Assignment of master's thesis

Title:	ClueMaker - visual configuration
Student:	Bc. Dávid Žalúdek
Supervisor:	Ing. Marek Sušický
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2021/2022

Instructions

The main goal of this thesis is to analyze, design, implement and test a new module into an existing application ClueMaker Configurator. The application is used to create and edit configuration which contains definitions of entities, relations between them and their mapping from data sources. The new configuration application module is intended to simplify and partially automate the creation and modification of the configuration and will enable:
- visualization of defined entities and links between them,
- searching for and defining probable entities and relations in specified data sources (when creating a new configuration or modifying an existing configuration), e.g. using a wizard,
- finding and defining a related entity (or link) to an existing entity in the specified data sources,
- finding and defining a related entity (or link) to an existing entity based on the match of the name and data type between the tables in the data source.

Electronically approved by Ing. Michal Valenta, Ph.D. on 13 October 2020 in Prague.

Master's thesis

ClueMaker - visual configuration

Bc. Dávid Žalúdek

Department of Software Engineering
Supervisor: Ing. Marek Sušický

May 6, 2021

Acknowledgements

My thanks go to my supervisor Ing. Marek Sušický, for his insights and guidance to the company Profinit EU s r.o for providing me the opportunity to collaborate on the ClueMaker project. Lastly, to my family, friends, and girlfriend, without their moral support and encouragement, I wouldn't be able to endure the years of studying CTU.

Declaration

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

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 6, 2021 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Dávid Žalúdek. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Žalúdek, Dávid. *ClueMaker - visual configuration.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Táto diplomová práca sa zaoberá analýzou, návrhom a realizáciou modulov do aplikácie ClueMaker Configurator firmy Profinit EU s.r.o., ktorej hlavnou úlohou je generovanie konfigurácie pre vizualizačný nástroj ClueMaker. Implementované moduly majú za úlohu pomáhať pri vytváraní, modifikácií a vizualizácií konfigurácie.

Klíčová slova ClueMaker, vizualizácia dát, analýza dát, konfigurácia, SQL, databáza, graf

Abstract

This thesis deals with the analysis, design, and implementation of new modules to Profinit EU ClueMaker Configurator, whose primary function is to create configuration containing data source mapping to entity model used in visualization done by Profinit EU ClueMaker. These modules aid in the creation, modification, and visualization of the configuration.

Keywords ClueMaker, data visualization, data analysis, configuration, SQL, database, graph

Contents

Introduction **1**
Motivation and goal of the thesis . . . 1
Thesis structure . . . 2

1 Introduction to ClueMaker **3**
1.1 Basic Terms . . . 3
1.2 ClueMaker configurator . . . 4
1.2.1 Supported data source types . . . 5
1.2.2 Generating configuration . . . 5
1.3 ClueMaker application . . . 5
1.3.1 Data import . . . 6
1.3.2 Data filtering . . . 6
1.3.3 Data visualization . . . 6
1.3.4 Useful features . . . 6
1.3.4.1 Pallete . . . 7
1.3.4.2 Exporting reports . . . 7
1.3.4.3 Timeline . . . 7
1.3.4.4 GIS . . . 8
1.3.4.5 Project export . . . 8
1.4 ClueMaker Server . . . 8
1.5 Used technologies . . . 8
1.5.1 Java . . . 8
1.5.2 Apache Netbeans platform . . . 9
1.5.2.1 Netbeans Visual library API . . . 9
1.5.3 Apache Maven . . . 9
1.6 Real-world deployment . . . 10
1.6.1 Investigative journalists . . . 11
1.6.2 Law firms . . . 11

2 Analysis and design **13**
2.1 Competing products . . . 13
2.1.1 IBM i2 Analyst's Notebook . . . 13
2.1.2 Cambridge Intelligence KeyLines . . . 14
2.1.3 Tovek Tools . . . 14
2.1.4 Maltego . . . 15
2.1.5 Summary . . . 15
2.2 Existing solution . . . 15
2.2.1 Creating configuration . . . 15
2.2.1.1 User prerequisites . . . 16
2.2.1.2 Step needed to create configuration . . . 16
2.2.1.3 SQL Query generation . . . 16
2.2.1.4 Creating links . . . 16
2.2.2 Visualization . . . 17
2.3 Requirements and use cases . . . 17
2.3.1 Use cases . . . 17
2.3.1.1 Use Case - creating configuration . . . 17
2.3.1.2 Use Case - edit configuration . . . 19
2.3.2 Functional requirements . . . 19
2.3.2.1 FR1 - Visualize configuration entities and relations in a graph. . . . 20
2.3.2.2 FR2 - Extract meta data from data sources . . . 20
2.3.2.3 FR3 - Find entities in data sources . . . 20
2.3.2.4 FR4 - Find relations between entities . . . 20
2.3.2.5 FR5 - Save data source metadata . . . 20
2.3.2.6 FR6 - Visually generate SQL query . . . 20
2.3.3 Non-functional requirements . . . 20
2.3.3.1 NR1 - Cross-platform deployment . . . 21
2.3.3.2 NR2 - Extensibility . . . 21
2.3.3.3 NR3 - GUI consistency . . . 21
2.3.3.4 NR4 - Backwards compatibility . . . 21
2.4 Configuration model . . . 21
2.4.1 Transaction design pattern . . . 21
2.4.2 Serialization/Deserialization . . . 22
2.5 ClueMaker configurator . . . 22
2.5.1 Configuration editor module . . . 22
2.5.1.1 Presenters . . . 22
2.5.1.2 View . . . 22
2.5.1.3 Actions . . . 23
2.5.2 Datasource model analysis . . . 23
2.5.3 Drxf library . . . 24
2.5.3.1 Functions . . . 24
2.5.3.2 Data model . . . 25
2.6 User inteface design . . . 25

2.6.1 Wizard 25
2.6.2 Configuration visualization 27
2.6.3 Visual SQL Query generation 27

3 Implementation **29**
3.1 Used libraries 29
3.1.1 Jackson 29
3.1.2 JSOG for jackson 29
3.1.3 Visual library API 29
3.2 Implemented Modules 30
3.2.1 Data sources meta data 30
3.2.1.1 Configuration model extension 30
3.2.1.2 Serialization/Deserialization 31
3.2.1.3 Datasources analyzer 32
3.2.1.4 Database analyzer 33
3.2.1.5 Drxf library integration 34
3.2.2 Configuration graph visualization 34
3.2.2.1 Model 35
3.2.2.2 Presenters 35
3.2.2.3 Contollers 36
3.2.2.4 Interactors 36
3.2.2.5 ViewModel 36
3.2.2.6 View 38
3.3 Configuration wizard 39
3.3.1 Presenters 40
3.3.2 View 40
3.3.3 Dialogs 41
3.4 SQLQuery builder 41
3.5 SQL query generation 43
3.5.1 Creating queries 43
3.6 Additional components 43
3.6.1 Searchable list 43
3.6.2 Create Node entity dialog 44
3.6.3 Create Relation dialog 44
3.6.4 Create Relation entities dialog 44
3.7 Additional changes 44
3.8 Evaluation of requirements 44
3.8.1 FR1 - Visualize configuration entities and relations as a graph. 44
3.8.2 FR2 - Extract metadata from data sources 44
3.8.3 FR3 - Find entities in meta data 45
3.8.4 FR4 - Find relations between entities 45
3.8.5 FR5 - Save data source metadata 45
3.8.6 FR6 - Visually generate SQL query 45

3.8.7 NR1 - Cross-platform deployment . . . 45
3.8.8 NR2 - Extensibility . . . 45
3.8.9 NR3 - GUI consistency . . . 45
3.8.10 NR4 - Backwards compatibility . . . 45

4 Testing **47**
4.1 Unit tests . . . 47
4.1.1 Unit testing Drxf databases analyzer module . . . 48
4.1.2 Unit testing configuration visualization module . . . 48
4.1.2.1 Additional tests . . . 48
4.2 Heuristical analysis . . . 48
4.3 Cognitive walk through . . . 50
4.3.1 Graph visualization . . . 50
4.3.2 Wizard . . . 51
4.4 Usability testing . . . 53
4.4.1 Tasks . . . 53
4.4.2 User 1 - Advanced user . . . 54
4.4.2.1 Task 1 . . . 54
4.4.2.2 Task 2 . . . 54
4.4.3 User 2,3 - Novice users . . . 54
4.4.3.1 Task 1 . . . 54
4.4.3.2 Task 2 . . . 55

Conclusion **57**

Bibliography **59**

A Acronyms **63**

B Contents of enclosed CD **65**

List of Figures

1.1 Cluemaker application logo [1] 3
1.2 Cluemaker configurator window. 4
1.3 Cluemaker timeline visualisation and graph view [2] 7
1.4 Apache Netbeans logo. [3] 10
1.5 Apache Maven logo. [4] 10

2.1 I2 group logo prior to being acquired by IBM. [5] 14
2.2 SQL text editor in ClueMaker configurator. 16
2.3 Element of a tree view with complex configuration opened. 17
2.4 Use case diagram for ClueMaker configuration process. 18
2.5 Class diagram of presenters handling components of configuration model. 23
2.6 Data model generated by analyzer library. 25
2.7 Configurator main window with opened entity editor and mapping. 26
2.8 Modal dialog used to create new entities. 27
2.9 Configuration visualization as a graph. 28
2.10 Visual SQL builder. 28

3.1 Class diagram of meta data structure. 31
3.2 Class diagram depicting composition of presenter classes. 35
3.3 Class diagram visualizing dependencies of view model implementation. 37
3.4 Visualization of entities and relations in graph using orthogonal routing for edges. 39
3.5 Visualization of entities and relations in graph using direct routing for edges. 40
3.6 Wizard panel for relation entities creation. 41
3.7 Visual SQL builder view. 42

List of Tables

2.1 Functional requirements 19
2.2 Non-functional requirements 20

Introduction

Never before in the history of the world has so much data been produced in such a short time. Most organizations now understand that if they capture all the data streams into their businesses, they can extract valuable information. To solve the problem of visualization of relations, links, and flows between subjects saved in multiple data sources, Profinit EU developed a commercial solution ClueMaker.

This application is a commercial product of Profinit EU s.r.o., of which I am an employee.

Motivation and goal of the thesis

With the ever-changing landscape of commercial applications, there is a need for ClueMaker to stay competitive within its respective field. Therefore, continuous development and improvement are a necessary part of the application life cycle.

To make the process of creating and managing configuration within the application more accessible to the less experienced users, we decided to implement new ways to create, edit and visualize configuration for the ClueMaker application. The current solution to extract entities and relations from database data sources relies on previous user knowledge of database-specific query language and database structure; this creates a barrier of entry for new potential customers. The proposed solution is the new wizard module which, in conjunction with our in-house developed library for database table relations detection, aims to simplify and partially automate the configuration's creation and modification.

This project's overall goal is to create a more readable representation of the current configuration and ease its creation.

Thesis structure

The work is divided into four logical units. The first part will describe the application, technologies used, and its functionality before the development.

The next part deals with analyzing competing solutions and designing a new approach to creating the configuration. This section also analyzes functional and non-functional requirements for the new module and wire-frame design.

The third section deals with the implementation of the new module. In addition to implementation details of individual functional requirements and a description of the technologies and procedures used, there is a description of the integration of the new modules with the rest of the application. Finally, at the end of this section is a brief evaluation of functional and non-functional requirements.

The last part describes how the new module was tested, i.e., its functionality using unit tests, user interface heuristic analysis, and user testing.

Chapter 1

Introduction to ClueMaker

ClueMaker (link www.cluemaker.com) is a visual analytics tool created for one of the premier Czech banks in 2012. It provides user-friendly querying across connected data sources and a clear display of results, including time contexts. Furthermore, with the help of a clear graphic connection, it is possible to search for more related facts, making it easier, for example, for a security analyst to form an overall picture of the case under investigation.

ClueMaker is divided into two smaller applications, the first responsible for displaying data and operations on them. The second is the configurator, which manages data sources and the mapping of individual entity attributes. The main goal of the thesis is to streamline and simplify configuration creation.

