele. Následně se v technické dokumentaci budeme věnovat ze širšího pohledu zdrojovému kódu, konkrétně jednotlivým modulům a komponentám a jejich účelu. Posléze se budeme věnovat tomu, jak připravit své vývojové prostředí k vývoji aplikace. Nakonec předvedeme, jak projekt sestavit a výsledek nasadit.

4.1 Použité technologie

Jak už jsme ve stručnosti naznačili v Sekci 3.4, k vývoji jsme použili framework React [16]. React zjednodušuje tvorbu webových uživatelských rozhraní a celých webových aplikací tím, že umožňuje vytvářet celé komponenty a skládat z nich další.

Dále také bylo zmíněno, že jako programovací jazyk jsme zvolili TypeScript [21], který k jazyku JavaScript přidává statické typování. Výsledkem kompilace je JavaScript kód, který lze interpretovat prohlížečem. Většina použitých knihoven, včetně frameworku React, má pro TypeScript přímou podporu v podobě typových anotací.

Pro správu globálního stavu aplikace jsme použili knihovnu Zustand [26]. V React totiž každá komponenta má svůj stav, který předává jako vlastnosti svým dětem. Úprava stavu musí být uskutečněna dětmi pomocí události, kterou odebírá rodič. V našem případě, kdy v jednom diagramu bude mnoho volných komponent (konstruktů), které mezi sebou potřebují komunikovat, tento přístup není vhodný. Přestože v React existují způsoby pro sdílení globálního stavu (React Context), jejich metoda sdílení stavu pro naši aplikaci není vhodná. Zustand umožňuje vytvořit React Hook, pomocí kterého můžeme v libovolné komponentě číst i upravovat jeden globální stav.

Knihovna Zundo [27] umožňuje globální stav vracet o stav zpět a dopředu (undo/redo). Ovšem zachytává i nepatrné změny, a tak se sledování stavů musí vhodně vypínat a zapínat, abychom podchytili jen ty logické změny, které pro nás mají nějaký ucelený význam.

Dále využijeme knihovnu Immer [23], která umožňuje v React jednodušeji aktualizovat stav komponent. Při použití frameworku React by se totiž stav přímo upravovat neměl, protože by nebyla zaregistrována jeho změna, React by na ni nemohl náležitě zareagovat a došlo by k nekonzistenci. React změnu stavu detekuje referenčním (a ne hlubokým) porovnáním předchozího a nového stavu komponenty. Při každé změně stavu se proto běžně musí vytvořit nový objekt. Pokud je objekt hluboce zanořený, je tvorba nového objektu velmi explicitní a zdlouhavá na programování. React by se správně měl používat bez zanořených stavů, takže každá komponenta se stará o svůj minimální stav. Protože však používáme jeden složitý globální stav, Immer je vhodná pomoc. Immer udělá většinu práce za nás tím, že za běhu vytvoří tzv. draft (pracovní verze) stavu a sleduje změny, které na něm program dělá. Pomocí těchto změn pak vytvoří novou instanci stavu. Porovnání práce se stavem bez a s knihovnou Immer lze vidět v Kódu 4.1.

Další důležitou použitou knihovnou je class-transformer [28]. V jazyce JavaScript, resp. v TypeScript, existují dva druhy objektů – *plain* a *class* objekt. Při serializaci class

```

1 // tvorba stavu v React
2 const [state, setState] = useState({
3   person: {
4     name: "Name",
5     age: 30,
6     inventory: {
7       description: "Description",
8       productIds: [1,2,3]
9     }
10  }
11 });
12
13 // aktualizace stavu bez knihovny Immer
14 setState(prevState => ({
15   ...prevState,
16   person: {
17     ...prevState.person,
18     inventory: {
19       ...prevState.person.inventory,
20       productIds: [...prevState.person.inventory.productIds, 4]
21     }
22   }
23 })))
24
25 // aktualizace stavu s knihovnou Immer
26 setState(produce(draft => {
27   draft.person.inventory.productIds.push(4)
28 })))

```

Kód 4.1: Použití knihovny Immer

objektů do JSON [15] a zpět ztrácí objekt své metody, metody předků a další informace o třídě, ze které byla zkonstruována jeho instance. Knihovna class-transformer umožňuje mezi těmito typy objektů převádět a my tak můžeme používat instanční metody na objektech deserializovaných z JSON. Knihovna nefunguje perfektně pro všechny druhy datových složek a pokládá na třídy další určité restriktce, které už jsme zmínili v Sekci 3.5. Dále knihovna nepodporuje generické typy, protože TypeScript nemá reflexi za běhu (v době psaní práce se na ní však pracuje). Naštěstí jsou výjimkami tohoto omezení zabudované generické typy `Set<T>` a `Map<T>`, se kterými v datovém modelu pracujeme.

