FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Development of a Tool for Creating Evolvable Study Materials
Student:	Bc. Svetlana Ekimova
Supervisor:	Ing. Vojtěch Knaisl
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 aim of the thesis is to develop a tool to improve the evolvability of study materials at the Faculty of Information Technology with the application of Normalized Systems theory principles.
Steps:
- Acquaint yourself with the Normalized Systems theory
- Acquaint yourself with the current approaches for creating study materials at our Faculty
- Analyze the requirements for the tool
- Design a solution
- Implement a software prototype solution
- Test and document your solution

–

[1] https://www.uantwerpen.be/en/
[2] https://ds-wizard.org/

Electronically approved by Ing. Michal Valenta, Ph.D. on 22 September 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Development of a Tool for Creating Evolvable Study Materials

Bc. Svetlana Ekimova

Department of Software Engineering
Supervisor: Ing. Vojtěch Knaisl

May 5, 2021

Acknowledgements

I would like to thank my supervisor, Ing. Vojtěch Knaisl, for his help, time and advice he gave me while I was writing this thesis. I would also like to thank Ing. Tomáš Kalvoda, Ph.D. and doc. Ing. Štěpán Starosta, Ph.D., for providing the information I needed to design the solution and for the willingness to participate in the testing of the thesis product. A special thanks is for Ing. Tomáš Kalvoda, Ph.D. for cooperating in the design of the tool and creating the input files for it.

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 5, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Ekimova, Svetlana. *Development of a Tool for Creating Evolvable Study Materials*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce se zabývá problematikou údržby studijních materiálů, které vznikají na Katedře aplikované matematiky na FIT ČVUT. Jejím cílem je najít řešení, jak zvýšit evolvabilitu dokumentů, které jsou poskytovány studentům jako studijní skripta, a jiných podpůrných materiálů pro výuku matematických předmětů na fakultě. Řešení je navrženo s použitím principů Teorie normalizovaných systémů. Koncový produkt je prototypem nástroje, který má pomoct vyřešit hlavní problém současného přístupu ke správě studijních materiálů, konkrétně inkonzistencím v obsahu mezi jednotlivými typy materiálů.

Klíčová slova evolvabilita, evolvabilní systémy, evolvabilní dokumenty, modularita, teorie normalizovaných systémů, studijní materiály

Abstract

This thesis addresses the problematics of study materials management. It focuses on the study materials created at the Department of Applied Mathematics at FIT CTU and its aim is to come up with a solution, how to improve the evolvability of documents distributed to students as lecture scripts and other supporting materials for teaching mathematics subjects at the faculty.

The solution is constructed based on the Normalized Systems theory principles. The end product is a prototype of a tool that should help to solve the main issue of the current approach to the study materials management, namely the inconsistencies in the content between the different study material types.

Keywords evolvability, evolvable systems, evolvable documents, modularity, normalized systems theory, study materials

Contents

List of Figures

List of Tables

Introduction

Evolvability is a term that is being used across different disciplines and describes, in rough words, the ability of an entity (a population, a system, etc.) to cope with the changing environment and to adapt to it. In software engineering, a system being evolvable means that it is flexible in terms of introducing updates and improvements – these changes are neither expensive nor time-consuming, even if the system is big and complex. The question, how to create evolvable software systems, is the subject of the Normalized systems theory. This theory proposes principles of evolvability and modularity, that can be applied in the process of software development, but also in other domains. An example of such a domain can be the creation and maintenance of text documents.

The idea that led to the origin of this thesis emerged from the Department of Applied Mathematics of the Faculty of Information Technology at the Czech Technical University: the study materials, which are created for mathematics subjects there, are being distributed to students in different formats that complement each other, and it is sometimes difficult to keep the contents of these formats in a consistent form.

The aim of this thesis is to develop a tool, which would help to improve maintaining the study materials at the faculty, make them more evolvable. This should be done by applying the principles of the Normalized systems theory, which are described in the first chapter of the thesis.

The second chapter presents the approaches of how the study materials for mathematics subjects at FIT CTU are created. These approaches are analyzed in chapter three, which also defines the requirements for the tool to be developed, and proposes the tool's design by creating use cases, a domain model, a state machine and activity diagrams.

The design of the user interface is developed in the forth chapter, followed by the implementation (chapter five) and testing (chapter six).

The tool shall serve as a prototype and implement a fundamental idea for the improvement of the study materials' evolvability, so that it would be

possible to expand its functionality in the future. Based on the results of the tool testing, further steps to be taken for the tool expansion will be discussed in chapter eight.

Chapter 1

Normalized Systems Theory

Normalized Systems Theory (NST) is a discipline that focuses on the question, how to design and develop complex systems to make them evolvable and flexible. NST was created by H. Mannaert, J. Verelst and P. De Bruyn at the University of Antwerp and its main application domain is currently the development of complex software systems – for example, information systems for business organizations. The Normalized Systems Theory book [1] describes in detail the theoretical background, the principles, and the possible practical usage of the theory. The key concepts from the book are presented in this chapter.