1.1 Basic Terms

Introduction to most common terms used when describing ClueMakers configuration and model. [6]

- Attribute: A data source field such as the surname, date, bank account number.
- Entity: In general, a set of information related to an object, for example, a person, company, contract, bank account. In the graph, the entities

Figure 1.1: Cluemaker application logo [1]

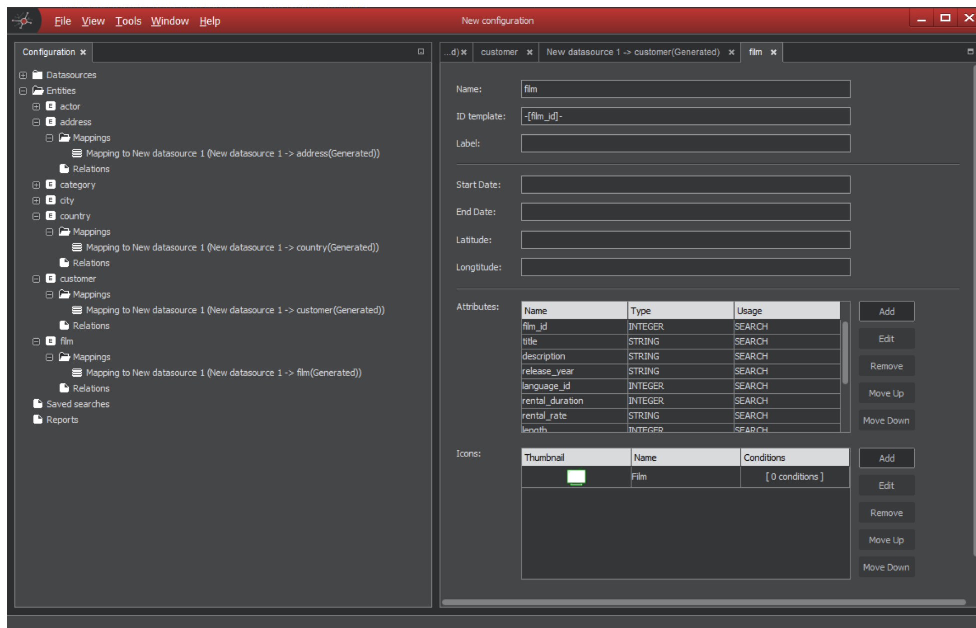

Figure 1.2: Cluemaker configurator window.

are represented as nodes and specific types of links.

- Relational entity: Item describes a relation that contains attributes, for example, transaction or contract.
- Node: A fundamental component representing a data entry in a graph (person, company, account) for which we search for links to its neighbors.
- Link/Edge: An expression for connection or relationship between two nodes. It is not necessarily only a technical connection; the link could bear several attributes, similarly to the node. (A good example is the bank transaction - the link connecting two accounts that also contains attributes such as the amount, currency, date)

1.2 ClueMaker configurator

The standalone application is responsible for defining data sources and their respective mapping to user-defined entities, creating relations, defining rules for generating reports. [1.2]

The output of this program is ClueMaker Workspace (.sws) file. This file contains all definitions and resources necessary to import and visualize data from data sources.

1.2.1 Supported data source types

ClueMaker supports a plethora of traditional relational databases (Oracle, PostgreSQL, Microsoft SQL Server, and MySQL) and less traditional database systems like Firebird, Impala, Teradata, Apache Hive, Aster Data, IBM DB2, Netezza a Splunk. This adds flexibility to our users and allows them to define entities that are spread across different data sources. Lastly, we support import from Excel using the wizard, which aids during the entire process of creating the mapping, entities, and their relations.

1.2.2 Generating configuration

The main feature of this application is to generate configuration subsequently used by the main ClueMaker application. The current implementation relies on a series of form windows that allow the user to edit the configuration file and create complex mappings to data sources; this is the functionality I want to extend in this thesis since users are not provided any guidance other than the user manual.

Currently, the application contains these main form panels:

- Data source editor - this panel provides the user with a form to create a new data source and define parameters needed during the import step
- Entity editor - this panel provides the user with specifying attributes of extracted entities.
- Relation entity editor - extends entity editor panel with source and target relation definition
- Mapping editor - form defining a mapping from data source to the selected entity and its attributes. When dealing with an SQL database, this panel contains a formatted text field used to create a query.
- Relation editor panel - form defining the relation between two defined entities.
- Saved searches - provides the user with functionality to create and save frequently run queries on the data set.
- Reports editor - Provides the user with the ability to define templates for generating reports.

1.3 ClueMaker application

The main application which visualizes graph from imported data based on configuration generated by ClueMaker configurator.

1.3.1 Data import

The configuration defines the mapping from data sources to entities based on which import queries are generated; this functionality provides the user ability to query data sources for entities and use provided filters to focus on investigated nodes. Results of queries are presented to the user in the form of a table from which he can select subjects of interest. Another way to import nodes to graphs is to expand nodes; this functionality also allows the user to specify edges (relations) to expand.

1.3.2 Data filtering

After importing data, the user can create filters based on attribute values or edge types, or entity types and highlight nodes in a graph or table. Another way to filter data is to use the full-text search functionality, which looks throughout the entire graph to find a matching value. These filters can be combined at will and saved for later use.

1.3.3 Data visualization

ClueMaker displays data in two ways, either as a graph or in the form of a table. The graph represents entities as nodes and relations as edges. Table maps entities to rows and their respective attributes as columns. Attributes are dynamically loaded from data sources and are viewable as tables after expanding nodes and edges.

ClueMaker offers multiple layouts to detangle complex networks and help uncover insight. Each arrangement tries to highlight different features of data and provide a new perspective. Supported layouts:

- Organic Layout - result in a natural distribution of nodes that exhibits clusters and symmetric properties of the graph.
- Hierarchical Layout - the nodes are distributed into layers based on degree. The order of the nodes within the layers ensures that the number of edge crossings is as small as possible. Horizontal and vertical hierarchy is supported.
- Timeseries Layout - Each node and edge has optional properties defining its valid time, based on which ClueMaker can organize them into time series layout.

1.3.4 Useful features

In this section, the author will highlight a few of the implemented tools aiding in data visualization.

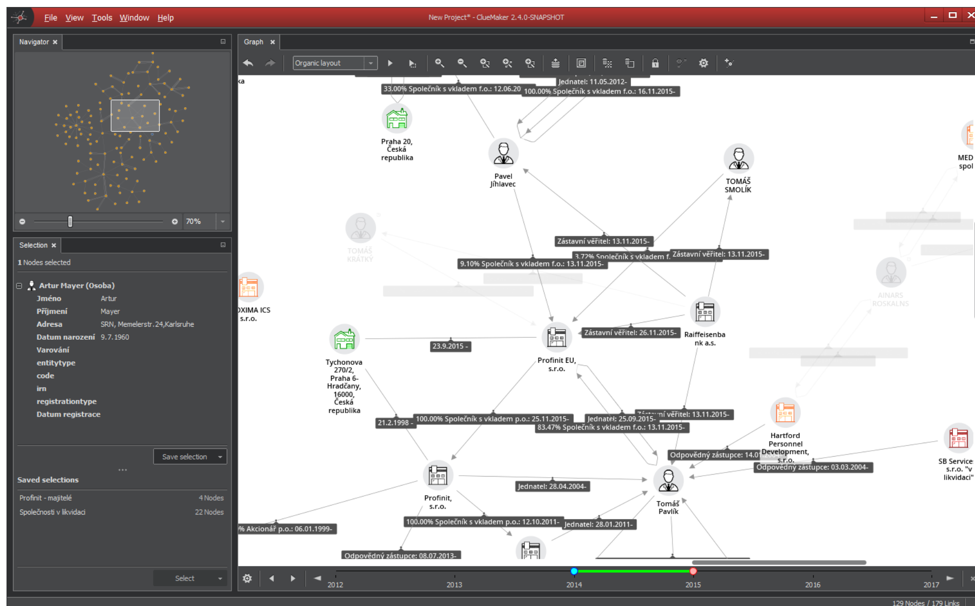

Figure 1.3: Cluemaker timeline visualisation and graph view [2]

1.3.4.1 Pallete

To ease visualization application provides a toolkit in the form of a palette, which provides a way to place custom nodes, pictures, text, and edges. This visual aid form helps organize complex structures and fill in pieces of information not available in configured data sources.

1.3.4.2 Exporting reports

Another helpful feature to simplify and speed up repetitive lookups is the creation of reports. These can be run after an update to provide fast insight into what is happening with the data or speed up the targeted nodes' analysis. Reports provide the following functionality: to execute parameterized data queries on the entire data set or by creating reports specific to the selected entity; this approach allows the use of entity attributes and user-defined parameters. In addition, the report's result can be exported in the form of a table (.csv or .xls).

1.3.4.3 Timeline

Another convenient feature is the timeline [1.3] which relies on entity property valid time and allows users to scroll around to a specific time and evaluate data from this perspective.

1.3.4.4 GIS

This system allows for geospatial analysis of data from the database according to their geographic location. In addition, this module also implements advanced filtering methods based on location data.

1.3.4.5 Project export

ClueMaker has the functionality to export the entire project, including the current graph layout and selection as a ClueMaker Project file (.spr). This feature allows continuous work even without online access to data sources.

1.4 ClueMaker Server

Extends ClueMaker with web scraper and web crawler features like scraping from social networks, forums, RSS feeds, and searching the web for interesting data like transactions or social connections.

1.5 Used technologies

ClueMaker is a Java language-based application developed on Netbeans open IDE platform with Maven as a package management system. This combination creates an extendable, maintainable, and multiplatform architecture for modern tools.

1.5.1 Java

Java is an object-oriented programming language created by James Gosling from Sun Microsystems (Sun) in 1991 [7]; its main focus is to write a single implementation translate it into bite code (.dex format) which is then run on Java Virtual Machine.

The Java language was designed with the following characteristics as main goals:

- Platform independent: Java programs use the Java virtual machine as abstraction and do not access the operating system directly. This makes ClueMaker highly portable and available on all supported platforms, e.g., Windows, Linux, and Mac.
- Object-orientated programming language: Except for the primitive data types, all Java elements are objects.
- Strongly-typed programming language: It is possible to define the type of each variable during compilation.

- Interpreted and compiled language: Java source code is transferred into the bytecode format, which does not depend on the target platform. These bytecode instructions are interpreted by the Java Virtual machine (JVM).
- Automatic memory management: Java manages the memory allocation and de-allocation for creating new objects. The program does not have direct access to the memory. The garbage collector automatically destroys objects to which no active pointer exists.

Java 8 version [8] is used throughout the entire implementation of ClueMaker, including some of the modern features introduced in Java 8 like Stream API and lambda functions.

1.5.2 Apache Netbeans platform

The NetBeans Platform is a broad Java framework for large desktop applications, like different IDEs or, in this case, visualization toolkit ClueMaker. The NetBeans Platform contains APIs that simplify handling windows, actions, files, and many other things typical in applications. One of the best know applications that makes use of this platform is NetBeans IDE itself. [9]

Each distinct feature in a NetBeans Platform application is represented by a separate NetBeans module comparable to a plugin. A NetBeans module is a group of Java classes that provides an application with a specific feature. ClueMaker uses most of the provided features. Few examples include streamlining the handling of opened windows (TopComponent objects) and implementation of custom action shortcuts.

1.5.2.1 Netbeans Visual library API

The API provides a set of reusable pieces - widgets. By composing them, developers can create a visualization. Each widget has several properties, including layout, border, assigned actions, and many more.

The library contains a set of predefined widgets that can be extended. All pluggable pieces are declared as interfaces or abstract classes; a few examples include - WidgetAction, Anchor, AnchorShape, PointShape, Animator, Border, GraphLayout, LookFeel, Layout, SceneLayout, Router, CollisionsCollector. For all of these, the library provides built-in implementations which can be extended or modified to fit specific needs of the application[10]. Licensed under the Apache License, version 2.0.

1.5.3 Apache Maven

Maven is a powerful project management tool that is based on POM (project object model). It is used for project build, dependency, and documentation.

Figure 1.4: Apache Netbeans logo. [3]

ClueMaker takes full advantage of Maven, including the use of plugins handling testing like the surefire plugin or custom-defined builds using the jar plugin [11]. Licensed under the Apache License, version 2.0

The following are the key features of Maven in summary:

- Dependency management including automatic updating, dependency closures (also known as transitive dependencies).
- Multiple opened projects at the same time.
- Extensible, with the ability to quickly write plugins in Java or scripting languages. ClueMaker uses build plugins for the NetBeans platform.
- Model-based builds: Maven can build many projects into predefined output types such as a JAR, WAR, or distribution based on metadata about the project.
- Dependency management: Maven provides a central repository of JARs and other dependencies. Profinit EU. runs one instance on-premises where current implementation can be located.

Figure 1.5: Apache Maven logo. [4]

1.6 Real-world deployment

ClueMaker is currently used by several Czech banks, insurance companies, and telecommunication providers to uncover, investigate and prevent loan and insurance fraud.

In this section, the author will summarize the type of customers that are the main focus of this update to configuration creation and management. It

is mainly aimed at less experienced users that might not have a technical background but would greatly benefit from a new viewpoint on their data.

1.6.1 Investigative journalists

ClueMaker helps journalists visualize and uncover connections between people and find illegal money flows. One of such cases happened in 2018 [12] where an investigative journalist from Czech news site Hlídací pes uncovered undocumented money flow between high ranking officials and private companies that would not be possible without ClueMaker.

1.6.2 Law firms

Connecting data from sources like business, bankruptcy, and insolvency registers and companies' internal notes are essential to recognize fraud patterns among subjects, accounts, and money flows. In addition, it helps lawyers to scrutinize and unwrap suspicious claims and create reports used during trials.

