



**CZECH TECHNICAL
UNIVERSITY
IN PRAGUE**

F3

**Faculty of Electrical Engineering
Department of Computer Graphics and Interaction**

Bachelor's Thesis

Organization of master-key systems

Jiří Zahradník

Open Informatics, Computer Games and Graphics

May 2019

Supervisor: Radomír Černocho, MSc., Ph.D.



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Zahradník** Jméno: **Jiří** Osobní číslo: **465916**
Fakulta/ústav: **Fakulta elektrotechnická**
Zadávací katedra/ústav: **Katedra počítačové grafiky a interakce**
Studijní program: **Otevřená informatika**
Studijní obor: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Správa systémů generálního a hlavních klíčů

Název bakalářské práce anglicky:

Organization of master-key systems

Pokyny pro vypracování:

Cílem práce je vytvoření aplikace pro organizaci systémů generálního a hlavních klíčů (SGHK). Tyto systémy se řadí do stromové struktury a jsou uloženy v relační databázi. Kromě samotné správy tohoto stromu (vytváření uzlů, přesuny, přejmenování, ...) je klíčovou operací výběr všech uzlů v podstromu, který exportuje všechny tyto systémy k dalšímu zpracování. Důležitým bodem zadání je tak návrh vhodné datové struktury.

1. Proveďte rešerši literatury k problému reprezentace stromových struktur v relačních databázích. Vyberte 2-3 nejvhodnější reprezentace a porovnejte je z hlediska rychlosti vykonávání operací, přenositelnosti a jednoduchosti implementace.
2. Na základě předchozího kroku vyberte jeden modelovací přístup a ten implementujte. Změřte výkonnost na dodaných datech.
3. Vytvořte aplikaci pro organizaci SGHK do stromové struktury. Kód řádně zdokumentujte a zaveďte automatické testy.

Seznam doporučené literatury:

Literatura:

- [1] Avi Silberschatz, Henry F. Korth, S. Sudarshan (2010). Database System Concepts. McGraw-Hill. ISBN 0-07-352332-1.
- [2] Mike Hillyer (2012). Managing Hierarchical Data in MySQL. Retrieved from URL: <https://explainextended.com/2009/09/24/adjacency-list-vs-nested-sets-postgresql/>
- [3] Quassnoi (2009). Retrieved from URL: Adjacency list vs. nested sets: PostgreSQL
- [4] Mohamed Taman (2014). JavaFX Essentials. Packt Publishing. ISBN 13-9781784398026

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Radomír Černoch, MSc., Ph.D., Intelligent Data Analysis FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

Radomír Černoch, MSc., Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání

Podpis studenta

Acknowledgement / Declaration

I would like to express my gratitude to my supervisor Radomír Černoch, MSc., Ph.D., for the time he spent with me discussing the topic, his patience and all his useful advice that helped me during the whole period of writing this thesis. Furthermore, I would also like to thank my family and friends for the support they provided me with.

I hereby declare I have written this thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis.

In Prague, 20. May 2019

Abstrakt / Abstract

Cílem této práce bylo vytvořit aplikaci pro správu systémů generálního a hlavních klíčů. Tyto systémy jsou uloženy ve stromové struktuře v relační databázi. Z toho důvodu bylo nezbytné navrhnout vhodný databázový model k reprezentaci této struktury. Zváženo bylo několik modelů, které byly testovány a byla změřena rychlost vykonání jednotlivých potřebných databázových operací v rámci těchto modelů.

Poté bylo navrženo a vytvořeno uživatelské rozhraní aplikace. Umožňuje uživateli manipulaci s daty uloženými v databázi. Konkrétně mu dovolí vytvářet nové uzly, přesouvat je, přejmenovávat, mazat a nastavovat uzlům uzávěry klíčů. Pak byl implementován model se všemi nezbytnými databázovými dotazy.

Nakonec byla implementace otestována pomocí testování použitelnosti a za použití jednotkových testů. Výsledkem této práce je plně funkční aplikace pro interakci uživatele s databází.

Klíčová slova: Stromová Struktura, SQL, Databáze, GUI, Aplikace

Překlad titulu: Správa systémů generálního a hlavních klíčů

The goal of this thesis was to create an application for the master-key systems' organisation. These systems are stored in a tree-like taxonomy in a relational database. Therefore it was essential to devise an appropriate database model to represent such structure. A few models were considered, tested, and the speed of the execution of the required database operations within each model was measured.

Afterwards, the graphical user interface for the application was designed and created. It allows the user to manipulate the data stored in the database. To be precise, it enables them to create new nodes, move them, rename them, delete them and set them key shapes. Then the chosen model, with all its necessary queries, was implemented.

Finally, the implementation was tested via usability testing and using unit tests. The result of this thesis is a fully functioning application for user-database interaction.

Keywords: Tree Structure, SQL, Database, GUI, Application

/ Contents

1 Introduction	1
2 Theory	3
2.1 Relational Database	3
2.2 JavaFX 2	4
3 Requirements	7
3.1 Functional requirements	7
3.2 Non-functional requirements	8
4 Database Models	9
4.1 Nested Set Model	9
4.1.1 Database Model	9
4.1.2 Set of Operations	9
4.2 Adjacency List Model	12
4.2.1 Database Model	12
4.2.2 Set of Operations	12
4.3 Materialized Path Model	14
4.3.1 Database Model	14
4.3.2 Set of Operations	14
4.4 Models Comparison	16
4.4.1 Theoretical Comparison	16
4.4.2 Experimental Comparison (Speed Measurement)	18
4.4.3 Choosing a Model	20
4.4.4 Additional Indexing	20
5 Application Design	23
5.1 Design	23
5.1.1 GUI	23
5.1.2 GUI Package	39
5.2 SQL Package	40
5.2.1 Transaction Management	40
5.2.2 TableInitializer	41
5.2.3 Adder	41
5.2.4 Renamer	41
5.2.5 Key Renamer	41
5.2.6 Selector	41
5.2.7 Table cleaner	41
5.3 Unit Tests	42
6 Usability Testing	43
6.1 User Testing	43
6.1.1 Testing process	44
6.2 Findings and revision	45
7 Conclusion	47
References	48

/ Figures

1.1.	Mechanical Pin Tumbler Lock...	1
5.1.	Application window	24
5.2.	Application start failed dialogue	24
5.3.	Application initialization UML Diagram.....	25
5.4.	Menu.....	26
5.5.	Failed key shape display dialogue	26
5.6.	Failed addition dialogue	27
5.7.	Addition UML Diagram	28
5.8.	Failed deletion dialogue	29
5.9.	Deletion UML Diagram.....	30
5.10.	Failed transfer dialogue	31
5.11.	Merge or Paste Folders dialogue	31
5.12.	Paste UML Diagram	32
5.13.	Rename textfield	33
5.14.	Wrong name tooltip.....	33
5.17.	Long name tooltip	33
5.15.	Wrong name warning	34
5.16.	Empty name warning	34
5.18.	Long name warning	34
5.19.	Merge folders confirmation	35
5.20.	Failed merge dialogue	35
5.21.	Failed rename dialogue	35
5.22.	Rename UML Diagram	36
5.23.	Key shape set dialogue.....	37
5.24.	Wrong key shape tooltip.....	37
5.25.	Long key shape tooltip.....	37
5.26.	Key Shape Set UML Diagram .	38
5.27.	Key shape delete dialogue	39
6.1.	Project deletion confirmation dialogue.....	46

Chapter 1

Introduction

Even though some of us might not realise it, we encounter tree structures on everyday basis. They are extensively used because they are native and most of all, intuitive. Often there is a need to group some items or some data hierarchically [1], meaning that some are ‘above’, ‘below’ or ‘at the same level as’ others. The best example from today’s world is online stores. More specifically, the way they display their merchandise. It is usually divided into categories such as Electronics, Health, Sport, Accessories or Hobbies. Under each category, we can see a more specific classification of goods. For example, under Electronics, we could find PCs, Laptops, Mobile phones, Components or Printers.

However, what many people do not realise is, that mechanical keys and locks are also hierarchically categorised. This thesis deals with a particular tree structure in the domain of mechanical keys and locks. Every key has its unique look defined by the number of notches, shape of the head, tip bevel etc. A combination of these attributes is called a platform. Within the platform, there is a profile map which determines the grooves cut in the key. So the key hierarchy is determined already by the production process [2].

The most common mechanical lock we use is a pin tumbler lock (Figure 1.1). There are pins and a tumbler inside the lock. The pins prevent the tumbler of the lock from rotating. Only when a corresponding key is inserted into this lock’s tumbler, the pins are aligned with the edge of the tumbler, allowing it to rotate.

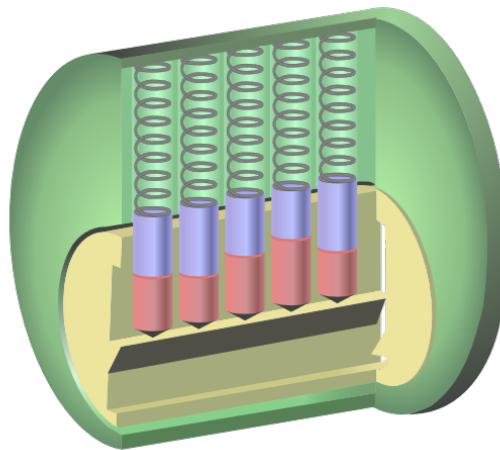


Figure 1.1. Mechanical Pin Tumbler Lock with pins (blue) and a tumbler (yellow), retrieved from [3]

Aside from standard keys which open only one or a few doors within a particular system, there is a so-called general key [4] which can unlock all the locks within the system. The general key has the least amount of grooves cut in it. It is essential not

Chapter 2

Theory

In order to fully understand all the database operations presented in this thesis, the reader is required to have some substantial knowledge of the relational database theory. We are going to go through some of the fundamental operations nevertheless in case the reader was not familiar with the relational database terminology.

We are also going to look into the JavaFX platform architecture and design briefly.

2.1 Relational Database

By the term, Relational Database [5, p. 37] is usually meant not only the database itself but also its software implementation. A relational database management system is needed to enable the user to have control over the database. There are many of such systems including PostgreSQL, MySQL or Apache Derby.

The database itself consists of tables [5, pp. 39-46], which store data. Each table is composed of columns (attributes) and rows (records, entries). Each attribute has a specific data type. The rows then serve as data holders.

Every table should have a *Primary Key*. It is a unique identifier used to identify a data record. One or more columns at once may form a primary key. The value of the primary key must be defined (i.e. it can not be NULL). Commonly used are artificial keys (IDs), which are often automatically generated for each entry.

A similar role is played by a *Foreign Key*. It determines the dependencies between data from different tables.

The data in the database are accessed and maintained by a series of database commands and operations called queries [5, pp. 48-52]. The most used query for data manipulation is the *SELECT* statement. It allows us to retrieve data from a specific table or tables. It has a few optional clauses such as the *WHERE* clause that lets us specify the rows we wish to retrieve.

However, before we get to fetch data, we need to make a table to store it. We can achieve this by using the data definition *CREATE TABLE* command [5, pp. 60-62]. There we define all the columns we want to have in our table along with the appropriate data types, keys and constraints. We fill the table with data by using the *INSERT* statement.

If we wish to change some data in a table, we use the *UPDATE* command. We can update all rows within a table or specify the rows using a condition (*WHERE*). To get rid of some of the stored data, we make use of the *DELETE* statement [5, p. 98]. It removes one or more records from a table.

The results of multiple queries can be combined into a single result by utilising database set operations. One of those is the *EXCEPT* operator. It takes all rows retrieved by a first query and returns only those that do not appear in a result set of rows retrieved by a second query.

between objects. When one object is somehow changed, the changes are automatically reflected in the other object. More specifically, the data modified in the model alters the view automatically.

Chapter 3

Requirements

The client has given us their job description.

The client calculates the shapes of keys for certain projects. Every key shape is usually a 4-7 digit number. The client needs to store these calculated key shapes along with the project names. The client divides the projects into a few categories based on the key platform, key profile, key type and others. When calculating a key shape, it usually helps to look at other already calculated shapes from the same or similar category.

From this explanation, we devised a structure needed for the storage of the calculated key shapes. We also devised an application allowing the user to interact with the stored data. This application should serve as a tool used for structuring and creating projects. Those projects can then be worked out using another application, which is already in use.

3.1 Functional requirements

The tree structure is fitting for the key shape data storage. That way, the projects containing key shapes can be divided into categories (where categories are tree nodes). The projects with the key shapes would be stored as leafs of the tree structure.

Based on the amount of data, there would be almost six thousand nodes in the tree. The degree of the tree would be eight hundred forty-six, which is the highest degree of a node in the tree, and the maximal depth would be five. However, there is not given any maximal degree a node can have. We can see the tree is not deep but wide instead.

The client could also make use of an application which would allow them to interact with the structure containing the data stored in a database.

Below is a list of suggested criteria laid down in order to judge the suitability of the application [12].

- The application should be easy to use and intuitive.
- The application should include a simple graphical user interface.
- The user should be able to see at least some parts of the tree structure.
- The user should be able to:
 - rename tree nodes
 - remove tree nodes
 - add new tree nodes
 - move tree nodes.

- Duplicate names are allowed.
- The user should be able to set a key shape in leaf nodes of the tree structure. The key shape is usually a 5 digit number.
- The application should automatically store all changes made by the user into the database.
- The user should be able to see all the key shapes set within a particular part of the tree.
- The application should display the key shapes fast.
- The tree structure management should reflect these demands.

3.2 Non-functional requirements

The client then specified some non-functional criteria judging the operation of the application.