1.1 The Concept of System Evolvability

Before starting to explore what evolvable systems are, it is necessary to define what evolvability itself does mean. Evolvability is a term that comes from biology and describes "the ability of a population to produce variants fitter than any yet existing" [2]. The concept of system evolvability is being developed since the 1990s and is being applied, inter alia, in the domain of software engineering.

One of the most significant problems with complex software systems is the need to maintain them: it is sometimes required to implement a new feature or replace an obsolete framework with a modern one. Because of the complexity, such improvements may require an enormous amount not only of time, but also money. The theory of system evolvability tries to invent the means how to solve this problem and develop complex systems in a way that would guarantee the efficiency of their maintenance. There are several definitions of system evolvability, which are summed up in a study by Borches and Bonnema [2], and the common trait of all these definitions is mentioning the ability of a system to deal with changes – in other words, systems should be adaptable. As an example, a simple but apposite definition by Rowe and

Leaney [3] can be cited: "System evolvability is a system's ability to withstand changes in its requirements, environment and implementation technologies".

1.2 Modularity and its Relationship with Evolvability

As the authors of NST point out, business organizations, which are nowadays fully dependent on software and information systems, are "in need of evolvable software systems" [1, p. 126], and the proposed solution to system evolvability is based on modularity.

Modularity is a concept of decomposing systems into modules – either in a hierarchical or recursive way [1, p. 22]. Modern software systems are built with respect to modularity principles: classes and data structures, methods or functions, libraries and packages are all examples of the modular approach to developing a software system. The hierarchical structure is most obvious in the concept of inheritance, which is one of the basic features of object-oriented programming.

Although introducing modularity principles to a system's design is supposed to be the way of how to improve the evolvability of a system, it is not always the case. Changes that are being applied to a system over time usually lead to the increasing complexity of the system. This process is known as Lehman's law: a system that is being continually changed becomes more and more degraded, which leads to a decrease in its evolvability [1, p. 127].

There are two main rules which are used to design modular systems: low coupling and high cohesion. Low coupling is a concept of keeping inter-modular dependencies as low as possible. The high cohesion rule is about intra-modular dependencies and it tells that elements of a single module should be related.

Violating these rules may lead to the occurrence of so-called ripple effects. It means that additional modifications have to be made to a system as the cause of other modifications – for example, adding a new parameter to a method would cause the necessity of updating all places in the code where this method is called. Combinations of such changes in modules performed due to ripple effects are known as combinatorial effects [1, p. 327]. Combinatorial effects are then the cause of what is described above as Lehman's law: maintaining and evolving a system becomes more demanding. The bigger a system is, the more time it will require to implement a change. On the contrary, in a system that is free from combinatorial effects, implementing a change will take the same amount of time no matter how big the system is.

Normalized Systems Theory tries to come up with a proposition, how to design modular software systems that are free from combinatorial effects and thus remain evolvable over time. It works with four design theorems

which ought to ensure the two rules of modularity (i.e. low coupling and high cohesion) and provide the stability of a software system:

- separation of concerns,
- data version transparency,
- action version transparency,
- separation of states.

Although these theorems are associated primarily with software engineering, the ideas behind them are generic enough to be utilized in other domains.

Separation of concerns theorem says that to ensure the stability of a software system, each function can only have one task. In other words, the processing modules of a system should have only one responsibility, or concern.

Data version transparency is a principle of keeping data structures transparent: if a data structure has several versions, the difference between these versions should not affect the way of how the data structure is passed to and processed by a function.

A similar description can be used for the Action version transparency theorem: a change in a processing function should not affect how this function is called by other functions.

Applying the Separation of states principle implies that state keeping has to be ensured when calling a processing function in another function. State keeping means that states should be stored in separate data structures, so that functions can access them when necessary.

The last term that should be mentioned in the context of systems modularity is cross-cutting concerns. Cross-cutting concerns are additional functionalities of a system that cut across the primary functional structure [1, p. 329]. In a software system, these can be for example access controls implemented in several different functional modules of a system.

1.3 NST Applications outside Software Systems Domain

The authors of NST suggest that the principles of evolvable modularity have potential outside information or other software systems' domain. As an example, they give both physical artifacts (like roads and houses built from blocks or modular transport architectures) and conceptual domains (accounting and pedagogical systems, electronic or paper documents). As the aim of this thesis is to improve the evolvability of study materials, we will take a closer look at treating electronic documents as modular structures in this section.

1.3.1 Document Evolvability

The current approach for managing different versions of a single electronic document is to store them in document management systems. In these systems, each document version is kept as a whole document, and it is basically an opposite to the principle of version control systems used to manage source codes (e.g. git), where single text lines serve as modules. Both approaches do not correspond with logical document structures, as almost every text document consists "naturally" of modules – which typically are chapters, sections, paragraphs or blocks of source code.

Managing documents as one single module is the source of ripple effects. Imagine a study text which is distributed as a PDF file, a web page and a presentation at the same time: when a definition present in this study text has to be updated, this must be done at three places, because the text of the definition is "hard-coded" in each version of the study text separately. However, if the definition were kept in an independent "reference" document and referred to from the PDF, the web page and the presentation, the change in the definition would require only an update of this single "reference".