Chapter 2

Analysis and design

This chapter will analyze the current process of creating and editing configuration in the ClueMaker configurator, analyzing the competing solution, and applying observations to the design process. The author will also explain the current solution's architecture and propose changes to the structure to make the app more flexible for future developments.

2.1 Competing products

The main focus of this section is the approach of selected competing products to creating configurations for their visualization tools.

2.1.1 IBM i2 Analyst's Notebook

After the acquisition of i2 by IBM in 2011 [13], this toolkit became the industry leader in data analysis and visualization. The toolkit is currently used by security agencies, law enforcement, banking sector all around the globe to detect crimes and fraud. IBM i2 Analyst's Notebook has vast visualization and analysis capabilities [14] with extensive documentation [15]. Functionality is split into several products :

- i2 iBridge - connects IBM i2 Analyst's Notebook users directly to enterprise databases with powerful search and query capabilities. This functionality is the most comparable to the capabilities of ClueMaker importer modules.

- i2 Charts - Charts display the intelligence that relates to an investigation. It can be created automatically by a toolkit or can be user-defined. Similar functionality to ClueMaker's saved searches and reports functionality.

- i2 TextChart - software for text extraction and visualization, which helps overcome problems associated with the processing of unstructured data.

Configuration creation The user creates import specifications using a wizard that recognizes some data source formats and guides the user during the configuration creation process. Import specification defines how the data is interpreted as entities and links or selects one of the provided templates. Analyst's Notebook inspects the data and lists any compatible import specifications; this allows to merge similar data sources into a single project definition.

Another useful feature is provided during the import step where similar entities are merged based on specified criteria; similar functionality is currently being implemented into ClueMaker.

Figure 2.1: I2 group logo prior to being acquired by IBM. [5]

2.1.2 Cambridge Intelligence KeyLines

JavaScript toolkit that produces graph visualization [16]. This approach has distinct advantages compared to ClueMaker; graphs are displayed in a web browser and are platform-independent, supports all data formats that can be loaded as JSON, this carries higher starting costs when wanting to use a private data source. Documented API is provided to ease the development of a custom solution for each deployment. At least basic knowledge of the JavaScript programming language is required.

2.1.3 Tovek Tools

Developed by Czech-based company Tovek that is a direct competitor to ClueMaker. Tovek Tools enable content exploration and analytical search with relationship visualization. Currently also supports capabilities that are to be implemented into ClueMaker like :

- explicitly defined database bindings or tables
- links based on the occurrence of the same automatically extracted objects (e.g., names)
- links based on the relevance of data content to simple or very complex queries

- links are given by the geographical location
- ties based on a time match or a match of another attribute

For more advanced use cases solution is tailor-made for customers, configuration and mapping are created by in-house developers, or trained professionals on a request basis [17].

2.1.4 Maltego

The Maltego application is a visual link analysis tool. The tool offers real-time data mining and information gathering and the representation of this information on a node-based graph. Maltego provides a data source store where data vendors sell already extracted data. It is possible to add a custom data source, but the internal developer team creates integration [18].

2.1.5 Summary

Solutions available on the market today usually require a deep knowledge of the used platform to get started. Although some like IBM i2 Analyst's Notebook provide the user with an entity and link detection on selected data sources, integration of similar feature to ClueMaker is one of our design goals.

2.2 Existing solution

In this section, the author will introduce the current approach to configuration creation and visualization. All actions involving the creation and removal of entities, data sources, mappings, and relations are handled from the main tree structure; with growing configuration, it is hard to keep track of relations between entities and their respective mappings.

2.2.1 Creating configuration

The current process of creating configuration is slightly cumbersome, mainly concerning defining entities and creating a mapping for SQL-based databases. For the advanced user, this does not pose much of a challenge since they generate queries outside of the configurator and paste them into the query text window, but with increasing access to open data, more and more people are keen to gain insight into their data.

Simplifying the entire process and streamlining it to the point that the users would connect to the database and would be able to create functioning configuration either through wizard or fully automatic process, which would be of excellent benefit to ClueMaker toolkit.

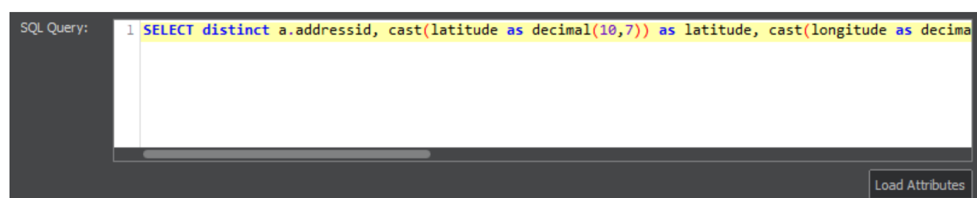

Figure 2.2: SQL text editor in ClueMaker configurator.

2.2.1.1 User prerequisites

Currently, ClueMaker Configurator is used by administrators who must have the information concerning the data sources they plan to use (such as the host and DB name, user, and password), including an understanding of the structure of data (a database schema in case of DB) in the data source they want to use for definition of entities. Basic knowledge of SQL is expected. [6]

The goal is to remove or significantly reduce the amount of time and required knowledge needed to create useful configuration so users can analyze data as soon as possible with added flexibility to have that detailed configuration option down the line.

2.2.1.2 Step needed to create configuration

Current workflow used to create configuration define in ClueMakers configuration guide [6]:

1. define one or more data sources
2. definitions of data entities and their mappings
3. definitions of relational entities and their mappings
4. define links between entities

2.2.1.3 SQL Query generation

Import of data from SQL database data source takes place through user-defined SQL query. Query creation currently requires writing SQL queries in a provided text editor [2.2] or outside of ClueMaker configurator; this solution is impractical for users who do not have access to database metadata or lack the knowledge of SQL language.

2.2.1.4 Creating links

Another part that needs some improvement is creating relations; this step in configuration creation relies on users' knowledge of the domain. One goal is to provide suggested relations between entities to the user based on foreign

Figure 2.3: Element of a tree view with complex configuration opened.

keys and matches of column names and types; this allows the user to select one of the suggested links, and configuration is automatically created.

2.2.2 Visualization

The primary way to display the structure of the currently opened configuration is a tree-like visualization. This type of view sacrifices the visibility of defined links between entities. As seen in figure 2.3, it is hard to keep track of relations between entities.

2.3 Requirements and use cases

After deliberation, these are requirements placed on the developed product extracted from discussed use cases and improvement ideas.

2.3.1 Use cases

The most typical use case for ClueMaker Configurator users is pretty straightforward and centered around creating the configuration file (.sws). The summary of the use case structure is represented in the use case diagram. 2.4

These use cases are later used during the design of usability tests.

2.3.1.1 Use Case - creating configuration

Actor - ClueMaker user Definition - Create a configuration for the ClueMaker application in the ClueMaker configurator.

2.3.1.1.1 Basic Flow Uses editor windows and menus.

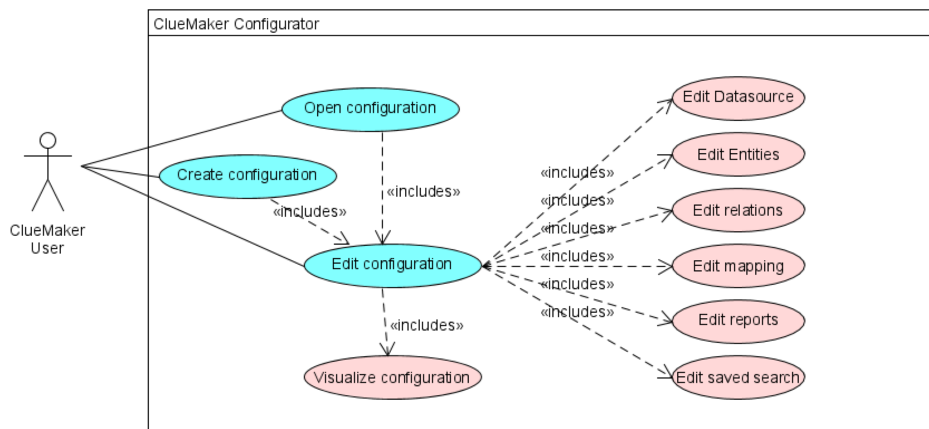

Figure 2.4: Use case diagram for ClueMaker configuration process.

1. Open ClueMaker Configurator application
2. Select new configuration option from menu file dropdown
3. Create and define connection settings for data source
4. Select analyzer from the dropdown menu and run analysis
5. Create new entities and define attributes
6. Create new relation entities and define attributes
7. Define relations between entities
8. Save configuration.

2.3.1.1.2 Proposed alternative flow - wizard Uses wizard to create a configuration. Guides users through various steps in configuration creation.

1. Open ClueMaker Configurator application
2. Open wizard window
3. Select new configuration
4. Select analyzer from a drop-down menu in the data source specification window
5. In next step create new entities or select from suggested
6. In next step create new relation entities or select from suggested

Code	Definition
FR1	Visualize configuration entities and relations in a graph.
FR2	Extract meta data from data sources.
FR3	Find entities in data sources.
FR4	Find relations between entities
FR5	Save data source meta data
FR6	Visually generate SQL query

Table 2.1: Functional requirements

7. In next step create new relations or select from suggested
8. Finish wizard
9. Save configuration

2.3.1.1.3 Proposed alternative flow - visualization Uses graph visualization to create a configuration. Allows the user to modify current visualization through the context menu in the graph and the toolbar.

1. Open ClueMaker Configurator application
2. Create a new configuration
3. Open configuration visualization window
4. Right-click anywhere on graph scene and open context menu, select new entity option, finish with new entity dialog repeat until required entities are added.
5. Select entities that analysis is to be applied. Select create relation or create relation entity. Choose from pre-filtered selection in dialog window or add new.
6. Save configuration

2.3.1.2 Use Case - edit configuration

It follows nearly identical steps to configuration creation, with the only difference that configuration is loaded from an existing file.

2.3.2 Functional requirements

Functional requirements define the basic system behavior and product features that focus on user needs.

Code	Definition
NR1	Cross-platform deployment
NR2	Extensibility
NR3	GUI consistency
NR4	Backwards compatibility

Table 2.2: Non-functional requirements

2.3.2.1 FR1 - Visualize configuration entities and relations in a graph.

- Create visualization of current configuration that depicts entities and relations between them.

- Add control elements to modify current configuration.

2.3.2.2 FR2 - Extract meta data from data sources

- Extract meta data from data sources which will be later used to provide insight eg. relations and probable entities.

2.3.2.3 FR3 - Find entities in data sources

- Process data source meta data and suggest probable relations to the user.

2.3.2.4 FR4 - Find relations between entities

- Process data source meta data and suggest probable relations to the user.

2.3.2.5 FR5 - Save data source metadata

- Save extracted metadata together with configuration. So on successive loads, no further need to connect to data source is needed.

2.3.2.6 FR6 - Visually generate SQL query

- Create visual tool to generate SQL queries with support for simple joins and attribute selection.

2.3.3 Non-functional requirements

New extensions must comply with these non-functional requirements.

2.3.3.1 NR1 - Cross-platform deployment

- Added functionality must preserve cross-platform deployment. Currently, ClueMaker supports Linux, Windows, and macOS.

2.3.3.2 NR2 - Extensibility

- New modules must be developed in such way that they are extensible and maintanable.

2.3.3.3 NR3 - GUI consistency

- Design of the new views must be in sync with the rest of the application and comply with 10 Usability Heuristics for User Interface Design defined by Jakob Nielsen [19].

2.3.3.4 NR4 - Backwards compatibility

- New configuration should be compatible with previous revisions. So new specification should be supper set of previous.

2.4 Configuration model

The primary model of the ClueMaker application contains all the definitions needed to import and visualize data from data sources. Configuration model consists of the following objects:

- Datasources - Definitions of connected data-sources with connection details.
- Entities - Data objects with attributes and relations.
- Mapping - Provides a way of extracting data from data sources by using queries.
- Reports - Templates for reports.
- Saved Searches - Predefined often searched relations

2.4.1 Transaction design pattern

Configuration model contains validation in conjunction with transaction design pattern; this ensures that configuration is always in a valid state, ready to be saved and used by ClueMaker application. Each operation that modifies configuration must be wrapped in the transaction; after success new event is created which notifies presenters.

2.4.2 Serialization/Deserialization

The jackson library does the saving and loading configuration model serializer [20] using specialized mappers that map configuration to DTO object. To minimize the size and complexity of generated JSON configuration name attribute is used as a key defining object during deserialization.

2.5 ClueMaker configurator

In this section, I will introduce and analyze the current project structure and architecture. It is composed of 4 main modules :

- Configurator application - Module that exports application as executable.
- Configurator branding - Module contains style definitions and look and feel for components.
- Configuration editor - The main module implements all editor panels, presenters, and actions that modify the configuration model.
- Datasources editor - ClueMaker supports multiple data sources, each specific in their definition and process of defining mapping to entities.