Pro zjednodušení práce s CSS styly používáme knihovnu Tailwind CSS [29]. Tailwind CSS proráží metodu nastavování stylů pomocí tzv. utilitních tříd. Velmi dobře se kombinuje s frameworkem React, protože místo psaní stylů do CSS souborů s arbitrárními názvy, Tailwind CSS umožňuje styl komponent upravovat přímo v jejich atributu `class`. Například, pokud chceme nastavit okraje ovládacího prvku na 24 pixelů, použijeme třídu `m-6`, kde `m` je zkratka ze slova `margin` (okraj) a `6` znamená „šestá úroveň“. Úrovně různých nastavení mají předdefinované (ale nastavitelné) hodnoty a umožňují konzistentní styl – délky, barvy i celé kombinace několika vlastností. Při kompilaci aplikace se ve výsledném balíčku CSS zakomponují z Tailwind CSS pouze třídy, které byly v aplikaci opravdu použity.

Pro kompilaci a vytvoření výsledného balíčku statické webové aplikace používáme nástroj Vite (čte se francouzsky, [vít]). Výstupem této kompilace je ideálně pouze

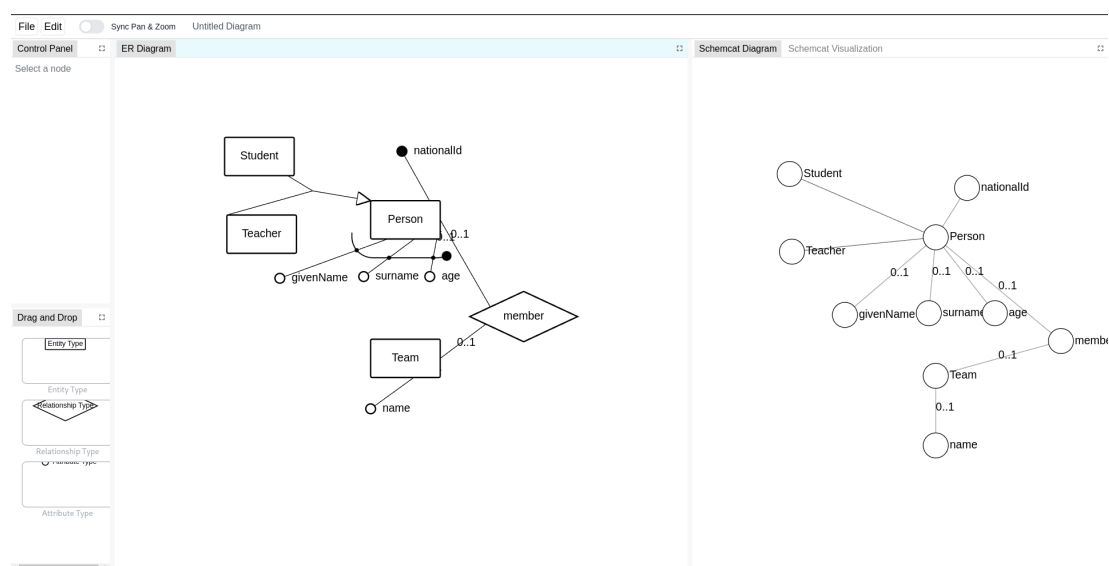
jediný HTML soubor, jediný JavaScript soubor a jediný CSS soubor, které jsou připraveny na nasazení. Vite totiž přeloží TypeScript do JavaScriptu, najde a propojí všechny importované moduly a spojí vše do jediného souboru.

Stěžejní knihovnou, která poskytuje prvky uživatelského rozhraní, je FlexLayout [30]. Jedná se o rozhraní s okny, která se dají přesouvat, roztahovat a přepínat záložkami. Tato knihovna byla zvolena, protože je flexibilní a uživatel výsledné aplikace si své pracovní prostředí díky jejím prvkům může libovolně upravit.