There are, nevertheless, examples of module-oriented approaches in the area of text documents. One of such examples is the LaTeX system, which enables to include .tex documents into other ("higher level") .tex documents or use different layouts and styles for the same text content. Although LaTeX has certain limitations regarding the modularity of created entities, the authors of NST see a potential of this system for improving the evolvability of documents [1, p. 471].

The cross-cutting concerns in the domain of modular documents may be, for example, references from modules to other modules (like sections, figures or bibliographic entries), or connections between text modules and their properties (like language or style). Such cross-cutting concerns should be independent, which means that changes in features provided by one concern would not affect the functionality granted by other concerns.

G. Oorts [4] defines evolvable documents as "documents that do not hinder or limit the application of changes made to their structure or content". He explains that cross-cutting concerns should be encapsulated in documents in the same way as they are encapsulated in software systems: there should be no dependencies between them and they should not affect or be affected by the base text modules [5].

There are further studies on the document evolvability – for example, [6] and [7], which propose that the key, how to keep documents evolvable, are the principles of modularity, separation of concerns and low coupling.

1.3.2 Implication for the Tool Implementation

To sum up the information provided in this chapter, a solution that is based on the principles of modularity and the encapsulation of the cross-cutting concerns should be proposed to comply with the aim of this thesis.

Chapter 2

Current Approaches to Create Study Materials at FIT CTU

The following chapter describes the state of the art of how study materials at the Department of Applied Mathematics at FIT CTU are created. There are several types of study texts which are distributed to students mostly in a digital format: each subject has either its own presentation slides, which serve as the basis for lectures, or textbooks (scripts); most of the subjects have both. Besides the scripts and the presentations, which are typically available as PDF documents, some of the subjects offer exercises in the MARAST system [8] and youtube videos with recorded lectures.

In this thesis, the focus will be set on text materials, primarily on the scripts. Generally, LaTeX is commonly used. While presentation slides are generated from .tex files, the source files for creating the PDF scripts are in a special WooWoo document format, to which the following section is dedicated.

2.1 WooWoo Documents

WooWoo documents are text files with a .woo extension. These files serve as input for a special program[1], which processes them and generates a text output in the given format (PDF or HTML). Universal WooWoo format has an abstract structure, which can be specified by templates. The "FIT template" is the realization used at FIT CTU.

The following description of the WooWoo format is based on the WooWoo Specs [9] and deals with the general document structure. For the detailed syntax description, refer to [9].

Each WooWoo document, regardless of template specifications, consists of document parts with a type, a title and optional meta-blocks. The structure is hierarchical: document parts can contain sub-parts and so on.

[1]The program is called WooWoo and is developed by Ing. Tomáš Kalvoda, Ph.D.

Document parts are further split into blocks (text-blocks) and objects. Each part is separated by at least two empty lines. WooWoo objects have a type, optional meta-blocks and a body. The body of an object is made of blocks. Blocks, which are consistently indented lines of text that may have so-called inner environments and be combined with outer environments, can thus be a part of an object or a stand-alone element in a document.

Inner environments serve to annotate small parts of the text (e.g. references or footnotes) and are written directly in text-blocks. Outer environments, on the contrary, resemble with their structure WooWoo objects and are suitable for larger parts of a text, like for example equations.

2.2 FIT Template

The FIT template defines the following document parts:

- Chapters
- Sections
- Subsections

with one obligatory meta-data field *label*.

Object types specified by the template are currently ten:

- Definition
- Theorem
- Lemma
- Proof
- Corollary
- Remark
- Example
- Question
- Figure
- Table

Meta-data fields for most of the objects are *label*, *title* and *index*.

The FIT template defines further a range of inner and outer environments, as for example `.cite`, which is used to refer to external sources, or `.footnote` to create a footnote (inner environments), `.equation` to write mathematical equations or `.caption` to create a caption for the Figure and Table objects (outer environments).

An example of a FIT template chapter follows:

```
.Chapter Title of the chapter
  label: chapter_label_1

.Section Title of the section
  label: section_label_1

This is the beginning of a block.
In the FIT template, such blocks are considered to be
paragraphs of text.
Here is the end of the block, another block is in
.reference:subsection_label_1.


.Definition:
  label: example_definition
  title: An example of a Definition object
  index: definition!example

  This is a Definition object.
  The lines within objects are consistently indented.
  Index field is used to build an index of notions.


.Subsection Title of the subsection
  label: subsection_label_1

This block has an inner environment "some text".quoted
which puts quotation marks around the words [some text].


.Example:

  This is an Example object. The object has no metadata
  fields, as they are optional.

  .enumerate:

    * Item
    * Another item

  Above is an example of "enumerate" outer environment,
  which creates enumerated lists.
```

Chapter 3

Analysis and Design

The aim of the tool which is subject to this thesis is to improve the evolvability of the study materials created for the mathematics subjects at FIT CTU. As mentioned in the previous chapter, these materials include not only scripts and presentations, but also video recordings of lectures and tutorials as well as exercises available through the MARAST web pages. The first part of this chapter analyzes the problems of the current approach and defines requirements for the tool. In the second part of the chapter, use cases, activity diagrams, an object state machine and a domain model of the tool are discussed.