2.5.1 Configuration editor module

In this section, I will introduce the overarching architecture of the Configuration editor module and describe the configuration model. The overarching architecture is based on MVP design pattern with tightly coupled presenters and views; each view is injected presenter on creation, updates to model are propagated through controller interface implemented by presenters with method writeToConfiguration() this method is called by the view when changes are detected. The class dependencies are visualized in diagrams provided in attachments.

2.5.1.1 Presenters

Each configuration element has Presenter 2.5 class which responsibility is to create and manage the view component. Instances of presenters are created by static factory method `open(Configuration Element)` defined in each Presenter class. Each presenter subscribes to `EventBus`, and implements handle methods.

2.5.1.2 View

View components are created and updated by presenters, all changes to the model are propagated through the controller interface to the configuration

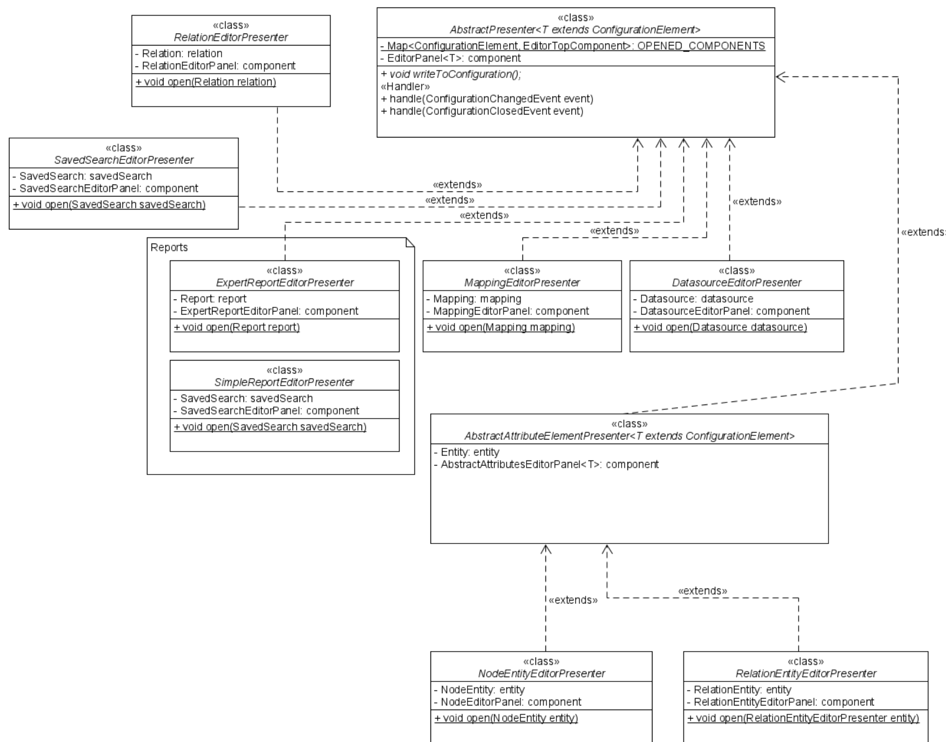

Figure 2.5: Class diagram of presenters handling components of configuration model.

model. Each view is based on `ViewInterface` which defines `update()` method this is a way for presenter to notify view of changes to model. View panels diagram was created and referenced during analysis and subsequent implementation; it is available in attachments.

2.5.1.3 Actions

Actions provide a way to interact with configuration from menus and by using shortcuts. Uses NetBeans open ide annotation to define menu position and short cut. For each of the main components of the configuration model, there are implemented actions to delete, edit and create new ones

2.5.2 Datasource model analysis

To simplify the current creation of mapping and with its interconnected SQL query generation, we need to have some analytics tool at our disposal. To allow future expansion, the whole system was designed with additional analysis libraries in mind. During design few constraints had to be taken into consideration :

- Multiple data sources are analyzed by one analyzer, ex. Drxf library detects relations across databases based on column names and types applicable when databases contain ICO or social numbers.
- Multiple analyzers might be used across configuration for specific data sources, ex. when importing from graph databases and SQL databases, a different analyzer might be used.
- Expected output of the analyzer is suggested entities and relations between them.

2.5.3 Drxf library

The library is internally developed by Profinit EU. Supports databases Postgres, MSQL, Oracle, and H2.

2.5.3.1 Functions

This library provides way of extracting database structure through jdbc driver including foreign keys, column types, table names and structure. Second functionality that I want to use from this library is detection of matches between columns based on type and name, this is useful when matches occur across databases for example telephone number or social security number are expected to be named similarily. Currently 3 types of matchers based on column name and type are implemented :

- `SameColumnNameNamingRule` - checks for identical column names and types.
- `ColumnNameWithTableNameAndUnderscoreNamingRule` - checks for column naming rule {referenced table}_{referenced column}
- `RegexTableColumnNamingRule` - a very general way of creating naming matches where the referenced column is matched with following regex expression .*{referenced table}.*{referenced column}.*

The latest functionality added to this library is the categorization of column data that might be sensitive, these can also be used to generate matches, but this feature is currently not fully implemented. Currently, the library supports 11 categories :

- Address - uses provided dictionary to detect addresses
- Credit card - regex matcher used to validate credit card numbers.
- CZ bank account - regex matching
- CZ tel number - detects phone numbers
- Email - uses apache commons EmailValidator

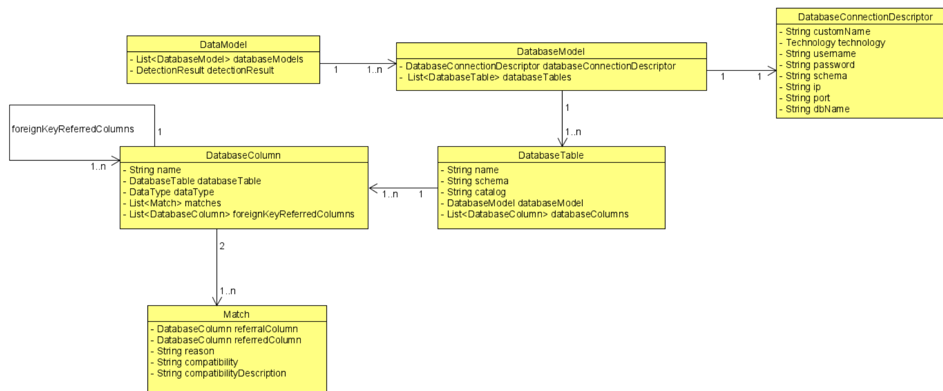

Figure 2.6: Data model generated by analyzer library.

2.5.3.2 Data model

The resulting data model is returned as POJO classes with data structure depicted in diagram 2.6. Each database and schema has separate `DatabaseModel` generated containing tables and columns, each `DatabaseTable` object has a list of `DatabaseColumns` that contain foreign key referred columns, these are useful in detecting relational entities, together with all detected matches.

2.6 User inteface design

In this section, the author describes user interface windows designed. All wireframes that were created and used during implementation are available in attachments.

2.6.1 Wizard

Wizard consists of panels guiding the user through the process of generating entities and links between them. The main emphasis while designing the wizard was placed on ease of use and repeatability of the steps with the already opened configuration; also, the wizard should be fully featured so all changes to the configuration can be done solely by it. Each panel should contain validation of inputs to ensure the validity of generated configuration.

Wireframes

Wizard window [2.7] contains navigation buttons and legend. The main panel of the wizard changes based on the current step. When creating entities/mappings/relations, suggestions are presented to user-generated based on extracted metadata in the form of modal dialog windows.

Figure 2.7: Configurator main window with opened entity editor and mapping.

1. Select configuration - Screen that gives the user the option to select a configuration to start the wizard.
2. Select data sources - Panel that deals with data source definition and data source creation. The left side contains a searchable list that can be used to select a data source to edit.
3. Edit entities and mapping - Panel that defines entities and their respective mapping. The panel is split in the middle to provide an overview of the entity and the related mapping.
4. Edit relation entities and mapping - Panel that deals with relation entities definition. Similar to entity panel with extra features concerning relation selection.
5. Edit relations - Define and edit relations between entities. Relation editor window.

Dialogs

Provide basic functionality to select from suggested values or fill the form to create a new entity/relation/relation entity, wireframe of new entity dialog can be found in 2.8. Each window contains a cancel button and validation element.

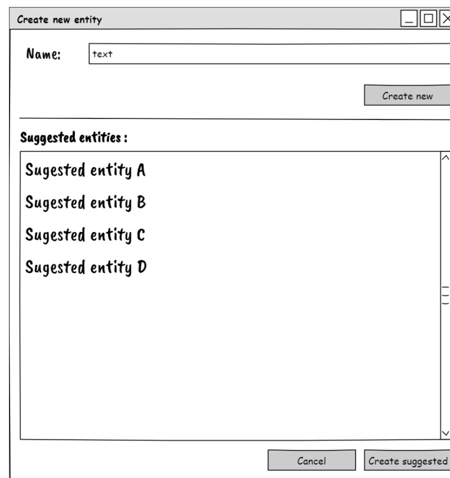

Figure 2.8: Modal dialog used to create new entities.

2.6.2 Configuration visualization

Visualization is designed as a graph view with widgets representing entities and edges representing relations, together with a searchable list of entities and a toolbar that provides extra functionality.

Wireframes

The main window contains zoomable, stretchable, and pannable scene with components laid out based on the currently selected layout. The toolbar is located on the top of the panel with functions like layout, fit to the scene, etc.

2.6.3 Visual SQL Query generation

To ease SQL request creation, I designed an SQL query builder that uses extracted metadata to provide GUI guidance to the user.

Wireframes

Depicted in figure 2.10. On the left side there is searchable list of tables contained in the selected datasource. On the bottom current query text editor. Central portion of the editor occupies window visaulizing currently selected tables and their connections. Each attribute of table contains checkbox to select the attribute.

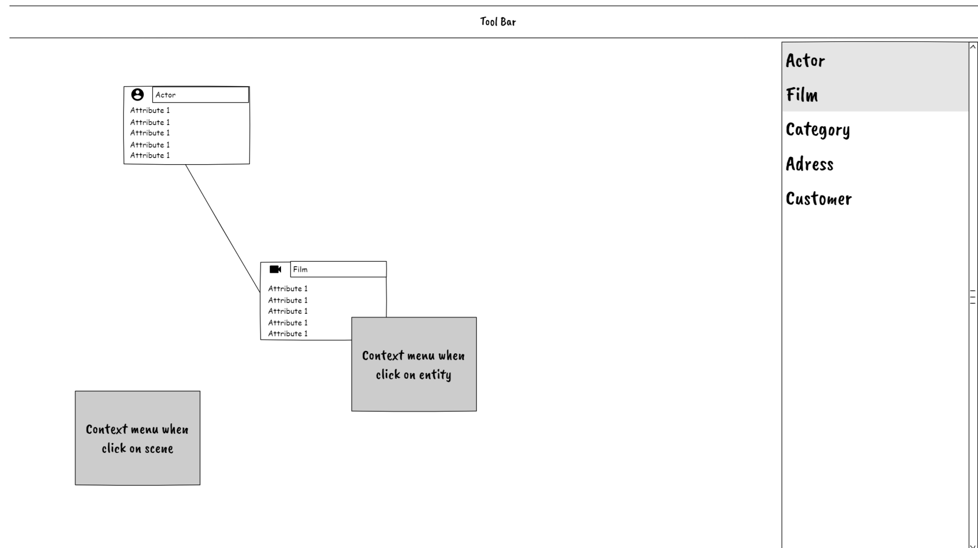

Figure 2.9: Configuration visualization as a graph.

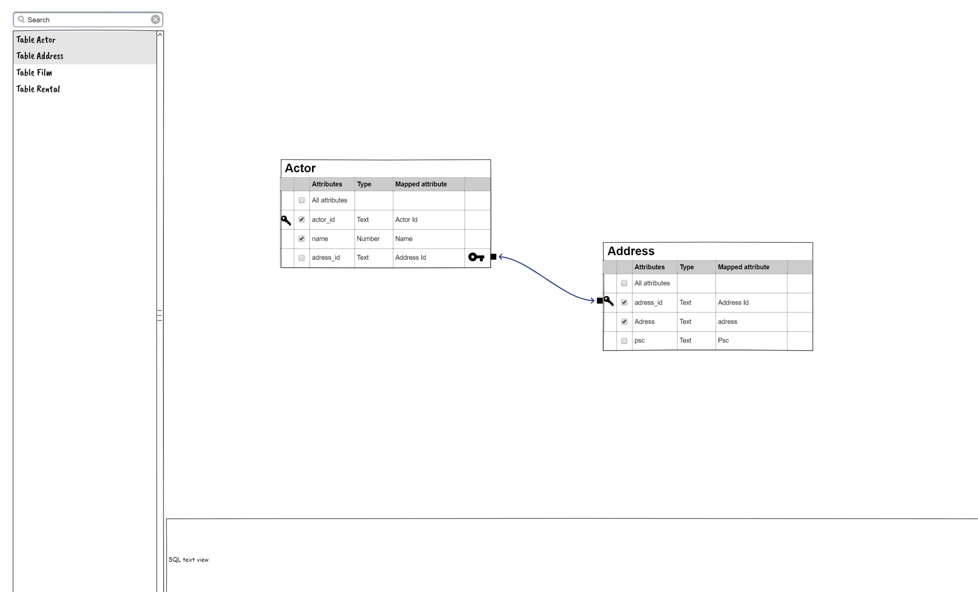

Figure 2.10: Visual SQL builder.