- The application should run on OS Windows 10.
- The application should be created with Java programming language.
- The application should be easily executable (preferably with a single file).
- The application should be highly reliable.
- The application should resolve all error conditions.
- The application response time should be less than two seconds for single operations.
- The application should be supported by Derby database layer.
- The application should use multithreading model.

Chapter 4

Database Models

Let us now go through the models we considered to represent the key hierarchy. First, explain each query we could use in the final implementation, then compare the models and choose the one most suitable for our application.

Most of the queries require some input parameters we get from the user. Those constants are marked in the queries with {}.

The inspiration for this chapter was drawn from [1], [13], [14], which describe the analysed models. It helped me to understand the models and also advised me to divide the sections into smaller sections, each dealing with the individual queries. However, I did not copy any of the queries. I devised every query in this chapter myself.

In the SQL queries, we are going to be using operands $+=$ and $-=$. SQL does not support the use of such operands; however, we are going to use them in this chapter for clarity.

4.1 Nested Set Model

In this model, we look at the tree structure as if it were nested containers. By containers we mean, each node in the tree has an interval containing all its descendants.

4.1.1 Database Model

The Tree table is created using a standard *CREATE* command.

```
CREATE TABLE Tree (  
    id INTEGER PRIMARY KEY,  
    lft INTEGER NOT NULL UNIQUE,  
    rght INTEGER NOT NULL UNIQUE  
);
```

The reason we do not use the names 'left' and 'right' is because they are reserved words in MySQL. The meaning of these two integers is to form an interval containing all the child nodes of their parent node (if it has any). We can also say, $lft = \min(child.lft) - 1$ and $rght = \max(child.rght) + 1$. The *rght* value is always greater than the *lft* value. To the root always applies $lft = 1$ and to every leaf node in the tree applies $rght = lft + 1$. All nodes have exactly $(rght - lft - 1) / 2$ child nodes and there are $root.rght / 2$ nodes in the tree. The left and right values are assigned via a pre-order tree traversal. Going from left to right, we set the left value and descend to the child nodes before setting the right value while always incrementing by one.

4.1.2 Set of Operations

The main advantages of using this structure are the avoidance of recursion and the usage of as few queries as possible.

4.1.2.1 Adding New Nodes

When adding a new node, space must be provided for its left and right values. Meaning, each node to the right of the added node must increment its left and right values by two. After that, a simple *INSERT* can be performed.

```
UPDATE Tree
SET lft += 2
WHERE lft >= {parent.rght};
UPDATE Tree
SET rght += 2
WHERE rght >= {parent.rght};
INSERT INTO Tree (lft, rght)
VALUES ({parent.rght}, {parent.rght} + 1);
```

The left and right values of the parent node used in the insertion are pre-update. The new node is added under the *parent* node.

4.1.2.2 Deleting a Leaf Node

In order to delete a leaf node (*del*), we apply the same method as if adding a node, only in reverse.

```
DELETE FROM Tree
WHERE id = {del.id};
UPDATE Tree
SET lft -= 2
WHERE lft > {del.rght};
UPDATE Tree
SET rght -= 2
WHERE rght > {del.rght};
```

We have to delete the node first, so there are no conflicts. If we removed the node after the updates, we would probably end up with some nodes having the same left and right values as other nodes (duplicates). That would result in an integrity violation error due to the left and right columns being unique.

4.1.2.3 Deleting a Subtree

To delete a subtree means to delete a node (*subroot*) and transitively all its child nodes. We can delete all child nodes at once using the values left and right, thus generalising the previous query. Since we know, there are $(rght - lft + 1) / 2$ nodes in a subtree, each with 2 values (left and right), there must be precisely $rght - lft + 1$ values altogether.

```
DELETE FROM Tree
WHERE lft >= {subroot.lft} AND rght <= {subroot.rght};
UPDATE Tree
SET lft -= ({subroot.rght} - {subroot.lft} + 1)
WHERE lft > {subroot.rght};
UPDATE Tree
SET rght -= ({subroot.rght} - {subroot.lft} + 1)
WHERE rght > {subroot.rght};
```

4.1.2.4 Selecting Descendants of a Node

Here we use the left and right properties as an interval containing all the child nodes of a parent node.

```
SELECT * FROM Tree
WHERE lft >= {parent.lft} AND rght <= {parent.rght};
```

4.1.2.5 Finding All the Leaf Nodes

In this query, we need to recall leaf nodes are those that have a difference of one between their left and right values.

```
SELECT * FROM Tree
WHERE right = left + 1;
```

4.1.2.6 Finding All the Leaf Nodes of a Subtree

The same approach can be applied to a subtree and its leaves.

```
SELECT * FROM Tree
WHERE right = left + 1
AND left >= {subroot.left} AND right <= {subroot.right};
```

4.1.2.7 Transferring a Subtree

In order to relocate a subtree, we must update nodes influenced by the transfer. We also need to change the values within the subtree itself. There are three types of transfer. It is necessary to differentiate between them. There is a transfer to the right ($\{subroot.right\} < \{newparent.right\}$) and a transfer to the left ($\{subroot.left\} > \{newparent.right\}$). Here we are going to take a look at the transfer to the right since they are both analogous. The third transfer is moving a subtree a few levels up in the tree. It works the same way as the transfer to the right.

```
UPDATE Tree
SET left =
CASE
  WHEN left BETWEEN {subroot.right} + 1 AND {newparent.right} - 1
  THEN left - ({subroot.right} - {subroot.left} + 1)
  WHEN left BETWEEN {subroot.left} AND {subroot.right}
  THEN left + {newparent.right} - {subroot.right} - 1
END
WHERE left BETWEEN {subroot.left} AND {newparent.right} - 1;

UPDATE Tree
SET right =
CASE
  WHEN right BETWEEN {subroot.right} + 1 AND {newparent.right} - 1
  THEN right - ({subroot.right} - {subroot.left} + 1)
  WHEN right BETWEEN {subroot.left} AND {subroot.right}
  THEN right + {newparent.right} - {subroot.right} - 1
END
WHERE right BETWEEN {subroot.left} AND {newparent.right} - 1;
```

The first *WHEN* clause handles the part of the tree between the subtree's original position and its destination node, the second only the subtree.

The transfer to the left is performed in a similar manner. Specifically, the interval of nodes the transfer affects is different and also the left and right values of the subtree are decreased instead of increased.

4.1.2.8 Finding Direct Descendants of a Node

The crucial thing here is to realise that we can divide a subtree into three levels (*lev1*, *lev2* and *lev3*). Level one would be the subroot *node*, level two all its descendants and level three their descendants. We can achieve this through a self-join (table is joined

with itself). First, we select all descendants of the subroot. Then we exclude the level three ones, thus leaving us with only the direct descendants of the *node*.

```
SELECT lev2.id, lev2.lft, lev2.rght
FROM Tree AS lev1, Tree AS lev2
WHERE lev1.id = {node.id}
      AND (lev1.lft < lev2.lft AND lev1.rght > lev2.rght)
EXCEPT
SELECT lev3.id, lev3.lft, lev3.rght
FROM Tree as lev1, Tree as lev2, Tree as lev3
WHERE lev1.id = {node.id}
      AND (lev1.lft < lev2.lft AND lev1.rght > lev2.rght)
      AND (lev2.lft < lev3.lft AND lev2.rght > lev3.rght);
```

4.2 Adjacency List Model

4.2.1 Database Model

Adjacency list model is the most simplistic one. The column *parent* in the table is a foreign key referencing the id of every node's parent.

```
CREATE TABLE Tree (
  id INTEGER PRIMARY KEY,
  parent INTEGER REFERENCES Tree(id) ON DELETE CASCADE
);
```

The root has a NULL value for its parent. Since the only properties of a node are its id and parent id, there is nothing we can deduce about the rest of the tree based on just one node.

We do not assume any change of values within the *id* column. Therefore, there is no need for an *ON UPDATE* [5, p. 133] clause in the definition of the *parent* column.

4.2.2 Set of Operations

This model is straightforward to understand and in most cases, even to implement. However, there are some queries where we can not avoid recursion.

4.2.2.1 Adding New Nodes

There is no need to update any nodes when adding a new one under a *parent* node. A plain insert is sufficient.

```
INSERT INTO Tree (parent)
VALUES ({parent.id});
```

Here, we do not insert the *id* value because we expect it to be generated automatically.

4.2.2.2 Deleting a Leaf Node

Deleting a leaf node (*del*) in this model is simple. Since no references are pointing to any leaf node, a simple delete can be executed.

```
DELETE FROM Tree
WHERE id = {del.id};
```

4.2.2.3 Deleting a Subtree

Thanks to the *parent* foreign key, we can just delete a single *subroot* node. All other nodes which are referencing this node's id are also deleted. It holds recursively for their child nodes as well.

```
DELETE FROM Tree
WHERE id = {subroot.id};
```

4.2.2.4 Selecting Descendants of a Node

In this query, we need to use recursion [5, pp. 190-192]. To do so, we create a temporary table *Subtree*. First we insert the *subroot* node. The insertion is the initiation of the table. Then we add all direct child nodes of each node in the table. This step is automatically repeated every time the table is updated.

```
WITH RECURSIVE Subtree AS(
  SELECT * FROM Tree
  WHERE id = {subroot.id}
  UNION
  SELECT Tree.* FROM Tree
  JOIN
  Subtree
  ON Subtree.id = Tree.parent
) SELECT * FROM Subtree;
```

4.2.2.5 Finding All the Leaf Nodes

The main characteristic of a leaf node is that it is not parental to any other node. In other words, there is no node in the tree that has a leaf node for its parent.

```
SELECT * FROM Tree
WHERE id NOT IN(
  SELECT DISTINCT parent FROM Tree
);
```

4.2.2.6 Finding All the Leaf Nodes of a Subtree

This is the combination of selecting descendants of a node (selecting a subtree) and finding leaf nodes of a tree.

```
WITH RECURSIVE Subtree AS(
  SELECT * FROM Tree
  WHERE id = {subroot.id}
  UNION
  SELECT Tree.* FROM Tree
  JOIN
  Subtree
  ON Subtree.id = Tree.parent
) SELECT * FROM Subtree
WHERE id NOT IN(
  SELECT DISTINCT parent FROM Subtree
);
```

4.2.2.7 Transferring a Subtree

When transferring a subtree, we only need to redirect the pointer to the *parent* node of the *subroot* node to a *newParent* node.

```
UPDATE Tree
SET parent = {newParent.id}
WHERE id = {subroot.id};
```

4.2.2.8 Finding Direct Descendants of a Node

Direct descendants are those that have the *parent* node's id in their parent column. We can get those by using a simple where clause.

```
SELECT * FROM Tree
WHERE parent = {parent.id};
```

4.3 Materialized Path Model

In this model, we store every ordered path. This way, we get a path to the root node for every node in the tree.

4.3.1 Database Model

Each database table entry consists of a node id and its predecessor. For each different node, there is one entry for every path from the predecessor node to the node itself. We also included reflexive paths from a node to itself to simplify queries.

```
CREATE TABLE Tree (
    id INTEGER NOT NULL,
    pred INTEGER NOT NULL
);
```

There are a few characteristics of this model we can use to make each operation easier.

4.3.2 Set of Operations

When creating the queries in this model, we have to keep in mind that every node's id appears multiple times in the table except for the root's id.

4.3.2.1 Adding New Nodes

We want to add a *new* node under a *parent* node. The *new* node is going to have the same predecessors as the *parent* node including the *parent* itself. Therefore we only need to select the predecessors of the *parent* node and insert them into the *Tree* table along with a new id.

```
INSERT INTO Tree (id, pred)
SELECT {idNew}, pred
FROM Tree
WHERE id = {parent.id}
UNION
SELECT {idNew}, {idNew};
```

We then also need to add a self reference from the *added* node to the *added* node.

4.3.2.2 Deleting a Leaf Node

We know that a leaf node has only one entry in the *Tree* table. A simple delete will suffice.

```
DELETE FROM Tree
WHERE id = {del.id};
```

4.3.2.3 Deleting a Subtree

When deleting a whole *subtree*, we first select all descendants of a *subroot* node and then delete all their entries from the table.

```
DELETE FROM Tree
WHERE id IN (
  SELECT id FROM Tree
  WHERE pred = {subroot.id}
);
```

4.3.2.4 Selecting Descendants of a Node

We already mentioned this query in the previous one. For every node of a subtree holds that one of its predecessors is the *subroot* node.

```
SELECT id FROM Tree
WHERE pred = {subroot.id};
```

This way, we select the *subroot* node as well. To avoid that we could add a specifying statement to the *WHERE* clause (*AND id != subroot.id*).

4.3.2.5 Finding All the Leaf Nodes

For every leaf node holds, it is a predecessor to only one node, itself.

```
SELECT pred FROM Tree
GROUP BY pred
HAVING COUNT(pred) = 1
```

This query does exactly that. It counts the occurrences of each node being a predecessor and selects only those counted once.

4.3.2.6 Finding All the Leaf Nodes of a Subtree

This query combines the previous two - selecting all descendants and finding all leaf nodes.

```
SELECT id FROM Tree
WHERE pred = {subroot.id}
AND id IN (
  SELECT pred FROM Tree
  GROUP BY pred
  HAVING COUNT(pred) = 1
);
```

4.3.2.7 Transferring a Subtree

To transfer a *subtree* we first need to delete all paths leading from the *subtree* to the predecessors of the *subtree* (predecessors of the *parent* node).

```
DELETE FROM Tree
WHERE id IN (
  SELECT id FROM Tree
  WHERE pred = {subroot.id}
) AND pred IN (
  SELECT pred FROM Tree
  WHERE id = {parentOld.id}
);
```

Now we need to add new predecessors to the subtree which come into existence by relocating the subtree. These are the predecessors of the new *parent* node.

```
INSERT INTO Tree (id, pred)
  SELECT * FROM
  (SELECT id FROM Tree
   WHERE pred = {subtree.id}) AS X
  JOIN
  (SELECT pred FROM Tree
   WHERE id = {parentNew.id}) AS Y
  ON 1 = 1;
```

We have to join two selects where the first selects the subtree and the second selects the predecessors of the node under which we transfer the subtree. We want to combine every node of the subtree with every predecessor of the new parent node. This specific type of join is called a cross join, and it is a Cartesian product of the selects. To join the selects correctly, we must name the subselects and also state when the rows retrieved by the select should join using *ON*. Here we want them to join every time, so we put $1 = 1$. We prefer this to the *CROSS JOIN* because it is a universal approach.