The following section presents the problem identification and general tool requirements that arose from the discussion with Ing. Tomáš Kalvoda, Ph.D. and doc. Ing. Štěpán Starosta, Ph.D. from the Department of Applied Mathematics.

3.1 The General Idea behind the Tool

The problem with the current approach is that it is hard to maintain the study materials in a consistent form. For example, when a definition of a term is changed, this change might require updates in other entities, like exercises or figures, which refer to the term. However, to perform such additional updates, one has to search through the materials for the references to the changed definition. If an exercise that should be updated were accidentally left out, this would create an inconsistency between the definition and the exercise.

This is an evolvability obstacle that must be distinguished from the one described in section 1.3: it is not about how to avoid modifying the same definition at three different places, but how to determine, where changes caused by another change have to be made. Although there is a certain duplication of study materials (for example, presentation texts are often identical – or at least very similar – with significant parts of textbooks), it is not required to

unify such identical parts throughout all the types of study materials. On the contrary, it is sometimes desired to have the same content in a slightly modified form depending on where this content is placed (a presentation, a script, etc.) – for example, due to stylistic reasons.

The general idea of the tool to be developed is that it should be able to bind objects[2] present in study texts with references to them. This should be done on two levels:

1. the level of a single document (e.g. scripts for a subject),
2. the level of all study materials created for a subject – and possibly other subjects, so that one would be able to track references from subject A to an object in subject B.

The tool must be able to store the relations between objects (i.e. which objects refer to a particular object, or which objects a particular object points to) and track changes made to objects. When a change to an object is detected, the tool must point the user to all the objects that might be affected by this change.

3.2 Tool Requirements

The tool to be developed is a prototype and will focus on the relations between objects on the level of a single document. However, the tool shall be designed in a way that it would be possible to extend the functionality to the inter-subject level and to support different types of study materials. The type that has been chosen as the basis for the prototype are text scripts – for the following reasons:

- among all the study material types, it has the richest structure in terms of the number of objects and references between them;
- the source files for the scripts are in the WooWoo format, which has a naturally modular structure (with modules being WooWoo objects and text-blocks);
- a further advantage of WooWoo files is that the program processing them and outputting PDF or HTML documents is capable of generating a list of relations that can be found within the produced document – this list will be described later in the section.

The idea applied on WooWoo documents can easily be transferred to LaTeX documents, which are used to create presentation slides, as these documents can be split into modules that would correspond with the WooWoo objects and blocks.

[2] "Objects" denote in this thesis entities like definitions, theorems, examples, etc.: they correspond with WooWoo objects defined by the FIT template – see section 2.2.

3.2.1 Tool Input and Output

As mentioned earlier, the WooWoo program can generate a list of object relationships – this is done based on the `.reference` inner environment. The list is in JSON format [10] – it is a JSON array with a *data* key containing JSON objects with the following structure:

```
{
  "title": "Study Subject Name",
  "code": "BI-SSN",
  "timestamp": "2021-05-04 12:01:02 +0200",
  "data": [
    {
      "type": "Definition",
      "label": "definition_1",
      "title": "Example of a Definition",
      "filename": "file_1.woo",
      "line": 10,
      "hash": 098765432109876543,
      "points_to": [

      ],
      "referenced_by": [
        "Example.1.123456789012345678"
      ],
      "content": "This is a definition."
    },
    {
      "type": "Example",
      "label": "Example.1.123456789012345678",
      "title": "",
      "filename": "file_2.woo",
      "line": 22,
      "hash": 123456789012345678,
      "points_to": [
        "definition_1",
        "Example.2.567890123456789012"
      ],
      "referenced_by": [
        "Example.3.789012345678901234"
      ],
      "content": "This is an example."
    }
  ]
}
```

In the above example, there are two JSON objects that represent two WooWoo objects in a study text: the first one is a Definition object and the second one is an Example object, which is reflected in the JSON object's element with the key *type*. The currently supported types are: Definition, Theorem, Lemma, Corollary, Proof, Example, Remark and Paragraph (which stands for the WooWoo text-block).

The key *label* corresponds with the meta-data field *label* defined by the WooWoo FIT template. If this field is not present, the *label* value is generated as [ObjectType.Index.Hash] – as shown in the Example object.

The key *title* corresponds with the meta-data field *title*. If this field is not present, the value is an empty string.

The *filename* element value contains the name of the file, and *line* is the line number in the file, where the object is to be found.

hash is a hash value of the WooWoo object contents – this value shall serve to determine, whether the object has been modified.

The *points_to* array is a list of object *labels* that the object refers to. In the given example, "definition_1" does not refer to any object, and "Example.1.123456789012345678" refers to the Definition "definition_1" and an Example "Example.2.567890123456789012".

The *referenced_by* array is a list of object *labels* that refer to the object. In the example, "definition_1" is referenced by "Example.1.123456789012345678", which is referenced by "Example.3.789012345678901234".

Text contents of the object are stored under the *content* key.

Besides the *data* JSON array, there are also "global" *title*, *code* and *timestamp* key-value pairs in the JSON file. *title* is the name of the subject, for which the study text is created, and *code* is its faculty code. *timestamp* is the date when the JSON list was generated, in a `(yyyy-MM-dd HH:mm:ss Z)` date format.