Chapter 3

Implementation

In this section, the author will describe the process of developing new modules and changes made to the existing application.

3.1 Used libraries

Selected libraries are heavily influenced by the current implementation of modules in ClueMaker. Each dependency is included in the External libraries module and included in implemented modules.

3.1.1 Jackson

JSON serialization/deserialization library [20] with useful feature set of annotations and serializer settings. The project is open-sourced and is provided under Apache License 2.0.

3.1.2 JSOG for jackson

Acronym for JavaScript Object Graph, this plugin [21] for Jackson can serialize cyclic object graphs in the JSOG format. Functionality is achieved by creating a set of all objects used during serialization and assigning them unique `@id` tags. When an already encountered object appears again during serialization, it is replaced by the `@ref` tag; a similar idea is employed during deserialization, instances of objects with filled `@id` tag are encountered first, and already deserialized objects replace `@ref`. This software is provided under the MIT license.

3.1.3 Visual library API

Part of the Apache NetBeans toolkit [10] used for generating graph visualization. This is the go-to visualization library in ClueMaker already used to

create graph visualization in the main application and is tied into the NetBeans ecosystem, licensed under Apache License 2.0.

3.2 Implemented Modules

The modified application consists of NetBeans modules which separate concerns into logical sections. In this thesis, I implemented five new modules, three relating to data source analysis, one for wizard and one for visualization of configuration.

- Datasource analysis
  - Datasources analyzer - common definitions of data sources analyzer.
  - Databases analyzer - definitions of metadata and mapping relating to database data sources.
  - Drxf databases analyzer - analyzer implementation based on Drxf library.
- Configuration wizard - Wizard guiding user through steps required to create a configuration
- Configuration graph visualization - Module handling visualization of currently loaded configuration as a graph.

3.2.1 Data sources meta data

The first functionality implemented was related to metadata extraction from data sources. This part will explain the realization steps taken to implement this functionality to the ClueMaker configurator.

3.2.1.1 Configuration model extension

To store extracted information from data sources, additional properties needed to be added to the current configuration model. Metadata depends heavily on the analyzer used, so this was chosen as the main point of differentiation and base for the current design. Each analyzer has to follow the predefined structure of extracted data 3.1.

For these reasons following data structure was implemented :

- `Datasource` - added `DatasourcesMetaData` object representing output of single datasource analyzer.
- `AbstractMetaData` - base class that each object in meta data must inherit. Provides annotations for JSOGgenerator and @JsonTypeInfo used during serialization.

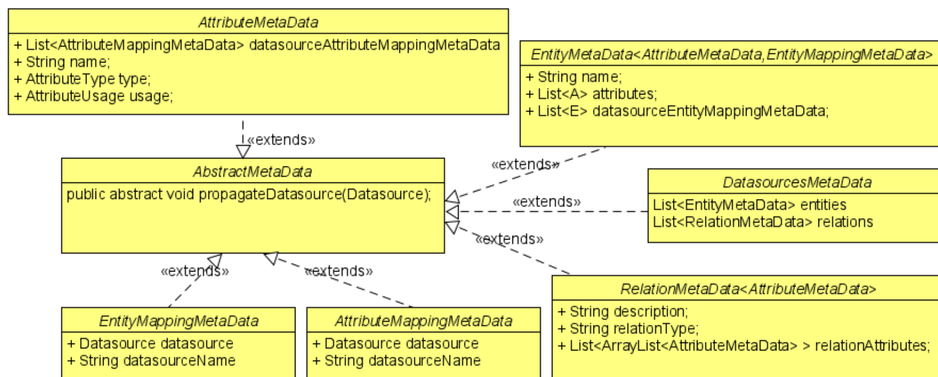

Figure 3.1: Class diagram of meta data structure.

- `DatasourcesMetaData` - contains extracted entities and their relations this information is used in automatic configuration creation.
- `EntityMetaData` - defined by extracted attributes and extracted entity name. Extracted mapping from datasources to this entity is also stored for future reference during assisted mapping generation for example table in database.
- `EntityMappingMetaData` - defines which components of datasource map to this entity. For example structured sql request or plain sql query string.
- `AttributeMetaData` - described attribute extracted during analysis.
- `AttributeMappingMetaData` - defines which sections of datasource map to this attribute. ex. Column in Table.
- `RelationMetaData` - defines connection between two `EntityMetaData` objects. Contains list of `AttributeMetaData` pairs.
- `SuggestedRelationEntities` - class defining detected `RelationEntity` in meta data that connects to currently defined entities.
- `SuggestedRelation` - class defining detected `Relation` in metadata for currently defined entities.

3.2.1.2 Serialization/Deserialization

The main challenge while implementing this data structure was the inclusion of multiple circular references, mainly `AttributeMetaData` is referenced throughout `RelationMetaData`, and backward references, to handle this during serialization/deserialization annotations provided to jackson library.

@JsonIdentityInfo with JSOGGenerator is used to handle circular reference of an object by serializing the back-reference's identifier (@id and @ref fields are added during serialization). During this step, I needed to take extra care in creating a structure that is backward compatible and would be able to use the current infrastructure to import data.

Loading configuration from the .sws file poses additional challenges, mainly when coupled with a refresh of analysis metadata. Since new objects are created on each call, and these should be mapped to existing objects, the cache was implemented in `DrxfMetaDataCatche` which uses overridden hashCode and equals methods, this solution together with factories creating metadata objects, handles consistency of objects after deserialization. These classes are required to stay as POJO to help with serialization.

This functionality is provided by `JsonConfiguration` module in the main ClueMaker application; changes to the DTO configuration class had to be made to facilitate serialization and deserialization of metadata.

3.2.1.3 Datasources analyzer

This module defines interfaces and implements a dummy data source analyzer. It also contains `@ServiceProvider` annotation for the factory, which creates instances of specific analyzers. Thus, this implementation creates a plugin infrastructure for analyzers.

`DatasourceAnalyzerInteface` - defines an interface that is used throughout the application. Each instance of this interface is self-describing, exposing getters for name and description and methods for registering data source for analysis. The main functionality of the analyzer is to extract metadata from the data source, which contains detected entities and relations between them, and subsequently generate `SuggestedRelationEntities` and `SuggestedRelations`. Dummy implementation of this interface is included in this module. Main interface methods description :

- `loadDatasourcesMetaDataCatche` - used to during deserialization to initialize object cache. This is handled internally by each analyzer.
- `getAnalysisTask` - returns analysis task that main goal is to extract metadata from data sources.
- `detectRelationEntities` - main method used during detection of relation entities from metadata and currently loaded mappings.
- `detectRelations` - used to create relations from current mappings definitions and metadata.

`DatasourceAnalyzerFactory` - creates instances of analyzers. This module implements default implementaion which looks for `@ServiceProvider` antotation which imeplement `DatasourceAnalyzerInteface`.

`DatasourceAnalyzerTask` - task object produced by an analyzer that handles extraction and analysis of metadata. It is expected that more advanced analyzers will require to run for a more extended time. To avoid unwanted freezes of the application, each analyzer must implement its analysis in a task and adhere to the interface defined by this abstract class. On-task fail message is provided with a reason for failure, and resulting metadata is left empty.

3.2.1.4 Database analyzer

This module is more specific implementation of Datasources Analyzer module with was implemented to define meta data structure for SQL databases. On top of extending `DatasourcesMetaData` properties and functionality encapsulates data source metadata and structure including tables and columns with types these are later used during creation of custom SQL queries. Implemented classes include :

- `DatabasesMetaData` - contains extracted entities and their relations this information is used in automatic configuration creation. Also contains description for each database datasource.
- `DatabaseDescription` - contains reference to datasource object from configuration and tables.
- `DatabaseTable` - contains name and columns list with back reference to database description.
- `DatabaseColumn` - contains definition of column with back reference to parent table.
- `DatabseEntityMappingMetaData` - extends `EntityMappingMetaDat` and adds structured SQL query which maps to selected `EntityMetaData`. Contains `DatabaseStructuredSQLQuery` object which defines sql query to import this entity.
- `DatabseAttributeMappingMetaData` - contains database column which maps to selected attribute. This object is used to detect relations between user generated entities and detected entities.
- `RelationMetaData` - defines connection between `EntityMetaData` objects. Contains list of `AttributeMetaata` pairs.
- `DatabaseMetaDataMappingConfiguration` provides new definition of sql query that uses `DatabaseStructuredSQLQuery` as base implementation.
- `DatabaseMappingConfiguration` and overrides default getSql() method this ensures backwards compatibility through out the application and in main ClueMaker app.

3.2.1.5 Drxf library integration

To take advantage of functions provided by the Drxf library, I implemented the module Drxf database analyzer, which wraps functionality into an already specified interface. The resulting analyzer is registered under the analyzer interface, so it can be easily accessible using NetBeans Lookup.

Mapping is created using factories with cache for objects. This ensures that subsequent calls to the analyzer generate the same objects already used throughout the application.

Since this library is developed in-house, additional changes were made, new hash and equals functions were implemented. Additionally, detection of primary keys was added; this is required for automatically creating entities and speeding up imports in the ClueMaker application.

`DrfxAnalysisTask` - this task generates DrxfMetaData object. Firstly generates connection and extracts schemas from database. Then creates list of `DatabaseDescription` objects that are based on registered data source configurations. Then runs reference search and name search on databases. Converts data from the Drxf library data model to the internal structure; this step is necessary to have an independent database analyzer implementation. After generating an internal database structure, the analyzer creates entities metadata found in the database structure. Last step is generating relations based on `Match` objects located in Drxf `DatabaseColumn` class.

Suggestions for relation entities are generated in 2 steps:

1. All possible relation entities are generated.
2. Mapping configuration elements are scanned for corresponding entities that satisfy conditions set in relations, filled entities

Generated suggestions are filtered by looking directly into the inner structure of `Mapping` configuration elements and scanning queries for structured implementation. When suitable match is found extracts `EntityAttribute` and `Entity` and assigns them to suggested relation entity.

3.2.2 Configuration graph visualization

This part of implementation is contained in separate module handling visualization of configuration entities and their relations as widgets on the graph. I used Visual library API as the main framework handling visualization. Implementation is separated into three main layers, each separated through defined interfaces and message specification; this architecture is loosely based on The Clean Architecture design specification [22]. To achieve loose coupling between layers, I use the dependency inversion principle [23] which ensures that boundaries are crossed using abstractions (interfaces). This design allows for replacing and testing each layer separately and mocking and injecting interactors to check if correct actions are generated.

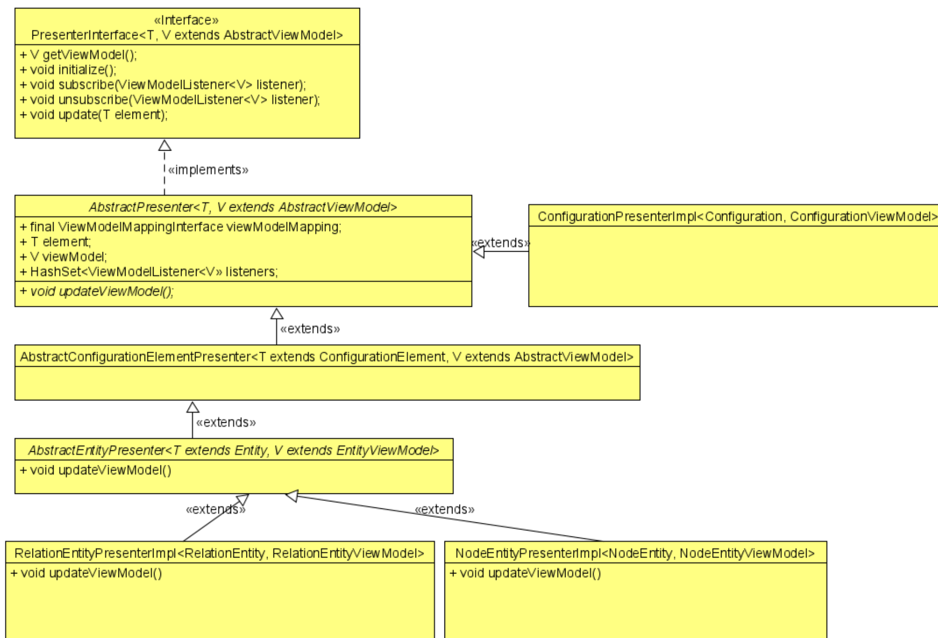

Figure 3.2: Class diagram depicting composition of presenter classes.

3.2.2.1 Model

The core of the implementation is still the configuration model [2.4], which defines entities that we want to visualize. To cross-boundary to presenters and controllers, there are defined following interfaces :

`ConfigurationControllerInterface` - defines methods which handle actions from view.