K tomu, abychom uložili projekt do PNG souboru tak, aby šel znovu plnohodnotně otevřít, jsme využili toho, že specifikace PNG umožňuje do souboru přidat libovolné textové informace [31, sekce 11.3.4]. Specifikace k tomu definuje tři typy datových bloků. Pro práci s PNG bloky jsme použili knihovny png-chunks-extract [32] k extrahování bloků ze souboru, resp. png-chunks-encode [33] k zpětnému zakódování bloků. Dále jsme využili knihovnu png-chunk-text [34], která kóduje a dekóduje právě textové bloky. Všechny tři knihovny jsou od stejného autora a jsou spolu plně kompatibilní.

4.2 Uživatelská dokumentace

Pro potřeby uživatele popíšeme uživatelské rozhraní aplikace, jehož ukázkou lze vidět na Obrázku 4.1.



Obrázek 4.1: Ukázka uživatelského rozhraní aplikace

V horní liště aplikace se nachází menu. Tlačítko **File** obsahuje možnosti pro vytvoření nového projektu a pro export. Tlačítko **Edit** obsahuje možnosti **Undo** a **Redo**. Následuje přepínač „Sync Pan & Zoom“, který zamkne synchronizaci pozice pláten. Přetažení a přiblížení v jednom plátně tak způsobí totožný přesun a přiblížení ve všech ostatních plátních. Poslední položkou horní lišty je editovatelné textové pole s názvem diagramu. Tento název se odráží v názvu souboru při exportu diagramu.

Rozhraní aplikace se skládá z oken, která obsahují záložky s různým obsahem. Záložky lze přesouvat mezi okny a přetahovat přes jiné záložky myší, čímž může být vytvořeno nové okno. Při přetahování je uživateli intuitivně vizuálně znázorněno, kte-

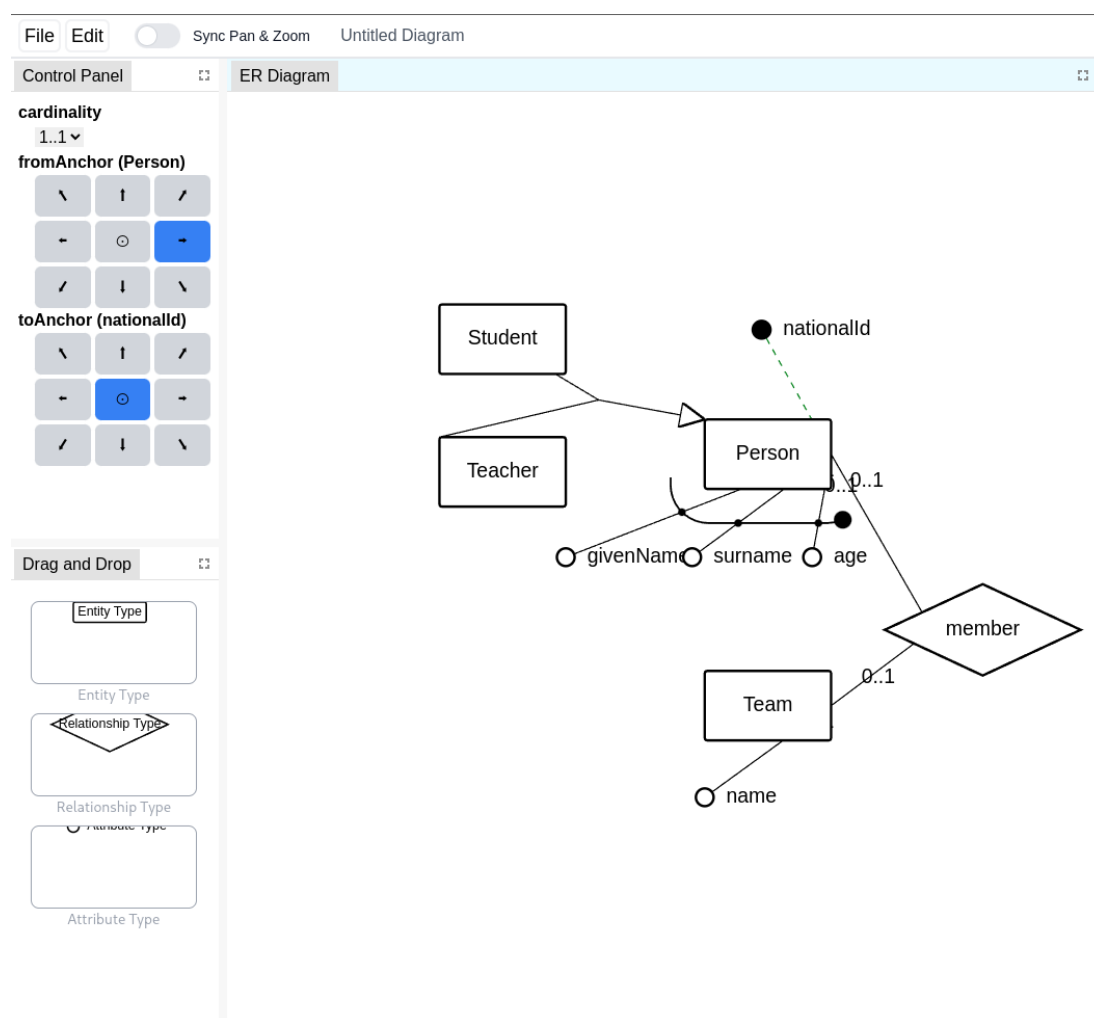
rá z akcí se stane po puštění záložky myši. Okna lze zvětšovat a zmenšovat přetažením jejich krajů. Pokud v nějakém okně nezůstane ani jedna záložka, okno zanikne.