The input for the tool shall thus be a JSON file with a relationship list as described above.

The output will be a list of changed objects together with their relationships – i.e. those objects which reference the modified object.

3.2.2 Functional Requirements

This section defines functional requirements that the tool must meet.

Func-01: Input Data Format
The tool shall be able to process JSON relationship lists generated by the WooWoo program and store the contents of such lists in its internal object representation.

Func-02: Object Persistence
The tool shall store objects parsed from the input JSON relationship list, if

they are not already present in the tool's database. Objects shall be uniquely identified by the label, which shall be identical with the *label* value of the JSON object.

Func-03: Object Presentation
The tool shall display objects from its database to the user.

Func-04: Object Search
The tool shall support searching for objects by their label and title.

Func-05: Object Versions Comparison
The tool shall compare the hash of the object processed as input against the hash of the internally stored object with the same label, if such is present. If the hash values differ, the tool shall consider the object as "modified".

Func-06: Changes Presentation
The tool shall present to the user the objects that have been marked as "modified" together with a list of objects that reference the changed object. The presented information about the objects shall contain, at a minimum, the following: object type, label and title (if not empty) as well as the name of the file where the object is placed.

Func-07: Changes Confirmation
The user shall be able to confirm that they have checked the objects related to the "modified" object and want to proceed.

Func-08: Object Deletion
The user shall be able to delete objects from the tool's database.

3.2.3 Non-Functional Requirements

In this section, non-functional requirements for the tool are determined.

Nonfunc-01: Target platform
The tool shall run on desktop and laptop devices with installed JVM.

Nonfunc-02: User Interface
The tool shall support graphical user interface.

Nonfunc-03: Logging
The tool shall log important events, warnings and errors, and store the log file in the tool's working directory.

3.3 Use Cases

The section defines the use cases that are summarized in the Use Case diagram in the figure 3.1.

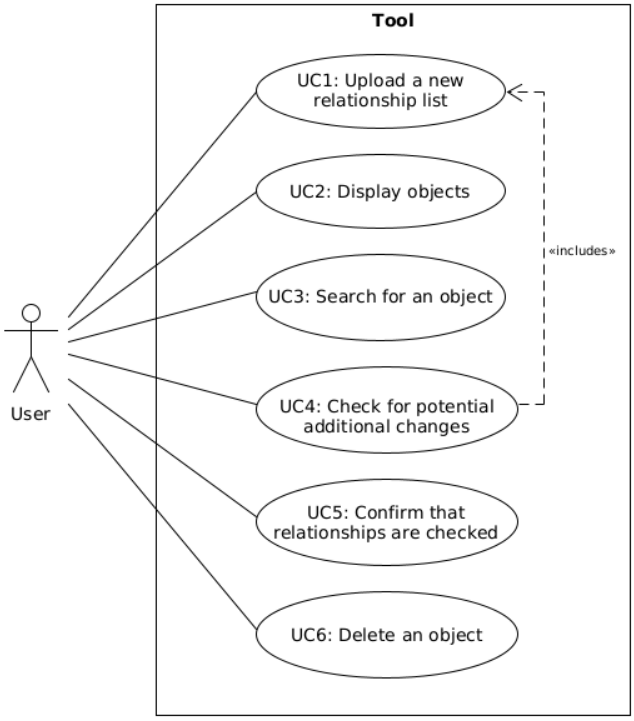

Figure 3.1: Use case diagram

3.3.1 UC1: Upload a new relationship list

The use case enables the user to upload JSON relationship lists generated by the WooWoo program into the tool.

Actors: User

Precondition: A new JSON relationship list exists. This can be a completely new list (e.g. for a newly added subject) or a list generated after certain changes to source files have been performed.

Basic Flow:
1. The user uploads a JSON relationship list.
2. The tool processes the list and stores its contents.

3. The tool notifies the user that the list has been uploaded.

Post-condition: Objects from the uploaded relationship list are stored in the tool's database.

3.3.2 UC2: Display objects

The use case enables the user to display objects from the tool's database.

Actors: User

Precondition: There are objects persisted in the tool's database.

Basic Flow:
1. The user displays the list of object labels.
2. The user selects a label from the list.
3. The tool displays the object's data – such as type, title, filename, contents and relationship lists.

3.3.3 UC3: Search for an object

The use case enables the user to search for objects in the tool's database by their label or title.

Actors: User

Basic Flow:
1. The user chooses if the search shall be performed by label or by title.
2. The user enters the label (title) of the object that they want to search for.
3. The tool displays the found object(s) to the user.

Alternative Flow:
3.1. There are no objects meeting the search criteria in the tool's database – the tool displays a message that no objects were found.

3.3.4 UC4: Check for potential additional changes

The use case enables the user to check if objects referring to a modified object have to be changed as well.

Actors: User

Precondition: A JSON relationship list has been generated after some changes to source files have been made, and this list is uploaded into the tool (see UC1).

Basic Flow:
1. The tool detects a change in an object and marks it as "modified".
2. The tool notifies the user of a change in the object and shows them object relationships that might be impacted by the change.
3. The user checks the relationships and performs updates if necessary.

Post-condition: The contents of the changed object and its relationships are in a consistent state.