`ConfigurationListenerInterface` - interface which allows to listen on changes to configuration.

In visualization implementation, we want to represent these parts of the model:

- Node entities and their attributes as nodes in the graph.
- Relation entities and their attributes as nodes in the graph connecting node entities.
- Relations as edges.

3.2.2.2 Presenters

Presenters handle transforming model entities to view models which represent graph structure of configuration. Presenters also implement publish-subscribe design pattern by exposing generic `ViewModelListener` interface.

Generic interface `PresenterInterface<T, V extends ViewModel>` was defined to create generic interface for any presenter that maps object of type `T` to `ViewModel` subclass. Abstract class `AbstractPresenter` implements subscription part of the interface and creates attributes that hold elements to be visualized and currently generated view model, also implements outward-facing update(T element), this implementation notifies on `ViewModel` change.

Presenter factories Presenters are created in factories with a cache to ensure one-to-one mapping of entities to the corresponding presenter. Each instance of a factory is registered as `ServiceProvider` of factory interface for the given entity type. In addition, the protected method createPresenter() is internally defined and provides initialization for each presenter.

Mapping to entities To facilitate mapping entities to `ViewModel` and vice versa, custom `ViewModelMappingInterface` was designed. The default implementation of this interface is registered as a service with bi-directional mapping of entities and view models using hash maps. Presenters register new instances of view models to this cache. Controllers map view model back to configuration model object; this separation ensures that all communication from view model must flow through controllers since the view has no direct reference to the model.

3.2.2.3 Contollers

Implements handling of user actions from view. ActionMessages are POJO classes that define action type and view models that this action concerns, view models are subsequently mapped to real entities, and updates on the model are executed.

3.2.2.4 Interactors

Interactors are used to communicate with the Configuration editor module. An example of an action that is handled in this fashion is an open editor action on an entity; this action is the opening of the editor window defined in the configuration editor module. The result of separating this module from the rest of the application through interactors is that tests can be written to check expected action calls instead of checking the effects on the entire application.

3.2.2.5 ViewModel

POJO classes that represent the current state of the model to view. Each view model implementation extends AbstractViewModel and contains a unique UUID assigned during initialization to the presenter. Overridden hash and equals methods provide a way to identify objects even when the view loses

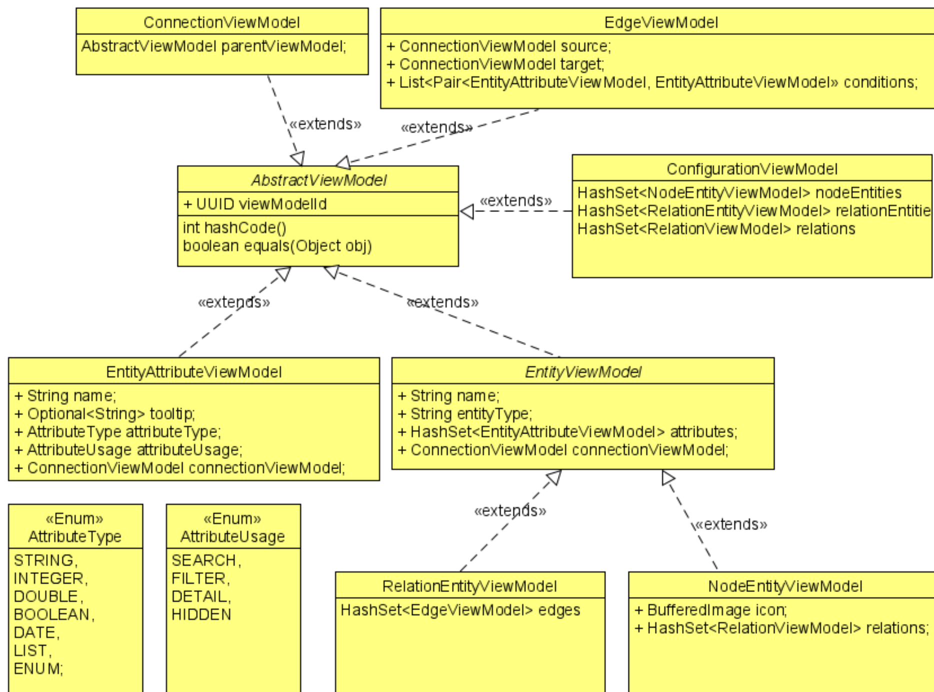

Figure 3.3: Class diagram visualizing dependencies of view model implementation.

object reference. These objects are serializable and can be used to implement different forms of views, for example, web-based. Implemented view models:

- `ConfigurationViewModel` - view model representing configuration.
- `ConnectionViewModel` - defines connection point.
- `EdgeViewModel` - contains 2 connection points and conditions representing relation.
- `EntityAttributeViewModel` - view model for repsresenting attribute.
- `EntityViewModel` - abstract class that provides base for entities view models.
- `NodeEntityViewModel` - adds relations and icon to base implementation.
- `RelationEntityViewModel` - adds edges property to view model which represent incoming and outgoing relations.
- `RelationViewModel` - represents relation.

3.2.2.6 View

The view is implemented as a standard `TopComponent` in the NetBeans platform and is registered in the main context menu Window.

Class `ConfigurationGraphView` is the main component representing graph. The base of its functionality is inherited from generic class `GraphPinScene`. This class provides structure and methods for managing graph models as well as mapping to widgets. The class is abstract and manages only data models and mapping with widgets. The graphics (widgets) are supplied by overriding the `attachNodeWidget`, `attachPinWidget`, `attachEdgeWidget`, `attachEdgeSourceAnchor` and `attachEdgeTargetAnchor` abstract methods. Each pin is assigned to a node. This class uses generics and allows to specify type representation for nodes, edges, and pins in the graph model. ViewModel is designed to conform to these specifications. Since the type of nodes, edges, and pins could be the same, all node, edge, and pin instances have to be unique within the whole scene.

Custom widgets were implemented to generate a view from view models :

- `AttributePinWidget` - implements listener for changes in entity attributes view model, also contains special anchor implementation for edge which position is calculated based on type of connection (source/destination).

- `EntityConnectionWidget` - extends `ConnectionWidget`. Represents edge in graph.

- `EntityWidget` - implements `ViewModelListener` for `EntityViewModel` split into header widget and attributes.

- `NodeEntityWidgetImpl` - extends `EntityWidget` and adds image icon to widget header.

- `RealtionEntityWidgetImpl` - extends `EntityWidget`.

3.2.2.6.1 Graph actions Multiple ways to interact with graph nodes were implemented, including zoom on a graph, pan action, select node action, box select action, via context menu anywhere on the scene or via the provided toolbox.

Context menu actions include :

- Hide/Show all nodes

- Layout

- Create new entity - opens create entity dialog window.

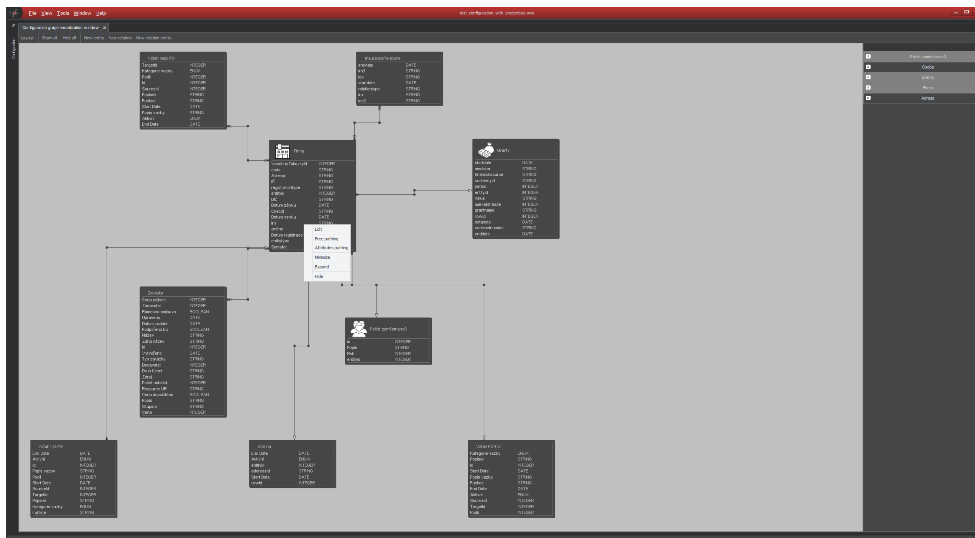

Figure 3.4: Visualization of entities and relations in graph using orthogonal routing for edges.

- Find relation entities/relations - when Entity widgets are selected this functionality searches for relation entities where currently selected nodes are present.
- 3 types of routing options, orthogonal [3.4], directed [3.5] or free routing (User sets points on edges that stay in place).

3.2.2.6.2 Widget actions Each widget implements a primary move listener. In addition, entity widget registers context menu action that opens up a menu that is constructed using the Swing component of `JPopupMenu` and menu items are created using `JMenuItem` with the following options:

- Hide - hides this widget from the graph view
- Expand - Shows related entities and relations
- Attribute pathing - edge connections are related to attribute widgets
- Node pathing - edges are routed from the edges of the entity
- Minimize - Shows only the header of the widget

3.3 Configuration wizard

It uses a more straightforward implementation based on the NetBeans wizard model; this stems from the fact that current editor panels are used and

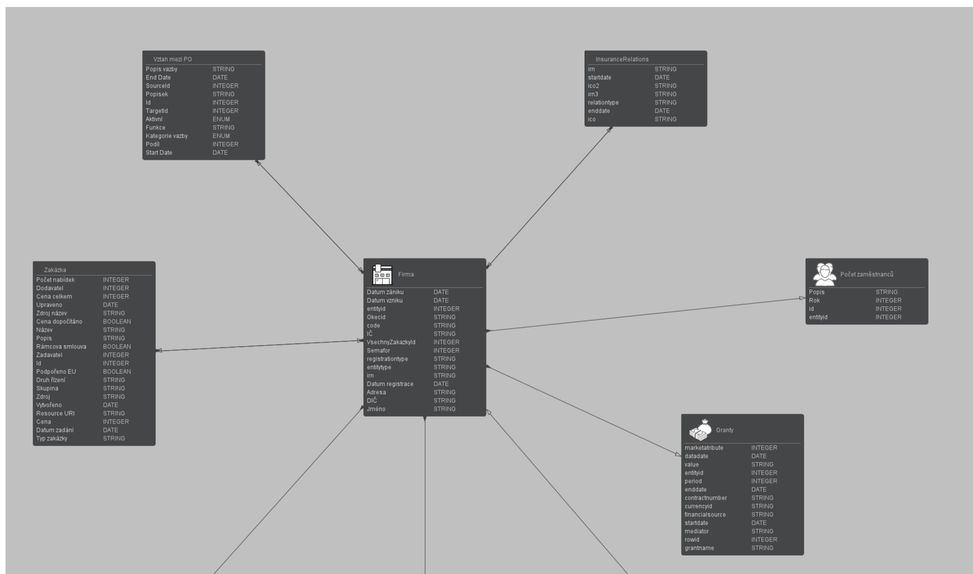

Figure 3.5: Visualization of entities and relations in graph using direct routing for edges.

wrapped in custom panels corresponding to wireframes in design. As a result, users can generate equal configuration through the wizard through the standard editor part of the application and edit any configuration.

3.3.1 Presenters

Each wizard step has assigned presenter witch handles validation and panel generation. Each presenter extends `ConfigurationListener` which subscribes to current configuration on `readSettings()` method invocation, then propagates update calls to assigned component.

3.3.2 View

View panels implementation adheres to wireframe designs. On the left side of the window, users can browse through all defined objects editable in the current panel; the right side is reserved for editor panels [3.6]. Editor panels used for representing entities were reused from an existing application; this ensures that users are not getting confused by two different editors and adds guided configuration creation. All user interactions that influence configuration are mapped to corresponding actions and executed.

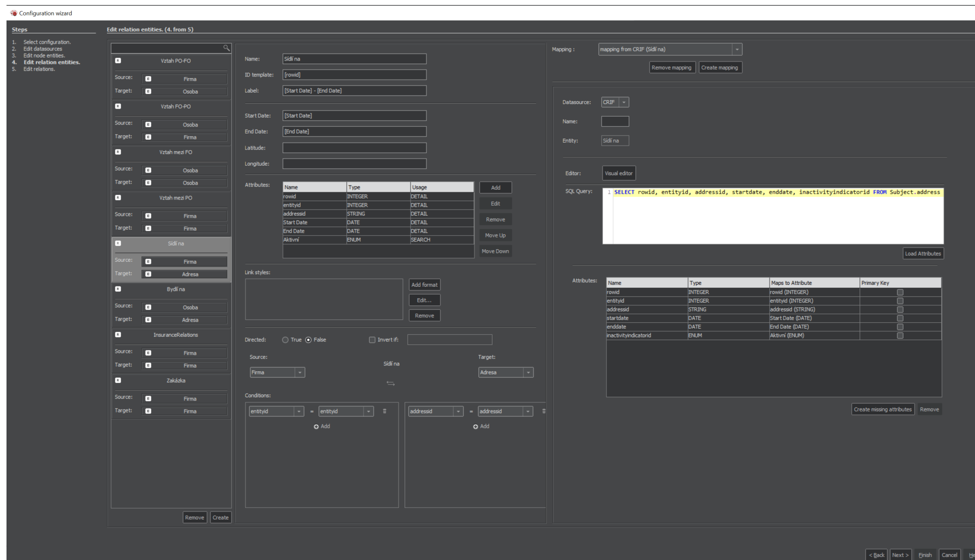

Figure 3.6: Wizard panel for relation entities creation.