Právě aktivní okno je zvýrazněno světle modrou barvou. Tato informace říká, kterým plátnem se bude řídit synchronizace pozic, a který diagram bude výchozí při exportování.

Záložka s názvem Control Panel obsahuje ovládací panel, ve kterém lze upravovat hodnoty jednotlivých vlastností elementů diagramu. Lze ho pozorovat na Obrázku 4.2, kde je zvoleno spojení, jehož kardinalitu a kotvy lze přenastavit v ovládacím panelu.

Záložka s popiskem Drag and Drop, která je také vidět na Obrázku 4.2 v levém dolním rohu, obsahuje konstrukty, které lze podržet a přesunout myši do plátna, čímž dojde k vytvoření daného konstruktů v daném plátně.

Význam záložek s popisky ER Diagram, Schemcat Diagram a Schemcat Visualization je zřejmý – obsahují popořadě plátna pro ER diagram, schematickou kategorii a vizualizaci schematické kategorie.



Obrázek 4.2: Ukázka ovládacího panelu

4.3 Technická dokumentace

Zdrojový kód aplikace je k dispozici v elektronické příloze A.1. Popíšeme zde některé důležité složky se zdrojovým kódem aplikace.

src › components obsahuje React komponenty. Podsložky seskupují více podobných komponent. Některé z nich uvedeme.

src › components › UserControls seskupuje komponenty základních uživatelských prvků – zaškrťovací pole, komponenta pro výběr barvy uživatelem, atd.

src › components › Menu seskupuje komponenty, které se týkají hlavního menu v horní liště aplikace a kontextového menu.

src › components › Dialog seskupuje komponenty dialogů pro export a potvrzování akcí.

src › hooks obsahuje React hooks, včetně důležitého `useStore.ts`, který obstarává instanci modelu s pomocí knihovny Zustand a jeho editaci pomocí knihovny Immer.

src › model obsahuje soubory, ve kterých se nacházejí jednotlivé třídy modelu.

src › utils obsahuje soubory s operacemi, které se týkají transformace textových řetězců, práce s poli, apod.

Vedle některých souborů, např. `Array.ts`, jsou umístěny i soubory se stejným názvem, pouze odlišnou příponou, např. `Array.test.ts`. Tyto soubory obsahují unit testy, které ověřují funkcionálnost některých operací.

Vstupním bodem aplikace je soubor `index.html`, který volá jediný skript – `src › main.tsx`. Úkolem tohoto skriptu je do DOM stromu připojit framework React a vložit do něj hlavní komponentu aplikace. Soubor obsahující hlavní komponentu aplikace je `src › App.tsx`. Tato komponenta se stará o přípravu uživatelského rozhraní za použití komponent z knihovny FlexLayout. Do nich poté nasazuje vlastní komponenty aplikace. Některé komponenty blíže popíšeme.