3.3.5 UC5: Confirm that relationships are checked

The use case enables the user to confirm that they have checked the relationships of a modified object and made necessary updates.

Actors: User

Precondition: The user has checked the relationships of the modified object presented by the tool.

Basic Flow:
1. The tool prompts the user to confirm that they have checked the relationships.
2. The user confirms the relationships are checked.
3. The tool marks the object as "checked".

Post-condition: The tool no longer notifies the user about the modified object, if it has been confirmed as "checked".

3.3.6 UC6: Delete an object

The use case enables the user to delete an object that is no longer present in study materials.

Actors: User

Precondition: The object to delete exists in the tool's database.

Basic Flow:
1. The user selects the object from the list of stored objects.
2. The user clicks on the delete button.
3. The tool asks to confirm the deletion.
4. The user confirms the deletion.

Alternative Flow:
1.1. The user searches for the object by label or title.

Post-condition: The object is deleted from the tool's database.

3.4 Use Case Mapping

Table 3.1 summarizes the mapping of the use cases to the functional requirements listed in the section 3.2.2.

	UC1	UC2	UC3	UC4	UC5	UC6
Func-01	+					
Func-02	+					
Func-03		+				
Func-04			+			
Func-05				+		
Func-06				+		
Func-07					+	
Func-08						+

Table 3.1: Use Case Mapping

3.5 Domain Model

This section explains the design of the domain model, which is shown on the figure 3.2.

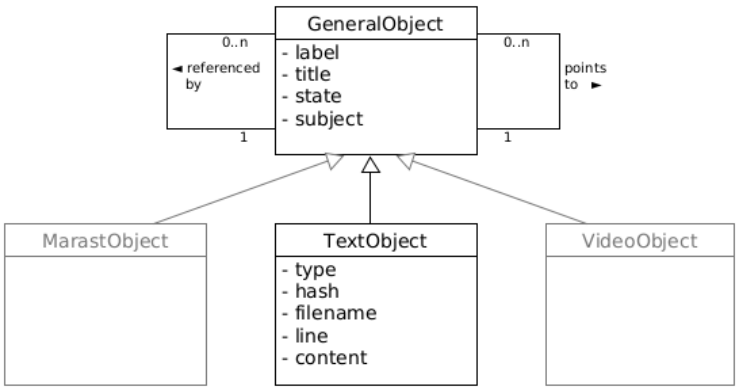

Figure 3.2: Domain Model

GeneralObject is an abstract entity with four attributes: *label*, *title*, *subject* and *state*. The attributes *label* and *title* shall be mandatory for every object stored by the tool – so that the user can search for objects based on these attributes. The *title* attribute may be an empty string, whereas *label* cannot be empty. Every GeneralObject instance comes from a specific study subject, which is reflected by the *subject* attribute. The *state* attribute keeps the current state of the object. Each GeneralObject may point to or be referenced by any other GeneralObjects.

TextObject represents an object in a text document, i.e. a WooWoo or a LaTeX source file. The attributes *type*, *hash*, *filename*, *line* and *content* correspond with the key-value pairs of objects in a JSON relationship list. These attributes are considered text-document specific, and thus are not a part of the abstract GeneralObject.

In the future, the model shall be extended to contain entities like MarastObject for exercises from the MARAST system and VideoObject for video recordings of lectures – these entities are grayed out, as they are not in the scope of the tool prototype.

3.6 State Machine

Figure 3.3 shows the possible states of GeneralObject entities (further referenced simply as *objects*). When a new object is parsed from a JSON relationship list and saved to the tool's database, it is in the *stored* state.

If an object parsed from a JSON relationship list is already stored and a change in the object's contents is detected, the object transfers to the *modified* state. The user is notified about modified objects and should check their relationships.

After the user has checked the relationships and performed necessary changes, they confirm this, and the object moves to the *checked* state.

When the user finishes checking the objects, they save the changes, and the *checked* objects move back to the *stored* state. If there are any *modified* objects left upon saving the changes, these objects remain in the *modified* state.

If the user wants to delete an object from the tool's database and the object's "referenced by" list is not empty, the object is not removed from the database right away, but moved to the *deleted* state. After all references to the deleted object have been removed, the object is erased from the database.

3.7 Activity Diagrams

In this section, two workflows are presented. They describe the process of how objects are uploaded into the tool from a JSON relationship list and how the user can search for an object and delete it.