4.3.2.8 Finding Direct Descendants of a Node

We can find direct descendants of a *subroot* node by selecting all descendants and excluding the ones that have other descendants of the same *subroot* for predecessors.

```
SELECT id FROM Tree WHERE pred = {subroot.id} AND id != {subroot.id}
EXCEPT
SELECT id FROM Tree WHERE id != pred AND pred IN(
SELECT id FROM Tree WHERE pred = {subroot.id} AND id != {subroot.id});
```

4.4 Models Comparison

Only a couple of queries defined in the Database Models chapter 4 are going to determine our choice of the model. The reason is, only some are used often and some rarely in our application.

4.4.1 Theoretical Comparison

In this section, we are going to compare all three models theoretically. We look at the models and determine which of the queries should be faster in which model and why.

4.4.1.1 Addition Comparison

In the Nested Set model, the addition of a node into the tree structure may turn out to be not as fast as expected. With every addition, we must renumber all nodes to the right from the node we are adding to make a ‘space’ for it. It is still almost instantaneous if the nodes are added on the right side of the tree, but we might feel the drawback when adding a considerable amount of nodes at once anywhere.

However, it is a common practice [15], that we would recognise many nodes are being added in an instance, add the nodes without numbering and then traverse the tree, numbering every node’s properties in the process. To do this, we would need to have a *parent id* column in our table.

When looking at the Adjacency List model, we can see right away; the addition is going to be nearly instantaneous, no matter the amount of data. Since we are only adding new rows in the table, we can count on this operation being fast.

The node addition in the Materialized Path model should be slower than the Adjacency List addition because we insert more data which we also need to retrieve from the table first. The amount of data we insert depends on the depth in which we add the node.

We need to keep in mind that we are never going to add more than one node at a time. Therefore we are not going to choose the database model based on this operation.

■ 4.4.1.2 Subtree Leaves Find Comparison

It seems as if the Nested Set model was designed specifically for this kind of operation. All we do, when selecting the leaf nodes is, we go through the table and filter data using the *WHERE* clause. This operation is going to be done rapidly.

The Adjacency List model, on the other hand, is not made to deal with this sort of requests. Here we are forced to use recursion. Diving deeper into the tree with each iteration and thus adding more rows into the temporary table which can eventually even contain all the nodes from the database. This query is extremely complex, especially with deep tree structures. Another drawback is that not all database systems can deal with this form of recursion.

The disadvantage of using Materialized Path model is that we use the *group by* clause, which has to sort the table first. We also use the intersection of two sets, thus slowing down the query even more.

This query is the most important one, and it seems that the Nested Set model is the best one to use if we want to execute the query fast.

■ 4.4.1.3 Transfer Comparison

The transfer is somewhat similar to the addition of a node in the Nested Set model. We are still updating a substantial part of the whole tree. Only this time, we are not just increasing the left and right values but decreasing as well.

In the Adjacency List model, the transfer is similar to addition as well. All we have to do is redirect the pointer of a node from its current parent (*parent* id attribute) to a different parent. It is only a simple update of one row in the table, which should be almost just as fast as inserting it.

The Materialized Path model handles node transfer by deleting a part of the tree that is supposed to be transferred and then inserting it again but somewhere else (with different predecessors). This in itself would not be much slower than the Adjacency list addition. However, we have to use the intersection of selects multiple times, which is going to slow down the query. The joining of two selects after that should not be a problem.

The Adjacency model is the best one to use when transferring a subtree. This operation is, however, most likely never going to be used (or only exceptionally). The problem is before we even allow a subtree to be transferred, we need to check if we are not transferring it under itself. The Nested Set model handles this check the best by comparing two values. In the Adjacency List model, we have to use recursion, and in the Materialized Path model a simple *select*. Despite that, the Adjacency List model still comes out on top.

■ 4.4.1.4 Direct Descendants Find Comparison

The *JOIN* operation is expensive when it comes to time complexity as well as memory consumption. We have to use it more than once when finding nodes one level deeper in the tree with the Nested Set model nevertheless. So many times, as is the depth of the

subtree to be precise. That is why this operation is going to cost us some computing time.

Again, thanks to the *parent* attribute, the Adjacency List model deals with this operation effortlessly. All we need to do is one simple select.

In the Materialized Path model, we only have to use a few *selects*; however, their intersection and difference (relative complement) are going to slow down the query noticeably.

The Adjacency model is the best to use in case of finding direct descendants of a node. We are, however, most likely not going to mind if the execution of this operations takes extra time because of the way we are going to be loading the data (Lazy loading 5.1).

■ 4.4.2 Experimental Comparison (Speed Measurement)

After we implemented the Nested Set, Adjacency List and Materialized Path model queries, we tested them on real data and timed them. For the purpose of testing, we used a database created in a folder on a hard drive. We are going to go through the results of the speed testing of some of the queries. Because we implemented these tests in java, there might be different constructs used for every operation. Every operation's speed test is implemented the way it would be in the final application (for example we have to create objects and store them in lists), and the way each operation would be implemented is a little bit different. Therefore we can only compare the measured times within a single operation speed test.

■ 4.4.2.1 Addition Speed Test

There are 5238 nodes in the test tree. We added them one by one into the database.

Regarding the Nested set model, the addition time was 5 minutes and 30 seconds. This means it took 63 milliseconds for a single node to be added. The more nodes there are in the tree, the more time it is going to take to add one. The time it took for this query to execute might be affected by the order in which the nodes are added. The worst case is when we add every node to the left side of the tree and need to renumber every other node in the table.

The time it took for all nodes to be added was 5 seconds when using the Adjacency List model, which means one node was added every millisecond. The time should remain consistent no matter the order in which we add the nodes into the tree.

The addition of Materialized Path model was indeed slower than the one of the Adjacency List model. It took 68 seconds for all the 5238 nodes to be added which makes 13 milliseconds per a single node.

Database model	Total time	Time per one Node
Nested Set	330s	63ms
Adjacency List	5s	1ms
Materialized Path	68s	13ms

Table 4.1. Addition performance times

■ 4.4.2.2 Subtree Leaves Find Speed Test

The operation we are most interested in is finding all leaf nodes in a subtree. For this test, we chose ten thousand nodes from the tree at random. These nodes were the subroots of the subtrees in which we were finding the leaves.

In the Nested Set model, the duration of this query was 410 milliseconds. It means leaf nodes of one subtree were found in 598 microseconds.

The Adjacency List model found the leaf nodes of a subtree in much longer time - 3 seconds, which makes it 315 microseconds per node. We expected this model to be much slower here because of the recursion and the intersection of selects.

The Materialized Path model handles this operation quite well. However, it is still slower than the Nested Set model with a total time of 750 milliseconds and 75 microseconds per single test.

Database model	Total time	Time per one test
Nested Set	410ms	41 μ s
Adjacency List	3s	315 μ s
Materialized Path	750ms	75 μ s

Table 4.2. Leaf nodes find performance times

4.4.2.3 Transfer Speed Test

The transfer of a subtree is probably not going to be used at all. The client did not mention they would transfer created projects at all. We should, however, test it nonetheless, because it is a feature we believe our application should have.

The transfer of a random thousand subtrees took 2 minutes and 41 seconds in the Nested Set model, which means the transfer of a subtree takes 161 milliseconds. As stated earlier, these measurements are including the checking whether we are trying to transfer the subtree under one of its nodes or not.

This check considerable slowed down the subtree transfer in the Adjacency List, because it took 2 seconds to transfer thousand subtrees and 2 milliseconds to transfer one.

The transfer speed of the Material Path model is comparable to the Nested Set model speed but is a bit faster. It took 2 minutes and 18 seconds to transfer all thousand subtrees and on average 138 milliseconds to transfer one.

Database model	Total time	Time per one transfer
Nested Set	161s	161ms
Adjacency List	2s	2ms
Materialized Path	138s	138ms

Table 4.3. Subtree transfer performance times

4.4.2.4 Direct Descendants Find Speed Test

The last query we need to time is finding direct descendants of a node. This operation is one of the more important ones for our application, but it is not going to slow it down since it is going to be called sooner than the user requests due to our specific implementation of lazy load (see 5.1). Due to the nature of our test data set, we expect this query to take longer. The tree topology is not deep but wide instead.

We tested the query on ten thousand randomly selected nodes with the execution time being 22 seconds, which is 2 milliseconds per node for the Nested Set model.

The Adjacency List model is exceptionally fast when it comes to this operation. It took 80 milliseconds to find direct descendants of ten thousand nodes and 8 microseconds to find direct descendants of a single node.

The speed of the Materialized Path model was again somewhere between the other two models with total execution time of 2 seconds and 236 microseconds per node.

Database model	Total time	Time per one node
Nested Set	22s	2ms
Adjacency List	80ms	8 μ s
Materialized Path	2s	236 μ s

Table 4.4. Direct descendants find performance times

4.4.3 Choosing a Model

Now, let us decide which of these models is going to suit our purpose the best and contribute to our application the most. The addition of a node is not going to be frequent. We are most certainly not going to add plenty of data in an instant. The Adjacency List model, however, still handles this operation a little bit better than the other models.

Finding leaves of a subtree is probably going to be the deciding factor since this procedure is going to be called whenever the user clicks on a node. It is apparent, the Nested Set model it at a considerable advantage over the other models.

The transfer of a subtree is probably not going to happen in practice at all, but we included it in the decision-making process just in case. The Adjacency List model comes much better out of this one. However, this operation is not important enough to tip the scales in its favour.

We can not deny the benefit of retrieving direct descendants. We are going to need to be able to do this in the implementation of our application. We surely would not want to retrieve all the data at the application start. The Adjacency List model has the upper hand when considering this query.

All things considered, we are going to go with the Nested Set model. Most of all because it is much better at retrieving leaf nodes, which is going to be a regularly used operation. Also, the drawbacks of this model are only minor, and it is not much slower regarding the other operations than the other models. This model is also universal. The Adjacency List model, on the other hand, uses recursion, which is something not all database systems can deal with.

4.4.4 Additional Indexing

Now that we have chosen a model to implement, we need to find out if we can make it perform any faster with additional indexing. Here we are going to test the speed of the queries again, but this time we try indexing some of the table columns. We would test the speed of the queries multiple times and put an index on different columns every time, however, in this instance, it only makes sense to put an index on the *lft* and *right* columns.

4.4.4.1 Indexed Addition Speed Test

We already know how long it takes to add nodes into the tree without indexing from the previous chapter. This time we add the nodes with an index on the *lft* and *right* columns. We should not forget, the *id* column is indexed by default because it is the *PRIMARY KEY*. This time the addition was more than twice as slow. The exact time was 11 minutes and 40 seconds. That makes 133 milliseconds per one node.

The conclusion for us would be that indices can, in fact, slow down the addition of nodes (or any operation in fact), because there are many updates to be executed. In our application, only a few nodes at the time are going to be added, so we should not worry too much about this. If we try to add only a thousand nodes into the database,

we get much more encouraging results. It took only 54 seconds, which is precisely 54 milliseconds per node.

■ 4.4.4.2 Indexed Subtree Leaves Find Speed Test

We do not suspect indexing would help in this case. In the *WHERE* clause we want *right* to be equal to *lft + 1*. The index does not know how to deal with the '*+ 1*' part. Not even the whole expression itself is not a constant; therefore, an index should most likely not help.

After placing an index on the *lft* and *right* columns, this query was executed in 483 milliseconds with leaves of a subroot found in 48 microseconds. We can not see almost any change here. Because the nodes are selected randomly, there is no way of determining whether the index sped up the execution or not.

This query is executed so quickly; there is apparently no need for an index. It might even seem as if the index would slow this query down as well.

■ 4.4.4.3 Indexed Transfer Speed Test

There is much adding a subtracting going on in this query. Because of that, the index is not going to help, and since we update the entries in the table as well, the additional indexing might slow the execution down.

When we use an index for the *lft* and *right* columns, we get to 5 minutes and 17 seconds. That is 317 milliseconds per transfer of a subtree. We can see a significant deceleration of the execution time.

■ 4.4.4.4 Indexed Direct Descendants Find Speed Test

With the *lft* and *right* columns indexed, we get the execution time of finding direct descendants 21 seconds. This is almost the same as without the index. The small change could be again caused only by the nodes being selected at random.

■ 4.4.4.5 Indexing Conclusion

To conclude the testing, we must say it turned out as expected. The execution times usually doubled and in best cases, remained roughly the same. The tests confirmed adding indices on column tables does not always have to be the right choice. We are therefore not going to use any additional indices in order to save space in the database and not slow the query execution down.



Chapter 5

Application Design

We wanted to create an application that would allow the user to manage master-key systems. Such systems are organised in a tree-like taxonomy which is stored in a database. We needed to allow the user to manipulate the data in all sorts of ways. To be precise, it is mainly creating new nodes, moving them, renaming them etc.

In this chapter, we are going to go through the implementation of such an application, initially in general and then more specifically through the most important classes and methods in the project. The classes are separated into packages. We wanted the SQL classes to be independent on the type of the application and enable us to test them efficiently. This way, we could separate the View layer from the Model. The language we chose to implement the application in is Java. The reason for it is, it is a well-established and widespread language, accessible also due to its portability. Also, it is one of the client's requirements.