ControlPanel představuje ovládací panel, ve kterém uživatel mění hodnoty vlastností elementů diagramu. V modelu je pomocí dekorátorů (funkce jazyku TypeScript) určeno, který ovládací prvek má být použit pro kterou vlastnost třídy daného elementu. Tato informace je pak za běhu použita k použití správné komponenty. Jde o způsob reflexe, který je v TypeScriptu v době psaní práce experimentální (`reflect-metadata`).

AnchorPicker slouží pro výběr kotvy, tj. bodu, kde má být přichyceno spojení na elementu diagramu.

Draggable je obecná komponenta, která očekává děti a umožňuje jejich přetahování uživatelem.

ErNode představuje element ER diagramu, tedy buď entitní typ, vztahový typ, nebo atribut. Který tvar vykreslí, rozhodne pomocí hodnoty typu `ErNodeType` z modelu.

IdentifierFence je komponenta zodpovědná za vykreslování složených identifikátorů. Ty jsou vykresleny jako část obvodu obdélníku s oblými rohy, kterému říkáme *fence* (neboli *plot*). V pracovní verzi se místo toho používaly Beziérovky křivky. Výsledek však nebyl dostatečně estetický.

PannableZoomableSvg reprezentuje SVG plátno, které lze přesouvat, přibližovat a oddalovat myší. Dosahuje toho přepočítáváním SVG atributu `viewBox`.

Většina souborů obsahuje komentáře a TSDoc [35] dokumentaci.

4.4 Vývoj

K vývoji je potřeba mít nainstalovaný Node JS [36]. Naše prostředí má Node verze 20. Po instalaci by měl být k dispozici program `corepack`, kterým lze aktivovat náš upřednostňovaný správce balíčků `pnpm` [37]. Spustíme příkaz¹

```
corepack prepare --activate pnpm@^8
```

a poté ve složce `schemcat` se zdrojovým kódem spustíme

```
pnpm install
```

což nainstaluje všechny potřebné závislosti.

Poté lze používat všechny příkazy, které jsme definovali v souboru `package.json`. Např. příkaz

```
pnpm run dev
```

spustí vývojový server, který sleduje změny ve zdrojovém kódu a sám stránku webového prohlížeče aktualizuje, když se kód změní. To umožňuje rapidní cyklus vývoje a testování změn.

Příkaz

```
pnpm run build
```

sestaví projekt a výsledné soubory připravené k nasazení uloží do složky `dist`.

4.5 Instalace a nasazení

Kromě zdrojového kódu je v Příloze A.1 také sestavený projekt. Stačí, aby správce soubory přesunul a nasadil na statický webový server. Tento statický server však musí při odesílání souborů přes HTTP přiřazovat korektní MIME hlavičky, jinak není fungování projektu zaručeno.

Uživatel musí mít moderní webový prohlížeč, který podporuje ES6 moduly². Pokud je potřeba přidat podporu starších prohlížečů, je třeba postupovat podle Sekce 4.4. Dále nainstalovat balíček `@vitejs/plugin-legacy`³. Poté je třeba definovat v souboru `package.json` položku `browserslist` s rozsahem webových prohlížečů, které chceme podporovat podle specifikace⁴. Nakonec je nutné projekt sestavit.

Pokud chceme sestavený projekt spustit lokálně a máme nainstalovaný správce balíčků ze Sekce 4.4, stačí spustit příkaz

¹rozsah `^8` specifikuje verzi použitou v době psaní práce, nejnovější verzi lze nainstalovat tím, že místo tohoto rozsahu použijeme `latest`

²prohlížeče definované v tabulce na této adrese <https://caniuse.com/es6-module>

³<https://www.npmjs.com/package/@vitejs/plugin-legacy>

⁴<https://browserslist.org/>

```
pnp http-server .
```

ve složce se soubory sestaveného projektu. Na obrazovce se pak vypíše lokální adresy, na kterých běží tento statický server.

Závěr

Tato práce se zabývala využitím schematické kategorie [3] jako prostředku ke konceptuálnímu modelování s vyšší vyjadřovací schopností, než mají běžné modely, jako ER a UML. Jedná se o čerstvý koncept, který by mohl unifikovat různé databázové systémy a smazat rozdíly mezi konceptuální a logickou vrstvou. Konceptuální modelování pomocí schematických kategorií je obecnější než kterékoli existující řešení, která jsme analyzovali v této práci.