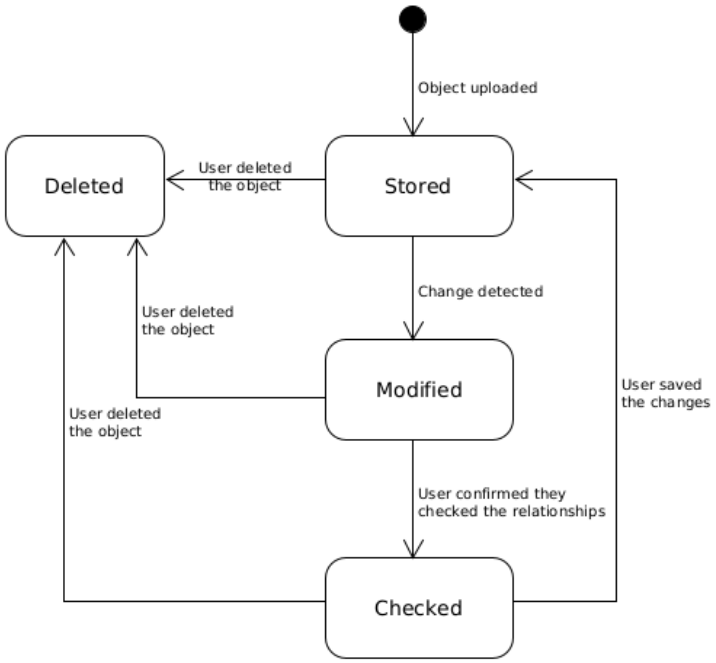

Figure 3.3: Object state machine

3.7.1 JSON Relationship List Upload

The first workflow, uploading a JSON relationship list, can be seen in figure 3.4.

In this workflow, the user first generates a JSON relationship list and uploads it into the tool. The tool parses single objects from the list. The first expansion region[3] describes actions and states of each parsed object.

If the parsed object is not yet in the tool's database, the object is stored and the tool proceeds to parsing another object.

If the parsed object is already in the database and its contents have not changed, no actions to the object are taken.

If the object's contents are different from the ones stored in the database, the object is marked as modified and the user is notified that they should check the object's relationships. The user can either skip checking the object – then it remains in the *modified* state, or check it and perform necessary updates to its relationships.

After the user has checked the object, they confirm it and the object moves to the *checked* state. It remains in this state until the user finishes working

[3]Expansion regions execute actions over a collection of values. The *iterative* mode means that the actions are performed in an iterative way[11].

with the tool and saves the performed changes. Upon saving the changes, the *checked* objects transfer to the *stored* state and their contents are updated in the tool's database, as shown in the second expansion region.

3.7.2 Search and Delete an Object

The second workflow in figure 3.5 describes object search and deletion.

The user can choose whether they want to search by *label* or *title*, then they enter the keyword they want to search for. If no results in the database match the keyword, a "No results found" message is displayed.

If an object is found and the user wants to delete it, and the object's "referenced by" list is empty, the object is removed from the database. If the "referenced by" list is not empty, the object is moved to the *deleted* state. The user can then change the objects that reference the deleted object and re-upload an up-to-date JSON relationship list, so that the *deleted* object's references are updated. A *deleted* object with an empty "referenced by" list is erased from the database.