A UML activity diagram [16, pp. 285-308] was created for every non-trivial function of the application using an online open platform software [17]. The diagrams were made in order to visualise the way the application should work. They cover exception handling and transaction management (more on that in chapter 5.2) just as well as all the interface elements described in this chapter.

5.1 Design

First of all, we had to design the GUI and choose suitable UI components. Then write all the SQL queries. We used one class per one type of query. Those classes are supposed to be called from threads. Since the thread class itself implements the Runnable interface [18], we wanted to make our SQL classes runnable as well and pass their instances to the threads. That way, the user interface would not freeze if the database was slow to respond (asynchronous update [5, p. 390]). We had to be sure the SQL queries are correct, so we introduced Unit tests (see chapter 5.3) and tested every query thoroughly. Then we connected the GUI classes with the SQL classes.

To lower the memory requirements, we implemented lazy load ([19], [20]). This means we load data from the database only when we might need them. To be precise, we load the data just before the user might request to see them. This way, it also helps with the performance and the speed of the response.

5.1.1 GUI

The main application window (see Figure 5.1) is composed of two parts - a file explorer and a key shapes display.

The initialization of the application can be seen in the following UML diagram 5.3.

If the application fails to start due to a problem with the database an error dialogue is shown (see Figure 5.2).

The file explorer shows all folders and projects retrieved from the database. Each folder can be expanded to show its subfolders by clicking the triangle next to it or

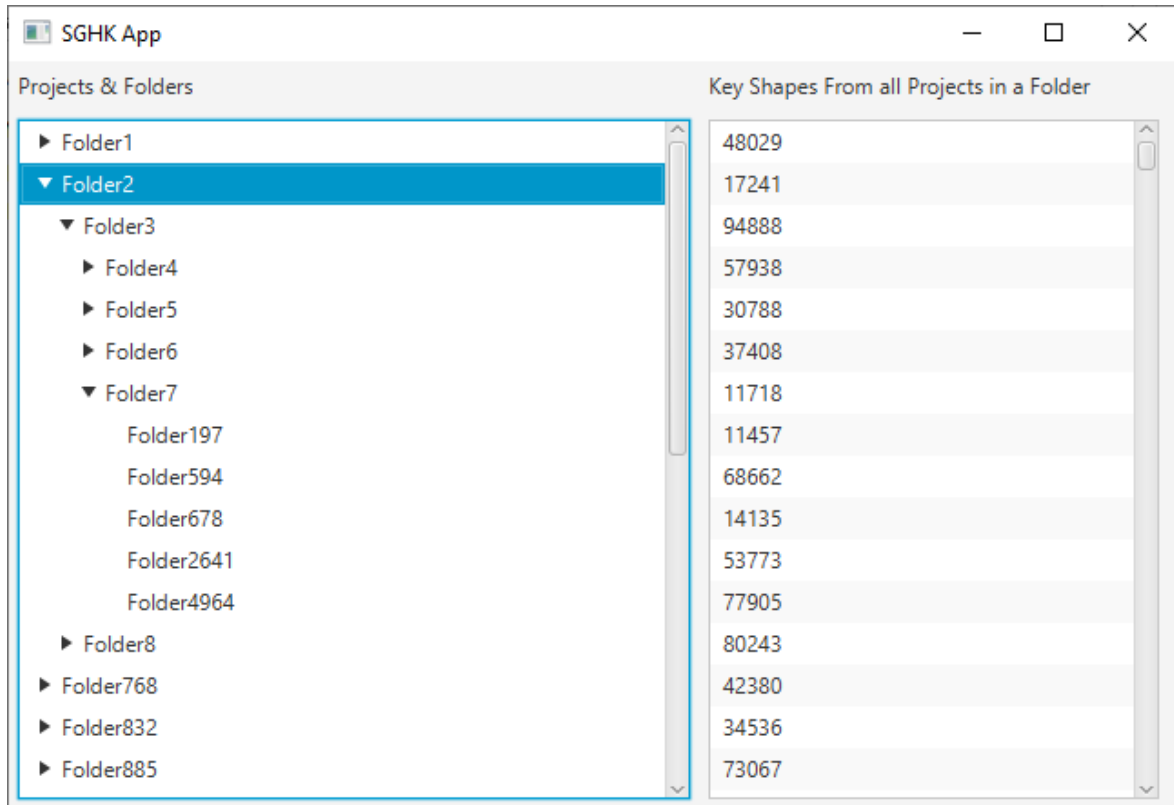


Figure 5.1. Application window

double-clicking the folder itself, then collapsed by clicking the triangle again. The selected folder can be renamed (see Figure 5.13) by clicking or using the menu. More on that in the Rename Menu Item section.

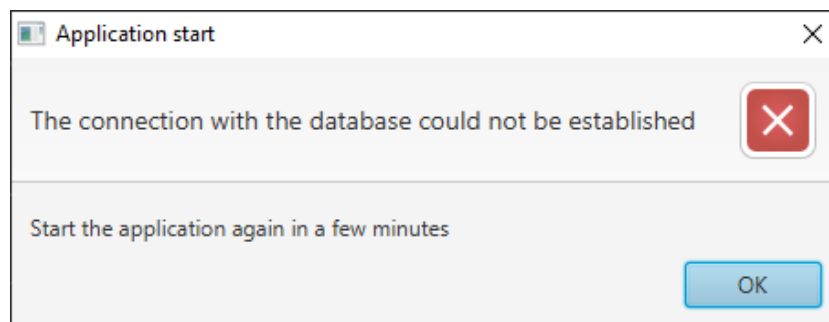


Figure 5.2. Application start failed dialogue

Every time the user adds a new key shape or selects a folder by clicking it, all key shapes belonging to that folder are shown in the key shape display. More specifically, the key shapes belong to the folders on the bottom of the hierarchy.

If the key shapes cannot be retrieved from the database, the user is promptly informed by a dialogue (see Figure 5.5).

A context menu can be brought up by clicking the right mouse button on any folder in the window, as seen in Figure 5.4.

The functions available in the menu are described in the following chapters.

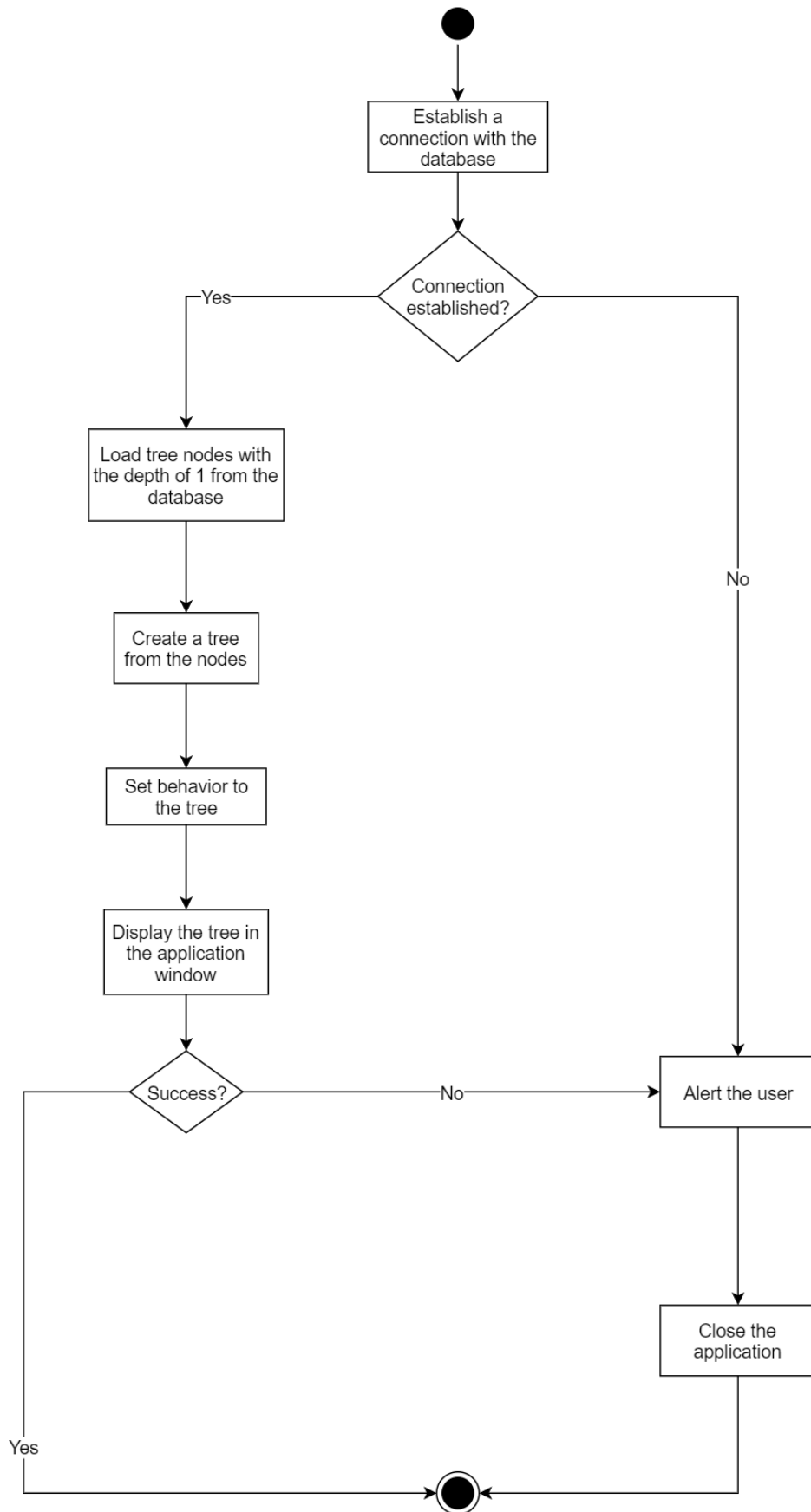


Figure 5.3. Application initialization UML Diagram

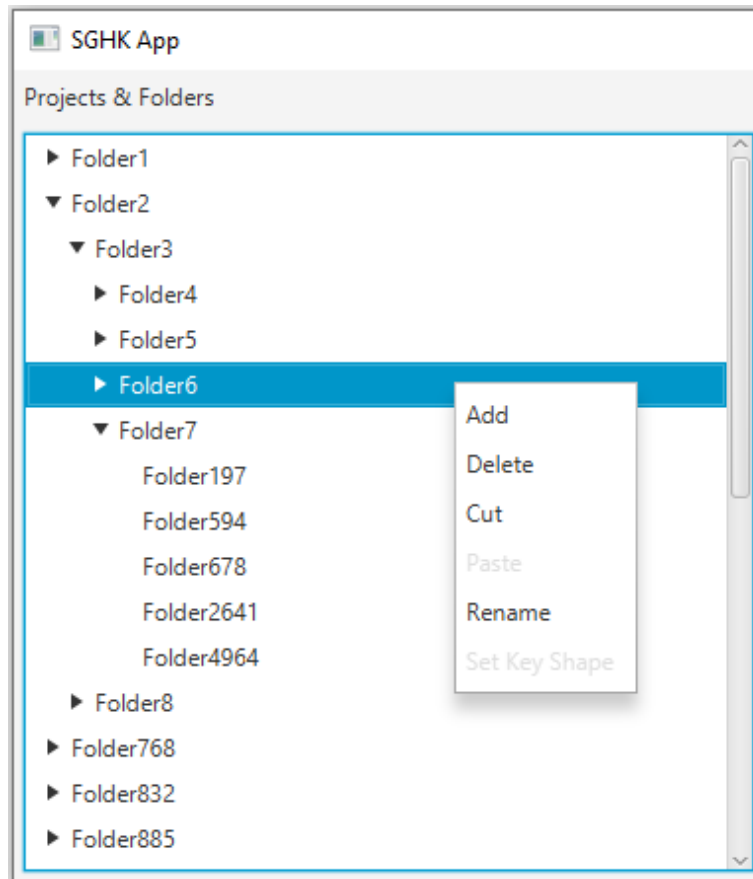


Figure 5.4. Menu

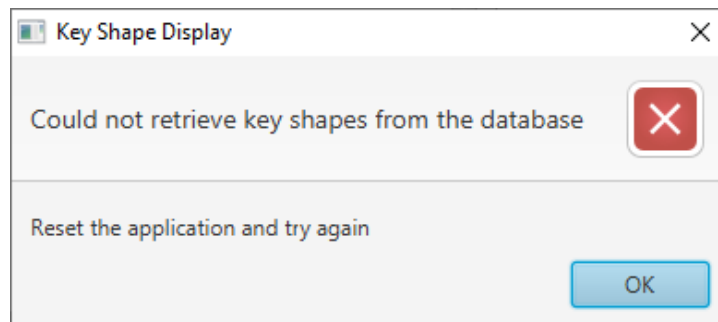


Figure 5.5. Failed key shape display dialogue

5.1.1.1 Add Menu Item

After the *Add* item is clicked, a folder is added into the current one. The new folder is going to have a unique name. In case there is a problem with the database and the folder cannot be added, an error dialogue (see Figure 5.6) shows up informing the user of the issue.