Za účelem přiblížení k realizaci tohoto ambiciózního cíle unifikace konceptuálního modelování byla vyvinuta webová aplikace, která umožňuje konceptuální modelování v ER a automatický převod schématu do schematické kategorie. Aplikace umožňuje interaktivně prozkoumávat schematickou kategorii a také přibližuje její strukturu pomocí navržené vizualizace schematické kategorie. Dovolí tak zkušeným databázovým inženýrům a softwarovým analytikům seznámit se se schematickou kategorií s pomocí něčeho, co už znají – ER modelu. Aplikace dále umožňuje výzkumníkům, kteří pracují se schematickou kategorií, rychlejší experimentování.

Při implementaci aplikace s pomocí zvolených technologií však došlo k mnoha problémům. Největším viníkem byla nejspíše volba frameworku React, která se před implementací zdála být vhodným kandidátem pro zrychlení vývoje jednostránkové moderní webové aplikace. Bohužel se při vývoji ukázalo, že React klade restriktce na model aplikace a dále například neumožňuje jednoduchý tok dat mezi sourozenci ve stromě webového dokumentu. Jelikož diagramy obsahují skoro výhradně komponenty, které jsou svými sourozenci, bylo obtížné vytvořit algoritmy, které kontrolují validitu diagramů apod. Dalším způsobeným problémem bylo, že React požaduje, aby všechny instance dat byly immutable. Úprava diagramu se tak ztížila.

Tyto a další problémy byly částečně vyřešeny knihovnamí pro React, které ale také měly své chyby. Například knihovna Zustand umožnila správu globálního stavu, který je k dispozici všem komponentám v aplikaci. Nicméně komponenty kvůli tomu musely obsahovat mnoho opakujícího se kódu, který zajistil reagování komponenty na změnu globálního stavu. React je více vhodný při práci ve větším týmu na dlouholetých projektech, kvůli svému zaměření na izolované komponenty. Autor práce neměl s frameworkem větší předchozí zkušenost a pro příště by volil jiný způsob implementace webové aplikace při samostatné práci. Například by se mohlo jednat o vanilla JavaScript, který umožňuje přímou mutaci modelu.

Zdlouhavým a problémovým procesem byl vývoj algoritmu, který vykresluje složené identifikátory. Při vývoji se prošlo dvěma odlišnými přístupy – Beziérovými křivkami a obdélníky s oblými rohy. U obou přístupů bylo obtížné najít polohu průsečíků křivky identifikátoru se spojeními, které křížuje, a posléze volba pořadí průsečíků pro vykreslení co nejkratší křivky. Algoritmus funguje stabilně, nicméně je určité prostor pro jeho vylepšení, co se týče stručnosti a výkonnosti.

Celkově byla aplikace a zvláště její model vyvinut s možností potenciálních rozšíření. Prostor pro rozšíření funkcionality aplikace spočívá například v přidání dalších známých prostředků k tvorbě konceptuálního modelu (např. UML), možnosti přímého modelování pomocí schematických kategorií, přidání distanční živé spolupráce, a nebo plnohodnotná validace korektnosti diagramů.