3.3.3 Dialogs

Creating new Entities, Relation entities, and Relations are done through dialog windows that contain two parts. A list of suggested entities extracted from metadata and simple form to specify the name of created entity/relation/relation entity.

3.4 SQLQuery builder

The biggest addition to the current workflow when creating configuration is visual SQL builder; this component allows a novice user to create SQL queries without having the database opened and creating them manually. This component is available on data sources that have metadata compatible with `DatabasesMetaData`. `VisualSQLBuilder` component is based on `GraphPinScene` and uses internal information about structure of database. To simplify query generation, users select one table that is the root node for query and click on relations visualized in the form of a button with a key label. SQL is built dynamically in the bottom text window; this text field is non-editable. Users can select individual attributes from tables or decide to select all attributes.

To extract query from the currently visualized graph, I implemented the following algorithm :

1. Find root node of the query that is being visualized. This is trivial since all our queries start with a single node selected.

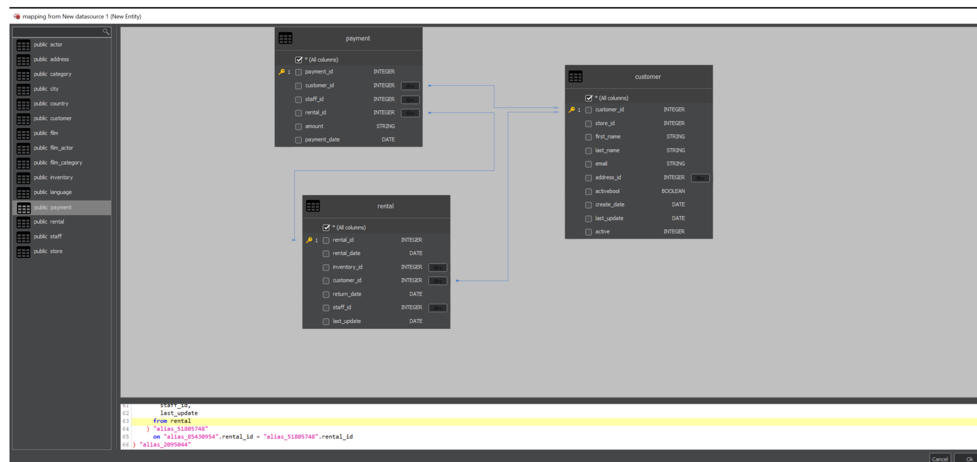

Figure 3.7: Visual SQL builder view.

2. Recursive function is then called on the top node with all currently opened nodes and aggregator for already visited nodes. The fact that our root node is not part of the query does not necessarily mean that there are absolutely no cycles in the graph.

3. On each call function looks for node successors that were not visited already.

4. Recursion ends if no additional successors are found.

```
1  struct SqlQuery {
2    bool isJoin;
3    SqlQuery left;
4    SqlQuery right;
5    Table table;
6  }
7
8  SqlQuery generateSql(Set<Table> graphTables, Table rootTable, Set
       <Table> expandedSet){
9  // add root node to expanded
10   expandedSet.add(root)
11
12 // returns childs of the Table by foreign key
13   referedTables = getReferredTables(root)
14
15 // remove all already expanded
16   referedTables.remove(expandedSet)
17
18   aggregator = SqlQuery{
19     table = rootTable
20     isJoin = false
21   }
```

```
22
23   for (Table nextTable in referredTables)
24     aggregator =  SqlQuery {
25       left = aggregator
26       right = generateSql(graphTables, nextTable, expandedSet)
27       isJoin = true
28     }
29
30   return aggregator
31 }
```

Listing 3.1: SQL structured query generation pseudo code.

One of the main limitations of this implementation is that queries are not optimized, and the algorithm cannot generate self-referencing table queries; each table must be used only once.

3.5 SQL query generation

One of the more algorithmically exciting parts of the implementation was the generation of queries from the graph.

3.5.1 Creating queries

To save structural queries, I implemented `StructuralSQLQuery` class that implements recursive data structure that defines semantical model as a binary tree of subqueries. Each node in a graph is defined either as a join or table. This data structure is subsequently traversed in order to generate an entire query. The backbone of this implementation is the JOOQ library which composes defined data into queries and wraps them into subqueries.

3.6 Additional components

During implementation, more generic components were developed to be reused throughout the entire application.

3.6.1 Searchable list

One of the most used components throughout implementation provides a searchable list of a generic type with a custom renderer for items. Components that represent items in the list need to be stateless objects since no update is called on change outside of the list. This composition dramatically simplifies the creation of complex list visualization used, for example, in suggestion boxes and dialogs.

3.6.2 Create Node entity dialog

This dialog window serves as a guide during the creation of NodeEntity; the primary purpose is to help the user with filling required data. It also provides the user with suggested entities generated from Datasources metadata and prefills mapping for this entity.

3.6.3 Create Relation dialog

To help generate relations between entities, a simple dialog window was used, where the user is asked to either select from suggested entities or define new relations by selecting the source and target node entities in provided combo boxes.

3.6.4 Create Relation entities dialog

The modal dialog is used to suggest a new relation entity to the user.

3.7 Additional changes

To facilitate the use of editor panels in the wizard, multiple changes needed to be made to the current implementation. Firstly static factory methods used in presenters were substituted and refactored to support successive calls. Only one instance of a presenter is required since they generate a single panel component. New actions were added to the Configuration editor module to facilitate the creation of entities and relations using dialogues as well as filtering based on selected nodes in visualization.

3.8 Evaluation of requirements

During this section, the author will summarize what functional requirements were full filled during implementation.

3.8.1 FR1 - Visualize configuration entities and relations as a graph.

This functional requirement is implemented in the Configuration Visualization module. With additional functionality to edit configuration straight from visualization, this feature was added during the design process to make the interface more user-friendly.

3.8.2 FR2 - Extract metadata from data sources

A new plugin infrastructure is implemented to support additional analyzers added to the implementation. Currently, only a single supported analyzer is

used based on an internally developed Drxf library that extracts data from SQL databases and matches column names and types.

3.8.3 FR3 - Find entities in meta data

Currently, all tables are considered entities; this solution works for simple databases, but when it comes to analyzing complex tables with multiple relational entities, these might get lost; further filtering is required.

3.8.4 FR4 - Find relations between entities

Dialog windows were implemented that suggest relations to the user. There are created as actions, so they are available across all application GUI.

3.8.5 FR5 - Save data source metadata

The custom data structure was created, and special serialization tags were used to serialize complex metadata with multiple cross-references. This model is extensible to suit any data sources analyzer in the future.

3.8.6 FR6 - Visually generate SQL query

Visual SQL query builder [2.10] was implemented to help new users with writing queries and defining metadata.

3.8.7 NR1 - Cross-platform deployment

No additional constraints were put on the application; all used libraries were either already in use or had multiplatform deployment.

3.8.8 NR2 - Extensibility

The entire design of analyzers is structured around being able to add additional libraries and advanced features. In addition, metadata structure was made flexible to allow for future expansion.

3.8.9 NR3 - GUI consistency

Existing panels were reused in the wizard, and graph controls were unified across applications.

3.8.10 NR4 - Backwards compatibility

Backward compatibility was one of the hardest things to implement. The current implementation of serialization was rigid and did not implement object references. The entire structure was mapped to flat DTO. Mapping metadata

flat was not an option since custom analyzers define them; this led to current implementation with annotation and repopulating the object references back based on names of objects from the configuration.

Chapter **4**

Testing

Testing is the development phase performed during the entire application implementation process used to detect software errors and prevent undesirable effects of changes to the codebase.

ClueMaker application already uses various testing frameworks to support a fully featured test suite. JUnit 5 [24] is used as the primary testing engine. For mocking and stubbing interfaces, ClueMaker uses Mockito framework [25]. Also, the use of the Maven surefire plugin plays a vital role during automated testing.

4.1 Unit tests

Unit tests were used during the implementation stage; this is due to the nature of the application and to speed up development and catch errors early, various mocking techniques were used to simulate a connection to other modules, for this functionality, the internal implementation of `MockLookup` implemented in testcommons module is used to mock default NetBeans looks up. This ensures that each module is testable independently with the use of mocked interfaces.

These are the principles the author utilized during test creation:

- Fast — test should be fast.
- Independent — tests should not depend on each other. Tests should be able to run in any order.
- Repeatable — tests should be repeatable in any environment.
- Self-Validating — Tests should have a boolean output.
- Minimize the number of assertions per test.

4.1.1 Unit testing Drxf databases analyzer module

Library interacts with the database, so the first step in creating unit tests was defining sample database; for this purpose, ClueMaker implements H2 database, which is entirely contained in memory. During test class set-up new instance of the database is created and populated with one of the two sample scripts located in test sources. This library is tested in two stages:

First is connecting to the database and extracting metadata; these tests are located in `TestDrxfMetaDataTask` class.

The second functionality that is being unit tested is the generation of suggested entities and the loading instance of configuration with MetaData already attached. These tests were implemented during the development of a serializable data structure. Tests reside in `TestDrxfDatabaseAnalyzer`.

4.1.2 Unit testing configuration visualization module

Strict separation of presenters, controllers, and view through dependency inversion principle and dependency injection allows for simple implementation of unit tests.

Presenters are fed configuration and checked for correct view model change. These tests are implemented per presenter basis. Thus, they greatly simplify and speeds up the implementation process since entire components can be developed independently of the main application.

Controllers would follow a similar pattern as presenters. Unfortunately, there is no function I could test since there are no actions directly affecting configuration; all current actions use dialogs and actions implemented in the Configurator editor module and are just relayed through the interactor.

4.1.2.1 Additional tests

Pagila [26] is a large sample database to check against during testing. I ran unit tests against this database, but it depended on running the Postgres database on the local server. There is no way to keep these tests in the unit tests suite and are commented out. Future unit tests could be run in a stable environment like docker or on a deployment server where these sample databases are available and versioned. This would allow for a faster response time to JDBC driver changes or run tests against multiple different versions of the database.

4.2 Heuristical analysis

The primary purpose of this analysis is to look through the application and find design antipatterns broadly.

1. Visibility of system status

- Application contains validation that keeps the user informed if there are any errors during wizard steps.
- Metadata extraction might take a long time that might upset users; that is why a dialog window with progress is opened during the import process.

2. Match between system and the real world

   - Currently, there is a disconnect in visualization style between the new and old modules. This should be easily rectified by using new custom icons for database keys and tables.
   - Wizard uses editor panels that users are already comfortable with.
   - Actions on the graph are intuitive, and controls follow well-known conventions. (Mouse wheel to zoom, Mouse wheel click to the pan)

3. User control and freedom

   - Each modal dialog has a well-defined way to exit through the cancel button, nullifying the effect of an action or the normal closing of the frame.
   - Even during wizard steps, users are not tied down to a single step and can move freely throughout.
   - The one thing that's not currently implemented is Undo and Redo features, but the user is always notified when an irreversible step is being taken (Deletion of part of the configuration)

4. Consistency and standards

   - As already described, the wizard uses already defined panels. During this thesis, the extra functions implemented to them propagated to the main editor interface, so novice users get accustomed to it.

5. Error prevention

   - Currently, all modal dialogs used in creation are set up to rectify user mistakes. For example, when the node entity name is left blank default name is used.

6. Recognition rather than recall

   - During wizard steps, entity editor and mapping are presented side by side so the user can reference extracted attributes and create mapping without the need to switch context.

7. Flexibility and efficiency of use

- Design of wizard was more catered for novice users, so no new shortcuts or optimizations were created for the power users.

8. Aesthetic and minimalist design

   - Minimalization of widgets and various routing options were implemented and actions to hide or extend node entities.

9. Help users recognize, diagnose, and recover from errors

   - Created dialogs are designed not to give the user much opportunity to fill invalid data.

10. Help and documentation

    - Currently, there is no documentation for provided functionality available for users; this will be rectified by the time this functionality is released to the public.

4.3 Cognitive walk through

"The CW identifies usability problems by simulating step-by-step user behavior for a given task" [27, p.463], and by answering the following questions at each simulated step:

- Q1 - Will the user try and achieve the right outcome?
- Q2 - Will the user notice that the correct action is available to them?
- Q3 - Will the user associate the correct action with the outcome they expect to achieve?
- Q4 - If the correct action is performed, will the user see that progress is being made toward a solution of his/her task?