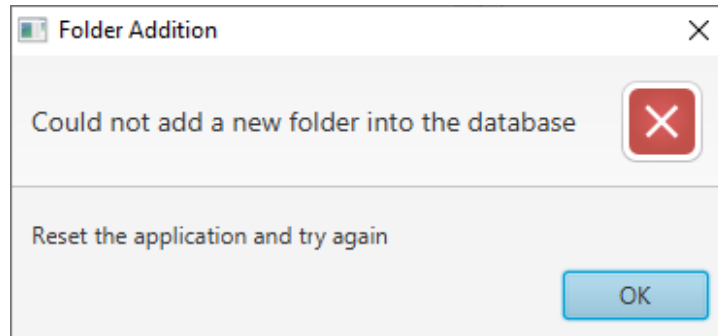


Figure 5.6. Failed addition dialogue

Following a successful addition, the new folder is going to be selected and centred. To understand this function better, see the Add UML diagram 5.7

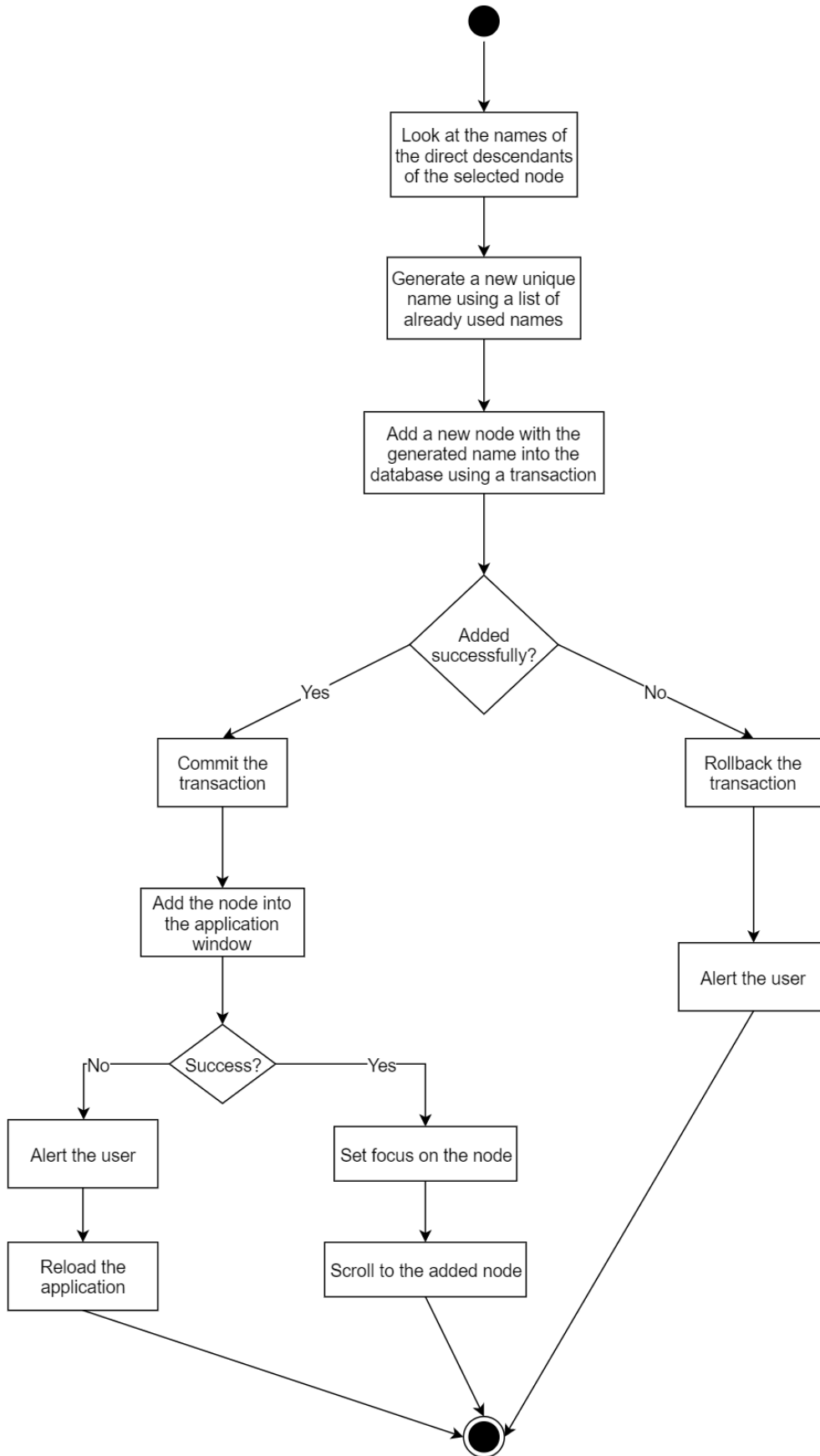


Figure 5.7. Addition UML Diagram

5.1.1.2 Delete Menu Item

The *Delete* item removes the selected folder from the explorer and the database. If the removal fails an error dialogue (see Figure 5.8) is shown.

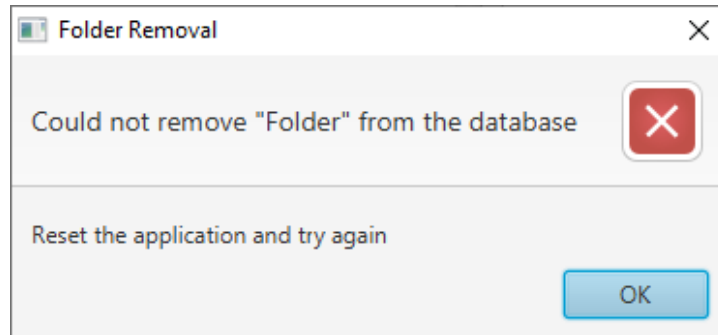


Figure 5.8. Failed deletion dialogue

The logic behind this function can be seen on the Deletion UML diagram 5.9.

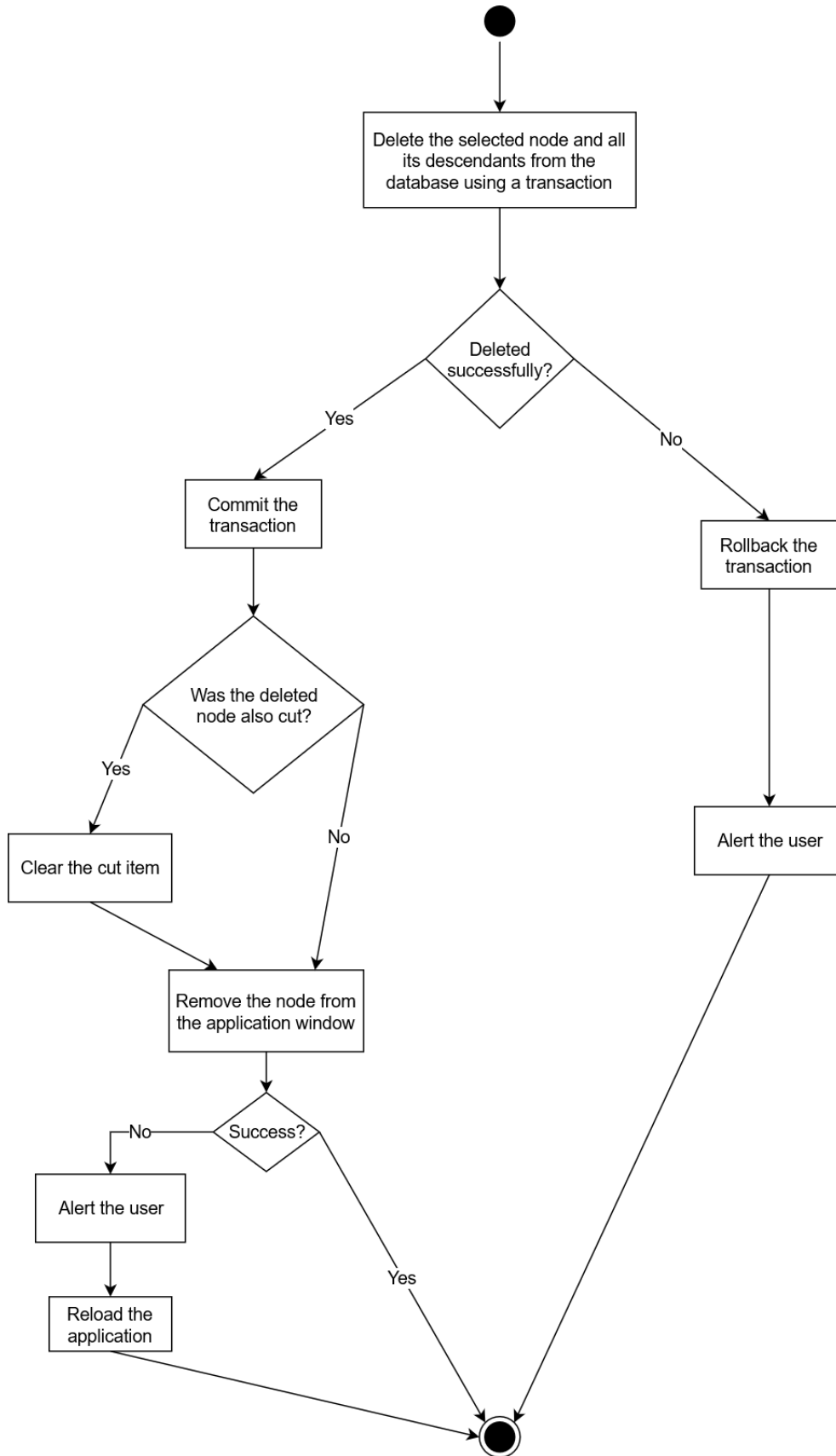


Figure 5.9. Deletion UML Diagram

5.1.1.3 Cut Menu Item

The user has the option to *cut&paste* a specific folder. After clicking the *Cut* item, the folder is copied to the clipboard. This means that if another folder is then copied, the previous one is replaced in the clipboard.

There is no diagram included for this function since it is very elementary.

5.1.1.4 Paste Menu Item

The default state of the *Paste* item is disabled. It is only enabled after a folder is cut and disabled again immediately after it has been pasted. It is necessary to remember that if the cut folder is removed, it can no longer be pasted. A folder cannot be pasted into itself or any subfolder of its own. If the folder could not be pasted, a dialogue (see Figure 5.10) informs the user.

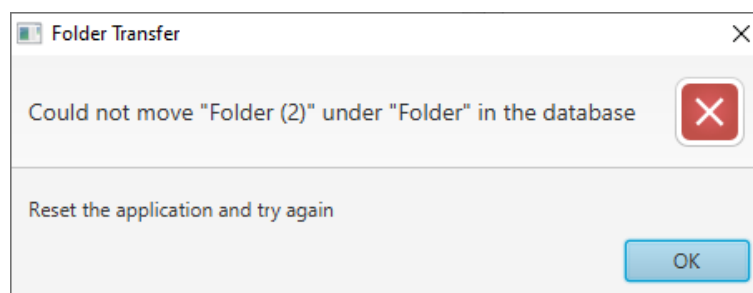


Figure 5.10. Failed transfer dialogue

When a folder of a particular name is pasted next to another folder of the same name, the user is given the option of merging those two folders or letting there be two folders of the same name (see Figure 5.11).

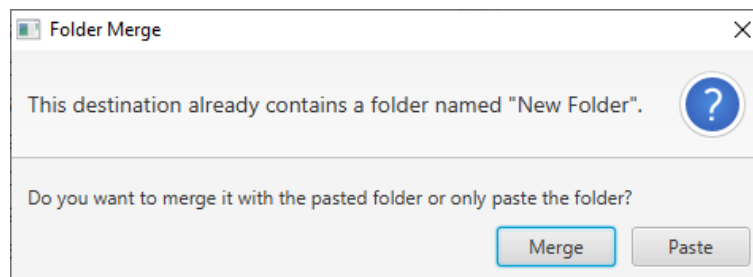


Figure 5.11. Merge or Paste Folders dialogue

Should another folder with the same name be pasted, the user is no longer given the option of merging the folders since there is no way of knowing which folders should merge.

This function is quite complicated and can be better explained via a UML diagram 5.12.