Seznam použité literatury

- [1] Peter Pin-Shan Chen. „The Entity-Relationship Model—Toward a Unified View of Data“. In: *ACM Transactions on Database Systems* 1.1 (č. 1, 1. 3. 1976), s. 9–36. ISSN: 0362-5915, 1557-4644. DOI: 10.1145/320434.320440.
- [2] Object Management Group (OMG). *Unified Modeling Language Specification Version 2.5.1*. Pros. 2017. URL: <https://www.omg.org/spec/UML/2.5.1>.
- [3] Martin Svoboda, Pavel Čontoš a Irena Holubová. „Categorical Modeling of Multi-model Data: One Model to Rule Them All“. In: *Model and Data Engineering*. Ed. Christian Attiogbé a Sadok Ben Yahia. Lecture Notes in Computer Science. Cham: Springer International Publishing, 14. 6. 2021, s. 190–198. ISBN: 978-3-030-78428-7. DOI: 10.1007/978-3-030-78428-7_15.
- [4] Samuel Eilenberg a Saunders MacLane. „General Theory of Natural Equivalences“. In: *Transactions of the American Mathematical Society* 58.0 (1. 8. 1945), s. 231–294. ISSN: 0002-9947, 1088-6850. DOI: 10.1090/S0002-9947-1945-0013131-6.
- [5] *DrawIo*. URL: <https://www.drawio.com/> (cit. 23. 7. 2021).
- [6] *DrawSQL – Database Schema Diagrams*. DrawSQL. URL: <https://drawsql.app> (cit. 21. 7. 2023).
- [7] *ERDPlus*. URL: <https://erdplus.com/> (cit. 24. 7. 2021).
- [8] *Nomnoml*. URL: <https://www.nomnoml.com> (cit. 25. 7. 2021).
- [9] *Visual Paradigm Online - Suite of Powerful Tools*. URL: <https://online.visual-paradigm.com/> (cit. 26. 9. 2022).
- [10] Bogdan Brinza et al. *Scalable Vector Graphics (SVG) 2*. Candidate recommendation. W3C, 4. 10. 2018. URL: <https://www.w3.org/TR/2018/CR-SVG2-20181004/>.
- [11] *Jgraph/Drawio on GitHub – Open-source, Not Open-Contribution*. JGraph. URL: <https://github.com/jgraph/drawio/blob/c7122cad617f52563c11d90890e64ab06db3a27a/README.md#open-source-not-open-contribution> (cit. 4. 8. 2022).
- [12] OpenJS Foundation. *Build Cross-Platform Desktop Apps with JavaScript, HTML, and CSS | Electron*. URL: <https://electronjs.org/> (cit. 10. 6. 2023).
- [13] MathJax Consortium. *MathJax*. MathJax. URL: <https://www.mathjax.org/> (cit. 2. 8. 2022).
- [14] Martin Seibert. *Extracting the XML from Mxfiles*. draw.io. 3. 8. 2016. URL: <https://drawio-app.com/extracting-the-xml-from-mxfiles/>.
- [15] TC39 Group. *The JSON Data Interchange Syntax 2nd Edition*. Standard ECMA-404. Rue du Rhône 114, CH-1204 Ženeva, Švýcarsko: ECMA International, pros. 2017. URL: <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>.
- [16] Meta Open Source. *React – The Library for Web and Native User Interfaces*. URL: <https://react.dev/> (cit. 15. 6. 2023).

- [17] Paolo Atzeni et al. *Database Systems: Concepts, Languages & Architectures*. New York: McGraw-Hill, 1999. 612 s. ISBN: 978-0-07-709500-0.
- [18] „Interim Report: ANSI/X3/SPARC Study Group on Data Base Management Systems 75-02-08“. In: *Bulletin of ACM SIGMOD* 7.2 (1975). Ed. Thomas B. Steel Jr., s. 1–140. URL: <http://portal.acm.org/toc.cfm?id=984332>.
- [19] Andrew Cron, Huy L. Nguyen a Aditya Parameswaran. „Big Data“. In: *XRDS: Crossroads, The ACM Magazine for Students* 19.1 (zář. 2012), s. 7–8. ISSN: 1528-4972, 1528-4980. DOI: 10.1145/2331042.2331045.
- [20] Ian Sommerville. *Software Engineering*. 9th ed. Boston: Pearson, 2011. 773 s. ISBN: 978-0-13-703515-1 978-0-13-705346-9.
- [21] Microsoft. *TypeScript – JavaScript With Syntax For Types*. URL: <https://www.typescriptlang.org/> (cit. 13.6.2023).
- [22] Stack Overflow. *Developer Survey 2022*. Stack Overflow. URL: <https://survey.stackoverflow.co/2022/#section-most-popular-technologies-web-frameworks-and-technologies> (cit. 13.6.2023).
- [23] Michel Weststrate. *Immer*. immer. URL: <https://github.com/immerjs/immer> (cit. 14.6.2023).
- [24] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, Mass: Addison-Wesley, 1995. 395 s. ISBN: 978-0-201-63361-0.
- [25] Gunnar Övergaard a Karin Palmkvist. *Use Cases: Patterns and Blueprints*. Software Patterns Series. Indianapolis, Ind: Addison-Wesley, 2005. 434 s. ISBN: 978-0-13-145134-6.
- [26] Daishi Kato. *Zustand*. Poimandres. URL: <https://github.com/pmndrs/zustand> (cit. 14.6.2023).
- [27] Charles Kornoelje. *Zundo*. URL: <https://github.com/charkour/zundo> (cit. 14.6.2023).
- [28] Attila Oláh. *Class-Transformer*. TypeStack. URL: <https://github.com/typestack/class-transformer> (cit. 14.6.2023).
- [29] Tailwind Labs. *Tailwind CSS - Rapidly Build Modern Websites without Ever Leaving Your HTML*. 15. 11. 2020. URL: <https://tailwindcss.com/>.
- [30] *FlexLayout*. Caplin. URL: <https://github.com/caplin/FlexLayout> (cit. 14.6.2023).
- [31] W3C. *Portable Network Graphics (PNG) Specification (Second Edition)*. 2003. URL: <https://www.w3.org/TR/2003/REC-PNG-20031110/>.
- [32] Hugh Kennedy. *Png-Chunks-Extract*. URL: <https://github.com/hughsk/png-chunks-extract> (cit. 12.7.2023).
- [33] Hugh Kennedy. *Png-Chunks-Encode*. URL: <https://github.com/hughsk/png-chunks-encode> (cit. 12.7.2023).
- [34] Hugh Kennedy. *Png-Chunk-Text*. URL: <https://github.com/hughsk/png-chunk-text> (cit. 12.7.2023).
- [35] Microsoft. *What Is TSDoc?* URL: <https://tsdoc.org/> (cit. 5.7.2023).