4.3.1 Graph visualization

Scenario user decided that he/she wants to extend current configuration with new entity actor to add analysis of favorite actors of customers. Graph visualization window create new entity task, steps taken:

1. User presses toolbar button "Create new entity"
2. User selects entity from suggestion list
3. User clicks "Create suggested"

Action 1 There were no main issues found with this step.

1. Yes - The user will attempt to find a button that allows the user to create a new entity.
2. Yes - These buttons are implemented in the context menu and toolbar and are predominantly visible.
3. Yes - The user wants to add a new entity to the current configuration; the button is named ”Create entity”.
4. Yes - User is prompted with a dialog to add a new entity to configuration.

Action 2 The main problem in this step is that not enough information is given to the user on entities being created. The solution is creating a new section that provides details on the currently selected entity.

1. Yes - The user’s main goal in this step is to select and view a new entity from a predefined list to be created.
2. Yes - The selected item is highlighted in the list.
3. Yes - The result is selecting this entity from the list.
4. No - Not enough visual information is provided to the user when selection happens.

Action 3 The problem within this step is that the entity might appear outside of the visibility of the graph window. The solution is positioning new nodes in the center of the screen.

1. Yes - User creates currently suggested entity.
2. Yes - The button is visible in the dialog window and is positioned in the expected location.
3. Yes - Button name and action correlate.
4. No - User might not see that entity was created when it’s positioned outside of the current scope of graph view.

4.3.2 Wizard

Wizard window create configuration task, steps taken:

1. User selects open configuration option
2. User clicks next on the wizard window

3. Select configuration file from selector
4. User selects entity from suggestion list
5. User clicks ”Create suggested”

Action 1 There were no main issues found with this step.

1. Yes - The form selection of configuration is designed in a way that matches the user’s mental model
2. Yes - All radio options are presented with correct labeling, and it should be easy for the user to recognize the correct action.
3. Yes
4. Yes - radio button is checked

Action 2 The next button should change the label based on the action selected in the radio button.

1. Yes - The user wants to continue to the next page of the wizard.
2. Yes - Next button is using a standard wizard layout.
3. No - There might be some confusion with what the next button is going to do, depending on the currently selected radio button.
4. Yes - File selector window will be opened.

Action 3 There were no main issues found with this step. A file selector is a standard component of the operating system.

Action 4 Uses identical dialog window to Graph visualization cognitive walkthrough 4.3.1, the identical analysis applies.

Action 5 The problem in this step is that wizard does not change windows to the currently created entity. The solution is to open both the mapping and entity editor panel relating to the new entity created.

1. Yes - User wants to create the selected entity.
2. Yes - The button is in a standard location.
3. Yes - The button is labeled as ”Create suggested”.
4. No - Wizard still displays the old entity selected previously by the user.

4.4 Usability testing

To review the current implementation and gather user feedback, usability tests were executed, with two novice participants and one more advanced user who has previous knowledge of the system. Users were given tasks which they tried to full fill using provided tools. Afterward, user feedback was gathered, and additional changes were implemented based on said feedback. Users were then provided with basic tasks to try to execute if they struggled; additional guidance was offered. All tests were executed through a remote session on well-defined system configuration as well as stable database definitions; tests were performed on Pagila [26] data set, whose structure was explained to participants using entity diagram.

4.4.1 Tasks

During usability testing of wizard component following tasks were defined:

Task 1 Tests usability of wizard interface.

1. Open ClueMaker configurator application.
2. Open configuration wizard window from Tools menu.
3. Open configuration file in Documents folder.
4. Check if the database has loaded metadata / if not provide username and password and load metadata.
5. Create new entities Customer, Actor, Film, Address and try to identify other entities that might have relevance.
6. Create relation entities that create a connection between actor and film.
7. Create a relation that connects customer and address.
8. Check created entities in the graph.

Task 2 Test graph interface and SQL query builder. This task is designed to check for the intuitiveness of making changes to the already loaded configuration.

1. Open ClueMaker configurator application.
2. Open configuration file in the Documents folder.
3. Check if the database has loaded metadata / if not provide username and password and load metadata.

4. Create new entity Address thought graph interface.
5. Edit Address entity mapping.
6. Edit SQL to include city and country through provided visual editor.
7. Save configuration.

4.4.2 User 1 - Advanced user

Developer of application ClueMaker that helps clients with modyfying configurations and has extensive knowledge of ClueMaker controls.

4.4.2.1 Task 1

The user had no problem navigating the interface of wizards since these panels were familiar to him. The one problem this user faced was while navigating the graph visualization, here he noticed that controls are different from those currently used by the main ClueMaker application, this was rectified, and controls were unified.

4.4.2.2 Task 2

During testing with this user, the functionality of the visual SQL builder was not yet implemented.

4.4.3 User 2,3 - Novice users

This was the first encounter with ClueMaker for these participants. At first, they needed an explanation of what their goal is and basic controls of the GUI. However, both were students of IT on FIT, so users with technical backgrounds and understood quickly.

4.4.3.1 Task 1

After a brief introduction to user controls, participants had no problem locating the wizard and opening configuration. Users quickly found the create button but were confused by the create dialog and needed to be guided to the suggestion box instead of typing and creating a new entity. Both users then used the search functionality to speed up locating desired entities/relations/relation entities.

User 1 of this group found it odd that already created relations are still offered in context menus, and it seemed there were duplicates without distinguishable differences. Additional filters were included that removed currently used entities based on the name. For relations and relation entities

User 2 thought the overall experience was much more straightforward than if he had to create the configuration by himself but noted that the process

should be even more streamlined with the suggestions based on some heuristic or metric; this feature is planned in future revisions.

4.4.3.2 Task 2

Users used the toolbar, clicked on the create new node button, and created entity without any issues. Both users need to be guided to a tree view of the configuration and shown where mappings reside. This should not be a problem for the more apt user since the process is pretty well documented in the user guide [6]. Using the SQL builder was intuitive for the users.

User 1 - didn't like that the created node entity was not laid out properly and only placed in the top left corner. He thought opening mapping should be more intuitive and straight from graph visualization instead of going through an expanded tree. This is a valid concern that will be addressed in future revisions.

User 2 - user noticed lag when opening mapping editor; this was due to the static import of library JOOQ that needed some time to initialize; this initialization was moved to startup sequence.

Conclusion

At the beginning of this thesis, the author describes the application ClueMaker and ClueMaker Configurator their interaction and functionality; the next part contains an analysis of the current solution for creating configuration as well as market research of competing products, the rest of the thesis deals with design implementation and testing of new modules.

The result of the thesis is three new modules implemented to ClueMaker configurator whose purpose is to guide and help users create a configuration and visualize entities and relations on a graph. Furthermore, to support the detection of suggested entities and their relations new plugin architecture for data sources analyzer was implemented, and a custom visual SQL builder was developed.

The solution implemented in this thesis fulfills all the functional and non-functional requirements, with some needing further refinement, mainly by adding user-facing features like better sorting of suggestions.

Further improvements can be made by implementing additional analyzers with more advanced features like detection of relations based on classification of data, merging similar entities during the analysis process, or detecting relations defined by multiple attributes.

Bibliography

[1] Cluemaker Logo. [Cited 2021-05-05]. Available from: https://cluemaker.com/wp-content/themes/svat/images/ClueMaker_logo_1x.png

[2] *ClueMaker application*. Profinit EU s.r.o., [Cited 2021-05-05]. Available from: https://docs.cluemaker.com/latest/assets/img/70_timeline_1_en.png

[3] Apache NetBeans Logo. July 2018, [Cited 2021-05-05]. Available from: https://commons.wikimedia.org/wiki/File:Apache_NetBeans_Logo.svg

[4] *Apache Maven logo*. Apache Software Foundation, [Cited 2021-05-05]. Available from: https://commons.wikimedia.org/wiki/File:Apache_Maven_logo.svg

[5] I2 Group logo. [Cited 2021-05-05]. Available from: https://upload.wikimedia.org/wikipedia/en/1/11/I2_Group_logo.png

[6] ClueMaker Configurator Guide. [Cited 2021-05-05]. Available from: https://docs.cluemaker.com/latest/en/manual/configurator.html

[7] Vogel, L. *Introduction to Java programming - Tutorial*. vogella GmbH, [Cited 2021-05-05]. Available from: https://www.vogella.com/tutorials/JavaIntroduction/article.html#introduction-to-java

[8] *Java 8*. Oracle Corporation, [Cited 2021-05-05]. Available from: https://www.oracle.com/java/technologies/java8.html

[9] Apache NetBeans Development Version Documentation: APIs Overview. [Cited 2021-05-05]. Available from: http://bits.netbeans.org/dev/javadoc/

[10] Kaspar, D. Visual Library 2.0 - Documentation. [Cited 2021-05-05]. Available from: http://bits.netbeans.org/dev/javadoc/org-netbeans-api-visual/org/netbeans/api/visual/widget/doc-files/documentation.html

[11] Introduction to Apache Maven: A build automation tool for Java projects. Dec. 2019, [Cited 2021-05-05]. Available from: https://www.geeksforgeeks.org/introduction-apache-maven-build-automation-tool-java-projects/

[12] Břešťan, R. Peníze na „analýzy a mediální zastoupení" Okamurovy SPD šly k lidem spojeným s TV Barrandov a Parlamentními listy. Feb. 2018, [Cited 2021-05-05]. Available from: https://hlidacipes.org/penize-analyzy-medialni-zastoupeni-okamurovy-spd-sly-k-lidem-spojenym-tv-barrandov-parlamentnimi-listy/

[13] IBM to Acquire i2 to Accelerate Big Data Analytics to Transform Global Cities. [Cited 2021-05-05]. Available from: https://www-03.ibm.com/press/us/en/pressrelease/35255.wss

[14] *IBM Security i2 Analyst's Notebook - Overview*. IBM, [Cited 2021-05-05]. Available from: https://www.ibm.com/cz-en/products/i2-analysts-notebook

[15] i2 Analyst's Notebook Documentation. [Cited 2021-05-05]. Available from: https://www.ibm.com/docs/en/i2-anb/9.2.3?topic=specification-creating-editing-import

[16] KeyLines - The JavaScript toolkit for graph visualization. Nov. 2020, [Cited 2021-05-05]. Available from: https://cambridge-intelligence.com/keylines/

[17] Tovek. [Cited 2021-05-05]. Available from: https://www.tovek.cz/cs/tovek-tools.html/textbf

[18] Transform Development Services. [Cited 2021-05-05]. Available from: https://www.maltego.com/transform-hub/

[19] in Research-Based User Experience, W. L. 10 Usability Heuristics for User Interface Design. [Cited 2021-05-05]. Available from: https://www.nngroup.com/articles/ten-usability-heuristics/

[20] FasterXML. FasterXML/jackson. [Cited 2021-05-05]. Available from: https://github.com/FasterXML/jackson

[21] Schnitzer, J. jsog/jsog-jackson. [Cited 2021-05-05]. Available from: https://github.com/jsog/jsog-jackson

[22] Martin, R. C. The Clean Code Blog. [Cited 2021-05-05]. Available from: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[23] Villena, K. Simplifying Dependency Inversion Principle (DIP). Sept. 2018, [Cited 2021-05-05]. Available from: https://medium.com/@kedren.villena/simplifying-dependency-inversion-principle-dip-59228122649a

[24] The 5th major version of the programmer-friendly testing framework for Java and the JVM. [Cited 2021-05-05]. Available from: https://junit.org/junit5/

[25] Mockito Framework. [Cited 2021-05-05]. Available from: https://site.mockito.org/

[26] Gündüz, D. Pagila. [Cited 2021-05-05]. Available from: https://github.com/devrimgunduz/pagila

[27] Blackmon, M. H.; Polson, P. G.; et al. Cognitive Walkthrough for the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '02, New York, NY, USA: Association for Computing Machinery, 2002, ISBN 1581134533, p. 463–470, doi:10.1145/503376.503459, [Cited 2021-05-05]. Available from: https://doi.org/10.1145/503376.503459

Appendix A

Acronyms

API Application Programming Interface

CW Cognitive Walkthrough

DB Database

DTO Data transfer object

FR Functional requirement

GUI Graphical user interface

ICO Company identification number

IDE Integrated development environment

JDBC Java Database Connectivity

JOOQ Java Object Oriented Querying

JSOG JavaScript object graph

JSON JavaScript object notation

NR Non-functional requirement

POJO Plain old java object

SQL Structured Query Language

Appendix B

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
exe ................................... the directory with executables
  cluemakerConfigurator.zip ... ZIP with executable of implemtation
  test ................................... the directory with test files
src ...................................... the directory of source codes
  source .................................... implementation sources
  thesis ............. the directory of LaTeX source codes of the thesis
text ......................................... the thesis text directory
  thesis.pdf ......................... the thesis text in PDF format
  diagrams .............................. the directory with diagrams
```