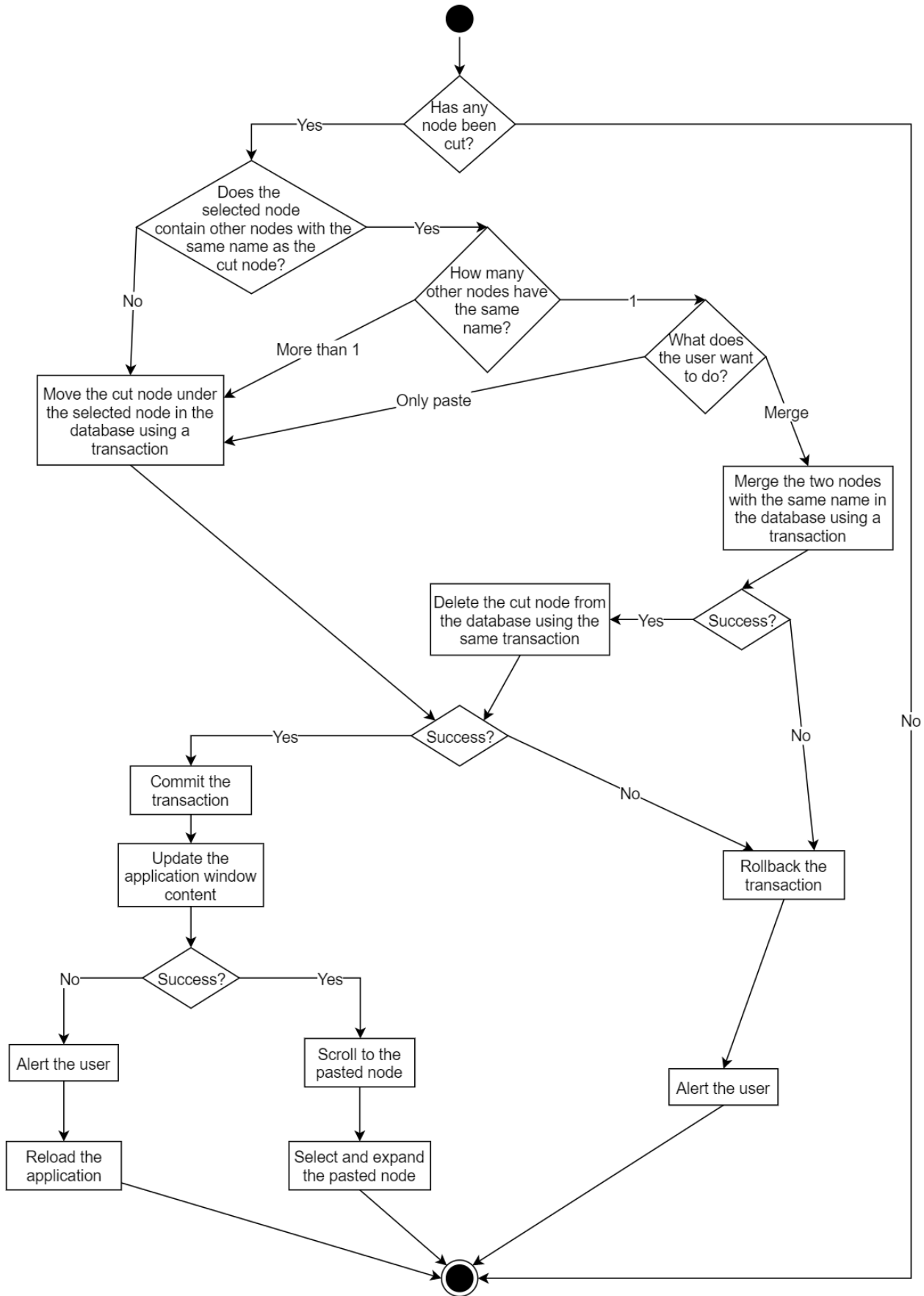


Figure 5.12. Paste UML Diagram

5.1.1.5 Rename Menu Item

One way to rename a folder is to click the *Rename* item in the menu. This creates a textfield (see Figure 5.13) for the user to type the new name in.

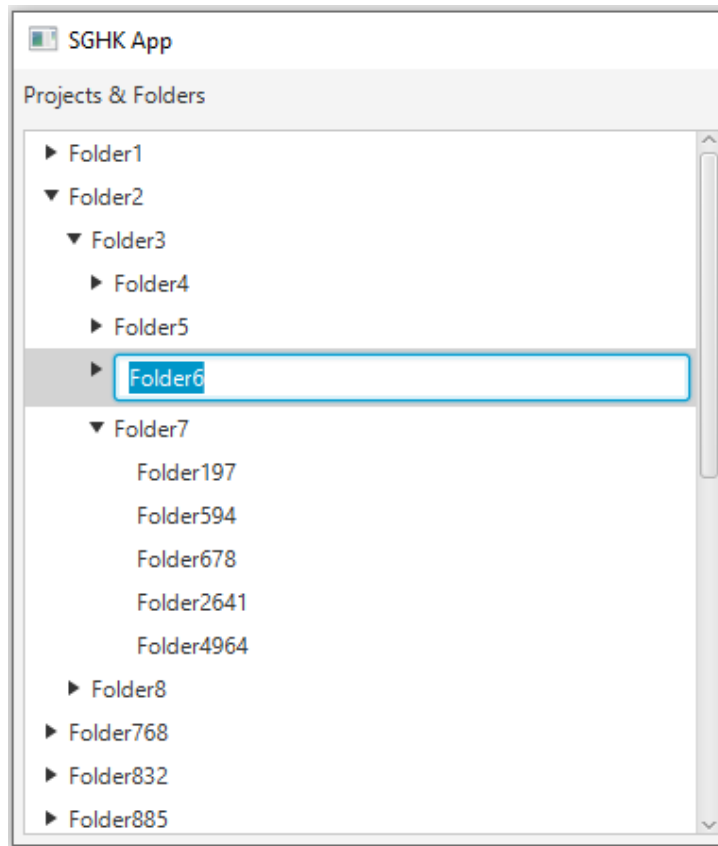


Figure 5.13. Rename textfield

The new name can contain only alphanumeric characters, white spaces and the following symbols: .,-()

As soon as the user writes any different character, a tooltip (see Figure 5.14) is shown to notify them.

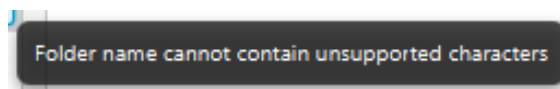


Figure 5.14. Wrong name tooltip

It disappears once the character is deleted. If the user, however, decides to confirm the name (by pressing the *Enter* key), even though the name is not valid, a warning (see Figure 5.15) pops up.

A different warning (see Figure 5.16) is shown if the new name is empty.

The new name can also not be longer than 255 characters. In case the user types a name that long, a tooltip (see Figure 5.17) informs them as such.

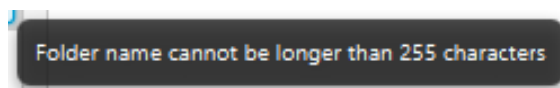


Figure 5.17. Long name tooltip

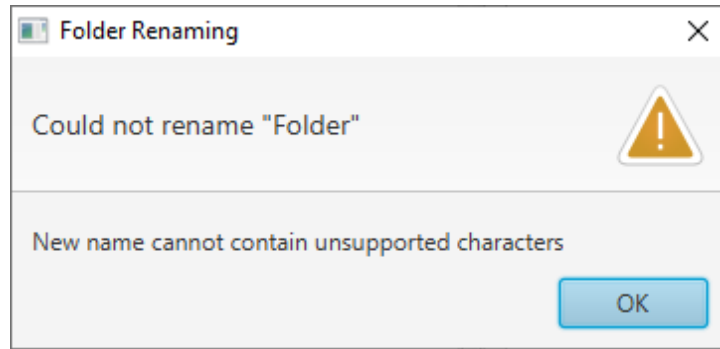


Figure 5.15. Wrong name warning

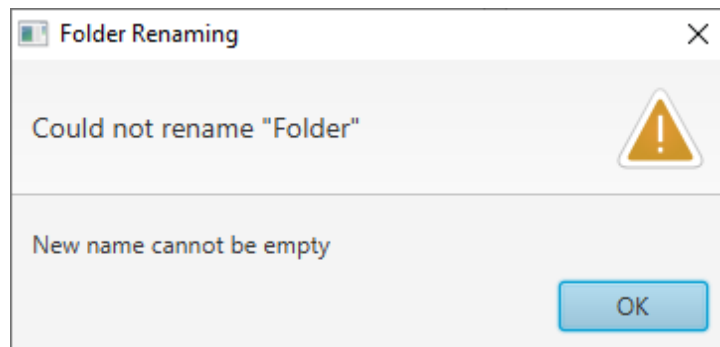


Figure 5.16. Empty name warning

When confirmed, a warning (see Figure 5.18) advises the user to change the name. The tooltip again disappears once the length has been shortened.

The length issue supersedes the unsupported character problem.

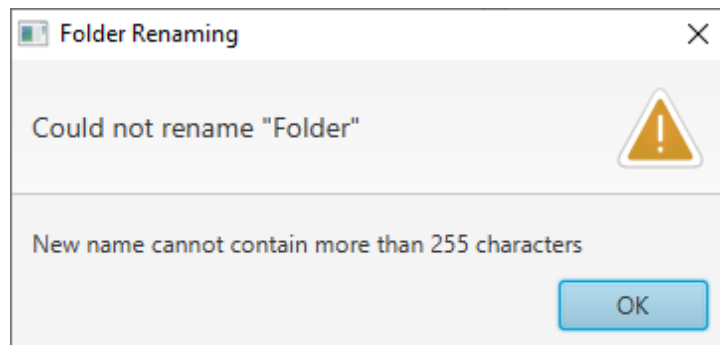


Figure 5.18. Long name warning

The user is encouraged to use unique folder names. If they decide to change the name of a folder to an already existing name, they will have the option to merge those two folders (see Figure 5.19).

If they choose to do so, all the subfolders of the renamed folder will be transferred under the other folder. This may however result in a database error (see Figure 5.20).

The user may also choose to rename the folder. In which case, there will be two (possibly more) folders with the same name. It is allowed but not recommended.

In case there are more folders with the same name as the newly renamed folder, the user is not given the option of merging them.

Renaming a folder might result into a database error (see Figure 5.21).

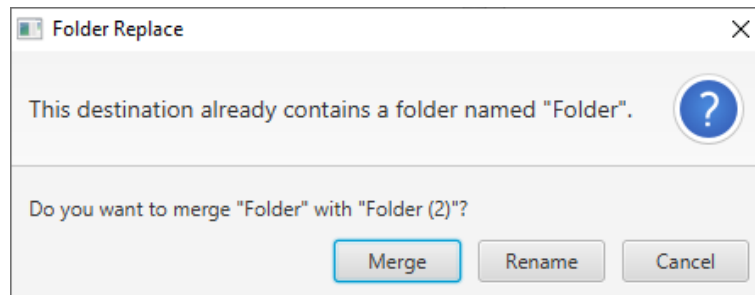


Figure 5.19. Merge folders confirmation

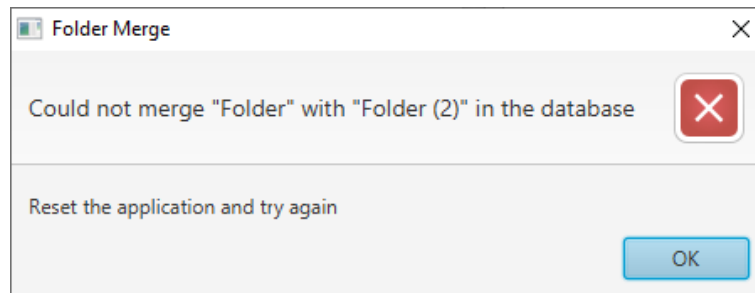


Figure 5.20. Failed merge dialogue

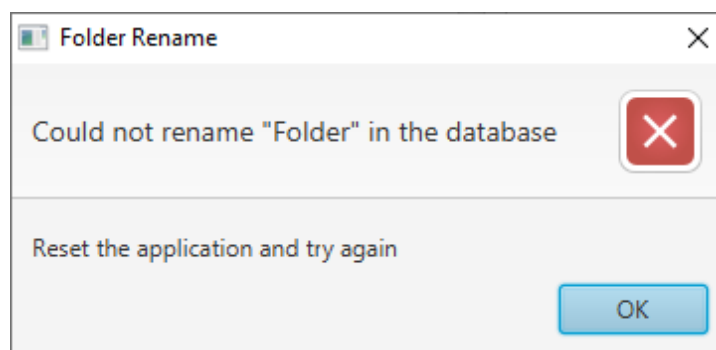


Figure 5.21. Failed rename dialogue

The *Cancel* option in the merge dialogue (Figure 5.19) cancels the renaming process, in the same way, pressing the *Esc* key would.

The rename function can be visualized using the Rename UML diagram 5.22.

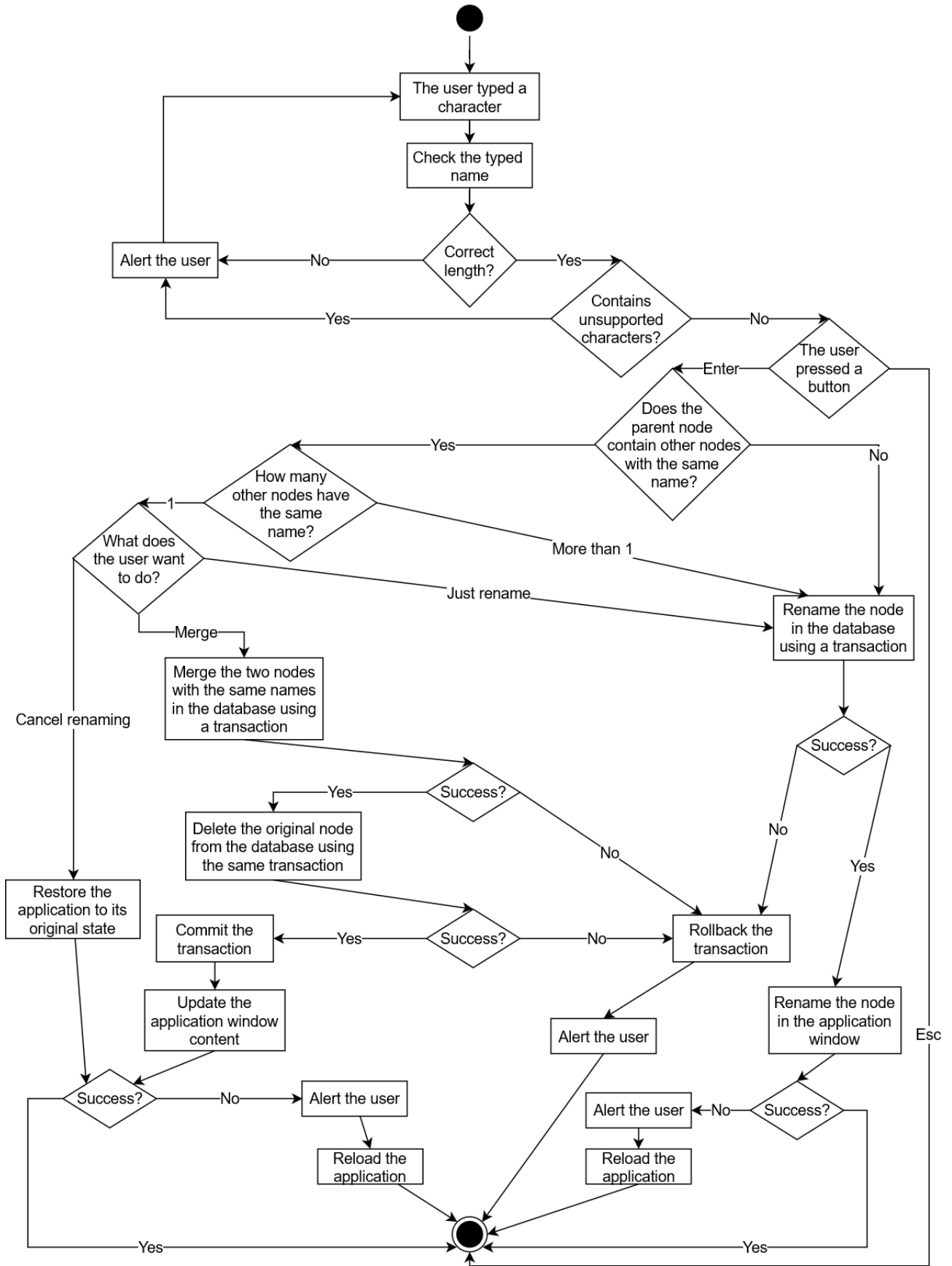


Figure 5.22. Rename UML Diagram

5.1.1.6 Set Key Shape Menu Item

The key shape is set using a dialogue (see Figure 5.23) and can be set only in a leaf folder. If the project already had a key shape, it will be shown in the dialogue window.