- [36] OpenJS Foundation. *Node.js*. Node.js. URL: <https://nodejs.org/en> (cit. 14. 6. 2023).
- [37] pnpm. *Pnpm – Fast, Disk Space Efficient Package Manager*. URL: <https://pnpm.io/> (cit. 5. 7. 2023).

Seznam obrázků

1.1	Tvorba ER diagramu v aplikaci diagrams.net	8
1.2	Tvorba diagramu v drawSQL	12
1.3	Tvorba ER diagramu v ERDplus	13
1.4	Tvorba UML diagramu v nomnoml	15
1.5	Tvorba UML diagramu ve Visual Paradigm Online	17
2.1	Ukázka redundantního a neredundantního identifikátoru	22
2.2	Zřetězení externích identifikátorů	22
2.3	Příklad kategorie	25
2.4	Signatury morfismů, \cdot je operace konkatenace (zřetězování) symbolů	26
2.5	Příklad schematické kategorie	27
2.6	Příklad vizualizace schematické kategorie	28
2.7	Vizualizace schematické kategorie s využitými složkami <i>uspořádání a duplicity</i>	28
3.1	Konceptuální schéma – ER diagram	34
3.2	Konceptuální schéma – schematická kategorie	35
3.3	Konceptuální schéma – Vizualizace schematické kategorie	36
3.4	Ukázka problému immutability a referencí	42
3.5	Diagram tříd – diagram	43
3.6	Diagram tříd – ER	44
3.7	Diagram tříd – schematická kategorie	45
3.8	Diagram tříd – Vector2	46
3.9	Diagram tříd – Angle	46
3.10	Pravotočivá a levotočivá kartézská soustava souřadnic	47
3.11	Diagram tříd – GlobalIdGenerator	47
3.12	Diagram tříd – SvgPathStringBuilder	47
3.13	Diagram případů užití	48
4.1	Ukázka uživatelského rozhraní aplikace	53
4.2	Ukázka ovládacího panelu	54

Seznam tabulek

1.1	Vyhodnocení srovnávacích kritérií pro nástroj diagrams.net	7
1.2	Vyhodnocení srovnávacích kritérií pro nástroj drawSQL	11
1.3	Vyhodnocení srovnávacích kritérií pro nástroj ERDPlus	13
1.4	Vyhodnocení srovnávacích kritérií pro nástroj nomnoml	14
1.5	Vyhodnocení srovnávacích kritérií pro nástroj Visual Paradigm Online	16
1.6	Srovnání existujících řešení	18
1.7	Exportované formáty existujících řešení	19
1.8	Úložiště existujících řešení	19
2.1	Grafická reprezentace konstruktů ER modelu	20

A. Přílohy

A.1 Elektronické přílohy

Součástí této práce jsou elektronické přílohy v archivu ZIP:

složka schemcat obsahuje zdrojový kód aplikace,

projekt.svg je soubor s projektem otevíratelný ve vytvořené aplikaci.