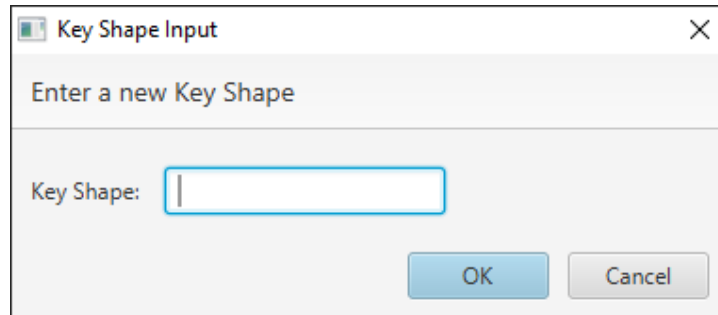


Figure 5.23. Key shape set dialogue

The *OK* button is disabled by default, enabled only after a valid key shape is typed.

A key shape can only contain numeric characters and can not be longer than 255 digits. After breaking these restrictions, an appropriate tooltip is shown (see Figure 5.24 and 5.25).

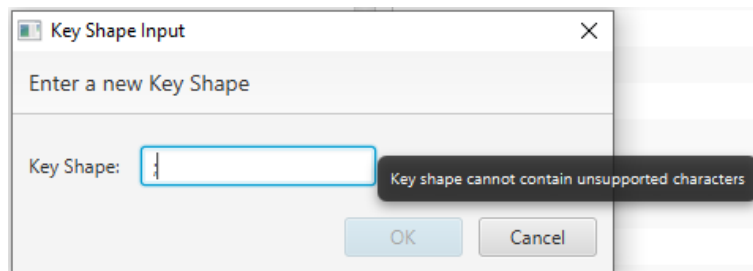


Figure 5.24. Wrong key shape tooltip

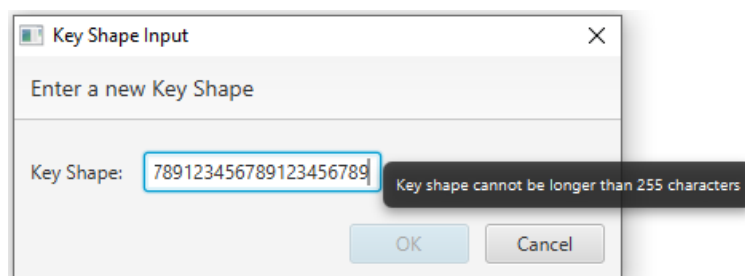


Figure 5.25. Long key shape tooltip

If the new key shape is empty (user deleted the shape from the text field), it will be deleted from the database. The user is required to confirm the deletion (see Figure 5.27).

When designing this function, the Key Shape Set UML Diagram was created 5.26.

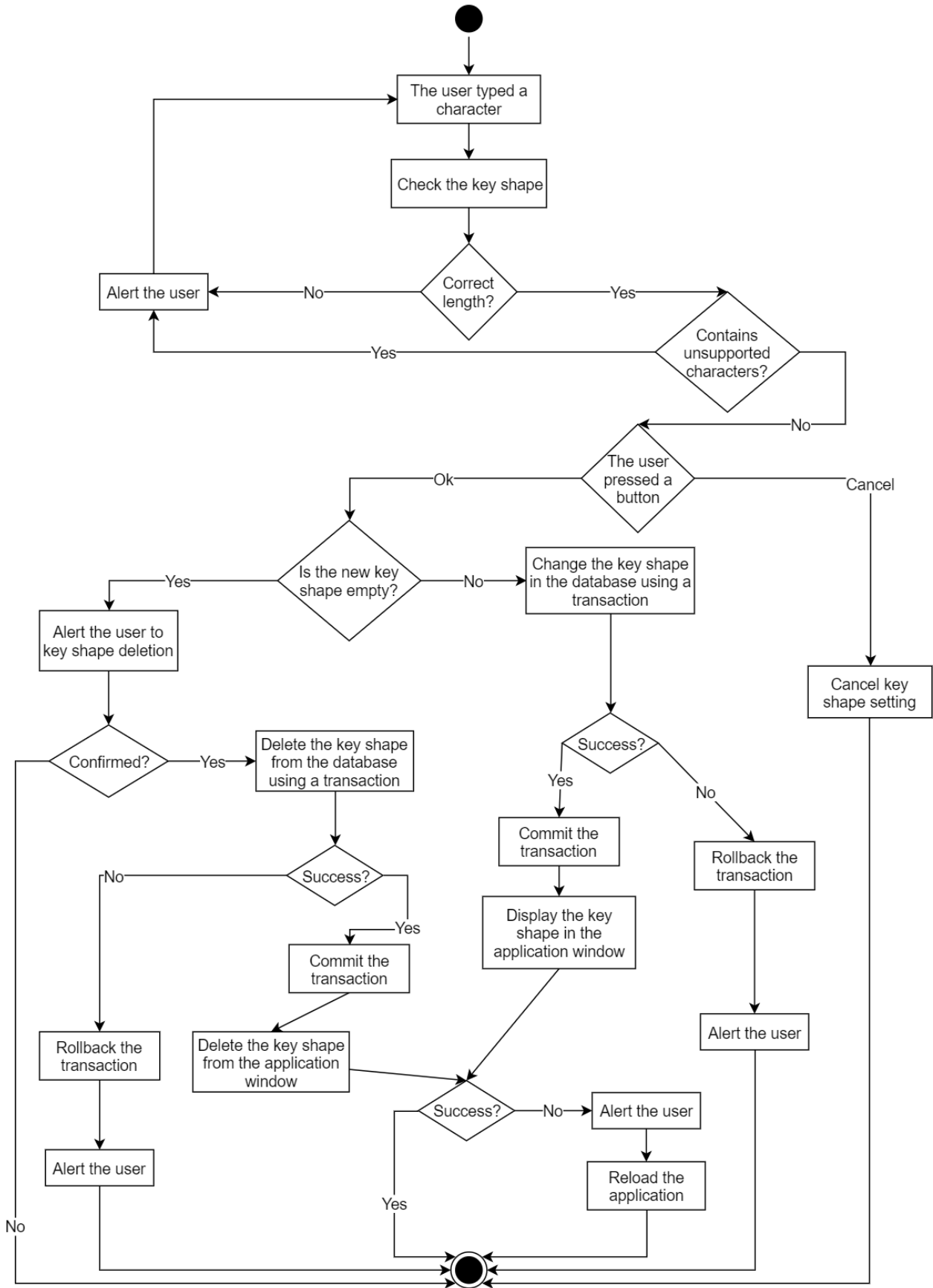


Figure 5.26. Key Shape Set UML Diagram

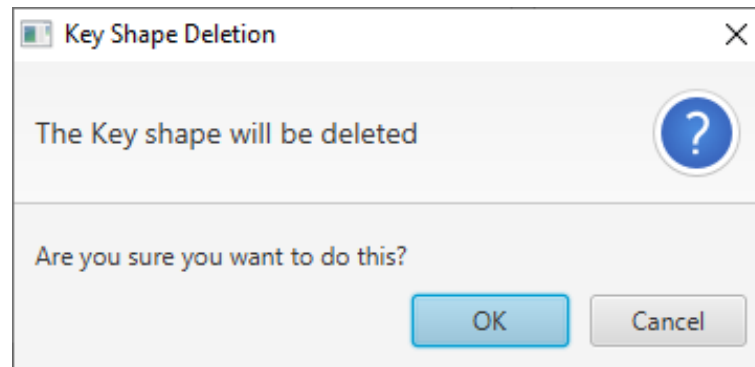


Figure 5.27. Key shape delete dialogue

■ 5.1.2 GUI Package

In this package, all classes handling the user interface are located (see documentation [21]). Their purpose is to allow the user the interaction with the database.

■ 5.1.2.1 ProjectUI

This is the main class of the project, where the application window is created. The only thing the main method does is call the launch method, which is the main method of the application.

In the application window, a *TreeView* component was used to represent the file explorer. We then used a *ListView* component to display the key shapes. As mentioned in chapter 5.1.1, the key shapes are displayed every time an item is selected. The selection evokes an event caught by a listener. The key shapes are retrieved by an SQL query run in a thread. If it fails, the user is informed of it by an *Alert* (see Figure 5.5).

These two components are placed in a *GridPane* into two columns. Before the main window is shown, a connection with the database is established. It is disconnected once the application is closed. In case the connection can not be established, the application is closed, and an alert is shown (see Figure 5.2).

■ 5.1.2.2 TreeItemNode

The *TreeView* mentioned in the ProjectUI chapter is composed of *TreeItems*. The *TreeItem* class is imported from the JavaFX scene control package. Usually, it would not be a problem to use the *TreeItem* class as the representation of a single node in the tree view. However, this way, all the nodes would be stored in memory. Since there is going to be a massive amount of data, we needed to load into the memory only those, that are visible to the user. We did this by extending the *TreeItem* class, thus creating our own *TreeItem (TreeItemNode)*. Then we had to override the *isLeaf* and the *getChildren* methods. These methods are called automatically by the tree view in order to display its items correctly. If the *isLeaf* and the *childrenList* properties of a tree item are already set, we simply return them from the superclass. Otherwise, we load the necessary data from the database and set the properties in the superclass accordingly.

■ 5.1.2.3 TreeCell

A *TreeCell* takes care of the selection model of the *TreeView*. It makes sure, to visually indicate to the user if they have selected it. It also works as a set of instructions each node follows. The *TreeCell* is assigned to the *TreeView* in the ProjectUI class by using the *setCellFactory* method.

To allow the user renaming of the nodes, we had to override a few methods. So naturally, our *TreeCell* class extends the JavaFX scene control *TreeCell*<> class. The main thing the overridden methods do is create a *TextField* for the user to type the new name in. After that, we only added a few name-checks (more on that in chapter 5.1.1) by checking the length of the typed name and also checking if it matches the regex `^[\\w., ()-]+$`. If the name passed the checks, we would rename the node in the database and displayed a matching alert (see Figure 5.21) in case it failed. Because we call the SQL classes in threads, we must explicitly use the JavaFX thread *Platform* [8] to display the alert.

If the name were already present in the folder (which we find out by checking all sibling folders), we would alert the user. We give them the options to merge the current folder with the other one with the corresponding name or rename the folder, thus having duplicate folder names.

The merge can also be unsuccessful due to a database issue. It is again treated by cancelling it and informing the user.

We also added a *ContextMenu* to the tree cell. More about the functionalities of the menu in chapter 5.1.1. The *ContextMenu* is composed of *MenuItems*. We added an *EventHandler* to every menu item using lambda functions. There we set the properties of the nodes in the tree based on the user's demands and did the same in the database. Again with proper alerts shown if something went wrong.

5.2 SQL Package

For the purpose of implementation, we chose the Apache Derby database [22]. It runs on any machine with Java and can be embedded in Java programs, which is why it was used in this thesis. It is possible to change the database system when releasing the application. We are not using any unique properties of the Derby database.

Every query needed for the interaction with the database has its own class in this package. The classes implement the *Runnable* interface so they can be passed to threads as arguments.

Let us now go through the way transactions are managed in the implementation of this thesis.

5.2.1 Transaction Management

To execute a query, we first had to create a statement using the connection with the database. We used *PreparedStatement* [23]. The main advantages of using *PreparedStatements* as opposed to classic *Statements* are that they are much faster to execute since they are pre-compiled. We can also feed them parameters very easily using question mark placeholders.

Sometimes we had to use a few statements that were dependent on one another. If one of them failed to execute and the others did not, there would be an inconsistency within the database. To prevent that, we used transactions [5, pp. 625-631]. A transaction is created with every statement, so there is no need to begin one manually. By default, the connection commits every transaction once the statement has been executed. However, we wanted a few statements to be executed together, meaning if one fails, do not execute the others, so we forbade the auto-commit. Because of it, we had to commit every transaction manually.

Any database operation can fail and throw an exception. That is why the statements are surrounded with try/catch blocks. If the exception is thrown, we cancel the transac-

tion by using rollback [5, p. 648] and throw another exception which is then handled in the GUI classes (see 5.1.2), so the user can be informed of the failed database operation.

Some of the statements would return a *ResultSet* of data in case the executed query was a *SELECT* or an *INSERT*. We would then retrieve the data by using the support class *ResultSetDrainer*, which would extract the nodes from the *ResultSet*.

Since we fetch the data in the void Run methods, we are forced to use Consumers to ‘return’ the data.

The transaction management fulfils client’s hidden functional requirements ([12], [24]).

Now we are going to take a closer look at a few of the runnable classes in the SQL package. Some of the classes are essentially the same except they execute different queries, which are however identical to the ones described in the Database Model chapter 4. So the ones we are going to be interested in are those that have been slightly changed compared to the original draft or were not there at all.

■ 5.2.2 TableInitializer

Each database record requires a unique identifier (*id*). Nowadays, mostly artificial ones are being used. Since it is a common practice, we went with that in our project as well. However, maintaining an id would be too difficult. Luckily, there is another way. When creating a table, we can say we want to generate the ids automatically.

In the *CREATE TABLE Tree* statement we added a clause which states, the id values are going to be generated starting from $id = 1$. Thanks to this clause, every time a record is added into the database, it is automatically assigned a unique id number, which is always higher by one than the previous one.

■ 5.2.3 Adder

When inserting a record into the database, we added the *RETURN GENERATED KEYS* option to the *PreparedStatement*. The statement then after its execution returns a *ResultSet* with the generated id.

■ 5.2.4 Renamer

First of the minor classes to mention is the *Renamer* class. It merely sets the name of a node with the given id in the database.

■ 5.2.5 Key Renamer

This class is the same class as the *Renamer* class, except this class, sets key shapes.

■ 5.2.6 Selector

The *Selector* is only a support class with static methods. It is used to retrieve a node from the database given the id or all nodes in the table.

■ 5.2.7 Table cleaner

This class allows us to *DROP* the Tree table. In a typical application run, we would never need to do such a thing. The table is dropped only in testing. More about that in the Unit Tests chapter 5.3.

5.3 Unit Tests

All the queries we used had to be adequately tested to assure the reliability of the final application. Therefore we devised a series of unit tests [25]. Every single database query used in our project has its own unit test. For the purpose of testing, we used an in-memory database and some dummy data.

Before running all the tests in a test class, a connection with a database was established. Then before each test in the class, a table was created and a root node inserted in it. After the test, we would drop the table to give us the option of writing more test cases. The reason we had to drop the table and not just delete all the data in it is, the automatically generated id would still be increasing, even with the data deleted, thus making the individual tests dependent.

Chapter 6

Usability Testing

After the implementation of the application had been completed, we had to find out if the application was fully usable and fulfilled the client's expectations. In this case, the client is also the user.

For the purposes of testing, we chose usability testing. Usability testing is a kind of testing where the user gets a task list, and we watch them go through the list. We also make notes of the tasks which prove problematic to the user. We also watch for any hesitations or actions unsuccessfully attempted by the user. It is also important for us that the user is satisfied and uses the application efficiently.

Before the testing itself, the functionality of the application was tested with an independent user. The goal was to find out if the application is fully functional, there are no bugs, and there is nothing that would endanger future usability testing; for example, the application does not crash or freeze. This user had skills on a similar level to those of the target group. Those skills are specified in the next paragraph.

We conducted the usability testing of the application with only one user. This particular user is the head of the department which is planning on using the application. The department consists of about three people, therefore the testing of the application with only one user is sufficient. This user reliably represents the target group. The users from the target group have these skills and characteristics:

- they often use PC, keyboard and computer mouse
- they have at least basic knowledge of English language
- they are familiar with the use of Windows File Explorer, MS Excel and MS Word
- they do not suffer from any form of visual impairment

Under other circumstances, we would have used a screener [26, pp. 124-127] to select our test subjects. Since this application is intended for specific users, it is better to test it with them.

In the following sections, we are going to go through the process of user testing with this particular participant. We are also going to list all the findings which arose from the testing and the ways we resolved them.

6.1 User Testing

At the beginning of the testing, we explained the client (hereinafter referred to as the participant) the reason they came and what is expected from them. We emphasized that it is the product (application) that is supposed to be tested, not the participant. Every complication we stumble upon is a fault of the application, not the fault of the participant. It is also essential that the test participant thinks out loud [26, pp. 199-207]. This helps us understand which tasks from the task list are difficult for the participant and potential user to execute and most importantly, why.

The task list given to the participant did not test every function of the application, but it tested the most used and crucial ones. If the participant manages to complete

After the data collection was completed, we moved to the post-test phase. We asked the participant about the tasks they had troubles with. There was not much to discuss in this post-test part of the usability testing since the participant explained his confusion over certain tasks during the previous part already. We merely confirmed all the things the participant said during the testing. The participant also added and cut-pasted a few folders and project in order to verify if they understand the logic behind adding and pasting well. There however seemed to be room for improvement in this area.

The client was satisfied with the way the application looks and behaves. They had only a few remarks concerning the issues we already mentioned.

In case there were some troubles during the testing, the participant might feel incompetent. It is necessary to make sure that the participant leaves in a good mood or preferably better mood than in which they came.

6.2 Findings and revision

We divided the findings into three categories based on their priority. Priority one is the lowest. Priority one findings bear on cosmetic or aesthetic problems and do not have any impact on the usability of the application. Priority two relates to moderately severe problems. These issues do not impede the usability of the application much, but they might relate to some features which would be nice to have. Priority three is the highest. Problems of this priority significantly compromise or restrict the functioning of the application, thus making it in some cases almost impossible to use [26, pp. 299-300].

In this section, we are going to go through the individual findings and the way we resolved each of them.

■ Finding #1 (priority 2 - moderate)

The participant seemed confused at first about the level where a folder is added. More specifically, whether it is added next to the selected folder or under it. The fact that they can add a new project under an already existing project further intensified the confusion.

Solution

Now it is impossible to add a folder/project under another project. The main characteristic of a project is that it has a set key shape.

■ Finding #2 (priority 2 - moderate)

The client wanted some safety check before the deletion of any project or folder.

Solution

We added a confirmation dialogue (see 6.1) which is displayed after the user attempts to remove a project. Similar dialogue is displayed in case of a folder deletion attempt. The project/folder is removed solely after the user confirms the deletion in the dialogue. This hugely improves the prevention of errors caused by the user.

■ Finding #3 (priority 2 - moderate)

The length of a key shape used to be restricted to 255 digits. In reality, the key shape can not be longer than 7 digits. The participant pointed out this issue.

Solution

The key shape setting restrictions are now tightened up. A key shape can no longer contain more than 7 digits. A key shape set tooltip has been adjusted accordingly.

■ Finding #4 (priority 1 - low)

The test subject tried to maximise the application window or at least enlarge it. They were unhappy with the application having a maximal size set.

Solution

The application was modified and now can be enlarged, maximised, and the projects navigation pane and the key shape display pane scale accordingly. And since the maximisation was the first thing the participant tried to do, the application is now maximised by default.

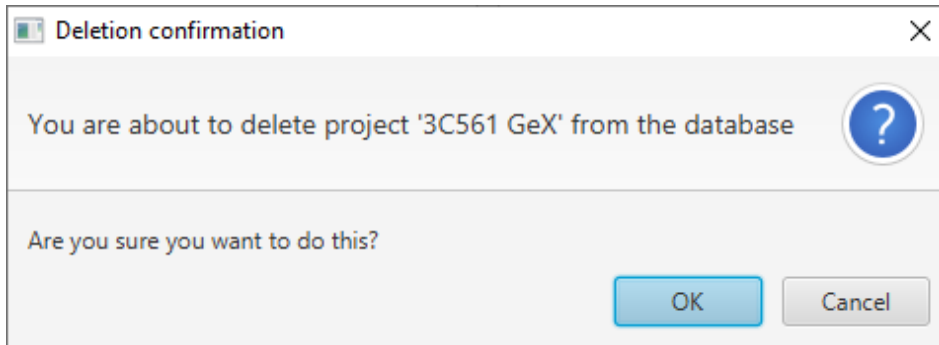



Figure 6.1. Project deletion conformation dialogue

All the findings from the usability testing are now resolved.



Chapter 7

Conclusion

We have created an application for the master-key system maintenance. The application comes with a user interface (see Figure 5.1) allowing the user the retrieval of the key shape data from a database as well as its storage within a project by the use of a context menu. We discussed every operation the user is likely to make.

We made sure the interface runs as swiftly as possible by using threads for all the database interactions, implemented lazy loading to lower the memory requirements, and we also covered all errors and exceptions that may occur.

Before the implementation itself, we had to propose a database model well suitable for hierarchy management. We considered a few of them, prepared the fundamental queries needed for the application run and compared them based on their complexity and comprehensibility and also experimentally measured the execution time of the queries within each model. We then implemented the most efficient one.

We thoroughly tested every query we used in the implementation using a series of unit tests.

After the application was fully functional, we tested it via usability testing. We then processed the results of this testing and resolved any issues, which emerged from it.

For future work, we would suggest expanding the user interface and adding a few minor features to it. Then prepare the application for the integration into a final product and eventually its release.



References

- [1] HILLYER, Mike. Managing Hierarchical Data in MySQL. Mike Hillyer's Personal Webspaces [online] (n.d.). Retrieved from: <http://mikehillyer.com/articles/managing-hierarchical-data-in-mysql/>.
- [2] PULFORD, Graham. *High-Security Mechanical Locks: An Encyclopedic Reference*. Butterworth-Heinemann, 2007. ISBN 978-0750684378.
- [3] Wikimedia Foundation [online], 2008. Retrieved from: https://commons.wikimedia.org/wiki/File:Pin_tumbler_no_key.svg.
- [4] Systém generálního klíče. FAB [online]. ASSA ABLOY (n.d.). Retrieved from: http://www.fab.cz/inspirace/prispevek/26740/system_generalniho_klice.
- [5] SILBERSCHATZ, Abraham, Henry F. KORTH, and S. SUDARSHAN. *Database System Concepts (6th ed.)*. New York: McGraw-Hill, 2011. ISBN 978-0-07-352332-3.
- [6] TIOBE Index for May 2019. TIOBE [online]. TIOBE Software, 2019. Retrieved from: <https://www.tiobe.com/tiobe-index/>.
- [7] LANGLEY, Nick. Write once, run anywhere?. ComputerWeekly [online]. TechTarget, 2002. Retrieved from: <https://www.computerweekly.com/feature/Write-once-run-anywhere>.
- [8] TAMAN, Mohamed. *JavaFX Essentials*. Birmingham: Packt Publishing, 2015. ISBN 978-1-78439-802-6.
- [9] FEDORTSOVA, Irina. Why Use FXML. Oracle Docs [online]. Oracle, 2014. Retrieved from: https://docs.oracle.com/javafx/2/fxml_get_started/why_use_fxml.htm.
- [10] JavaFX 2.2.5 System Requirements. Oracle Docs [online]. Oracle, 2013. Retrieved from: <https://docs.oracle.com/javafx/2/index.html>.
- [11] HOMMEL, Scott. Using JavaFX Properties and Binding. Oracle Docs [online]. Oracle, 2013. Retrieved from: <https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm>.
- [12] Functional Requirements. Science Direct [online]. Elsevier, 2019. Retrieved from: www.sciencedirect.com/topics/computer-science/functional-requirement.
- [13] QUASSNOI. Adjacency list vs. nested sets: PostgreSQL. Explain Extended [online]. Explain Extended, 2009. Retrieved from:

- <https://explainextended.com/2009/09/24/adjacency-list-vs-nested-sets-postgresql/>.
- [14] TROPASHKO, Vadim. Nested Sets and Materialized Path SQL Trees. Rampant Techpress [online]. Oracle, 1996-2017.
Retrieved from:
http://www.rampant-books.com/art_vadim_nested_sets_sql_trees.htm.
- [15] PTÁČEK, Pavel. Ukládáme hierarchická data v databázi – III. Zdroják [online]. Devel.cz Lab, 2012.
Retrieved from:
<https://www.zdrojak.cz/clanky/ukladame-hierarchicka-data-v-databazi-iii/>.
- [16] ARLOW, Jim, and Ila NEUSTADT. *UML 2 a unifikovaný proces vývoje aplikací*. Brno: Computer Press, 2011. ISBN 978-80-251-1503-9.
- [17] draw.io [online].
Retrieved from:
<https://www.draw.io/>.
- [18] Interface Runnable. Oracle Docs [online]. Oracle, 2018.
Retrieved from:
<https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html>.
- [19] WAGNER, Jeremy. Lazy Loading Images and Video. Google Developers [online]. Google Inc.
Retrieved from:
<https://developers.google.com/web/fundamentals/performance/lazy-loading-guidance/images-and-video>.
- [20] What is Lazy Loading?. GeeksforGeeks [online]. GeeksforGeeks (n.d.).
Retrieved from:
<https://www.geeksforgeeks.org/what-is-lazy-loading/>.
- [21] JavaFX 2.2. Oracle Docs [online]. Oracle and/or its affiliates, 2008, 2014.
Retrieved from:
<https://docs.oracle.com/javafx/2/api/index.html>.
- [22] Apache Derby [online]. The Apache Software Foundation, 2004. Retrieved from:
<https://db.apache.org/derby/>.
- [23] Using Prepared Statements. Oracle Docs [online]. Oracle, 2017.
Retrieved from:
<https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html>.
- [24] WAZLAWICK, Raul Sidnei. *Object-Oriented Analysis and Design for Information Systems*. Morgan Kaufmann, 2014. ISBN 978-0-12-418673-6.
- [25] Unit Testing. Software Testing Fundamentals [online]. STF, 2019.
Retrieved from:
<http://softwaretestingfundamentals.com/unit-testing/>.
- [26] BARNUM, Carol M. *Usability Testing Essentials : Ready, Set... Test!* Morgan Kaufmann, 2011. ISBN 978-0-12-375092-1.