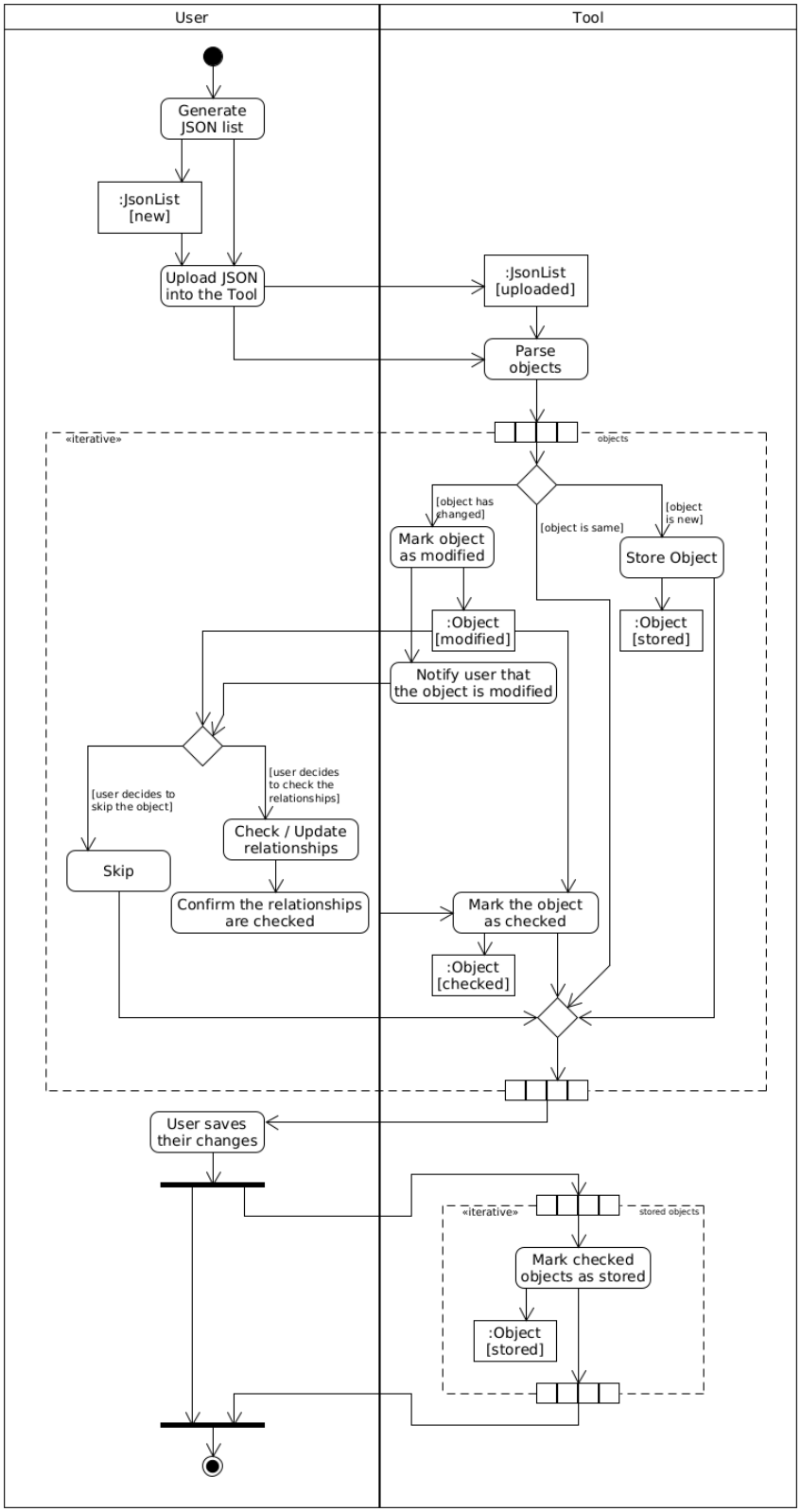

Figure 3.4: Activity Diagram – Upload a JSON list

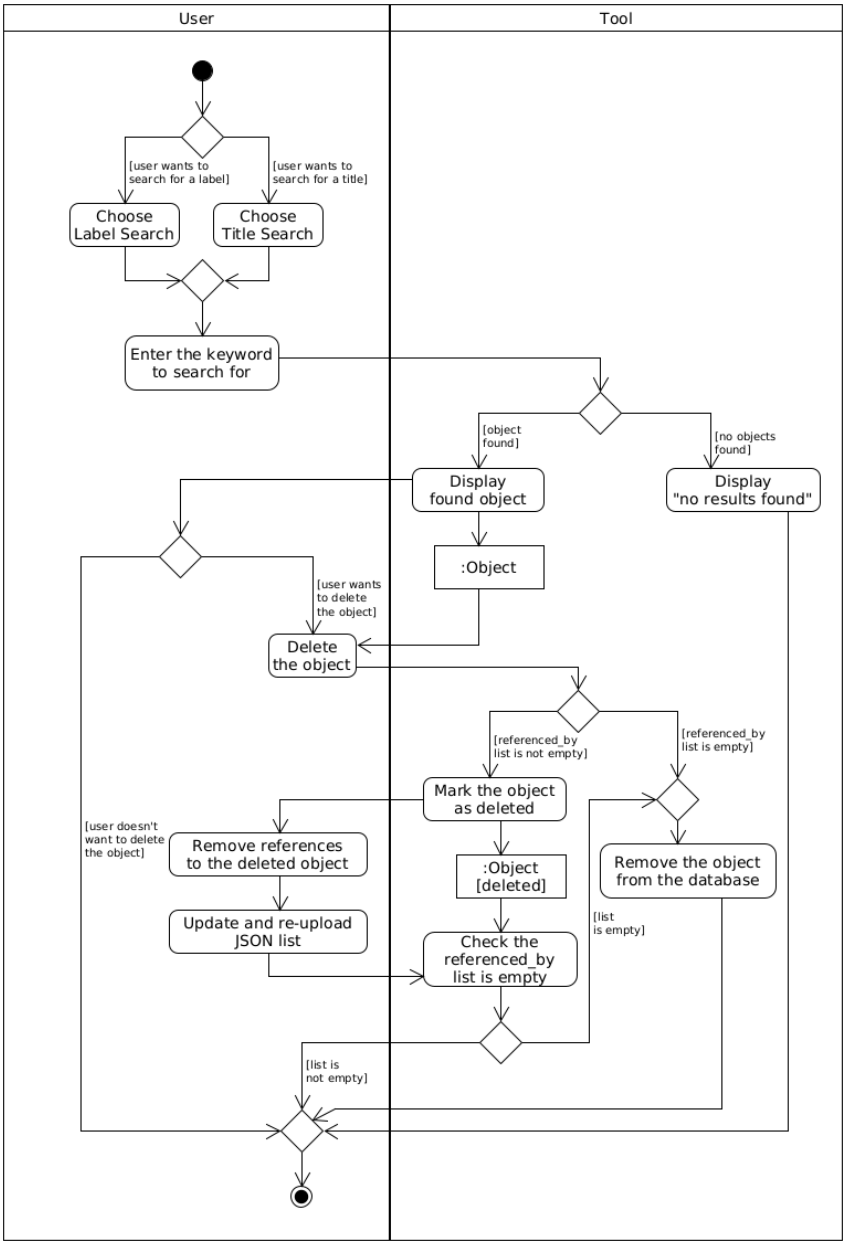

Figure 3.5: Activity Diagram – Search and delete an object

Chapter 4

User Interface Design

The tool shall support Graphical User Interface. The chapter presents the graphical design of the tool.

Figure 4.1 shows a Welcome Window wireframe. This is the first window that shall be displayed when the tool is started. The user can choose from two actions: either upload a JSON relationship list or go to the Main Window and browse the stored objects.

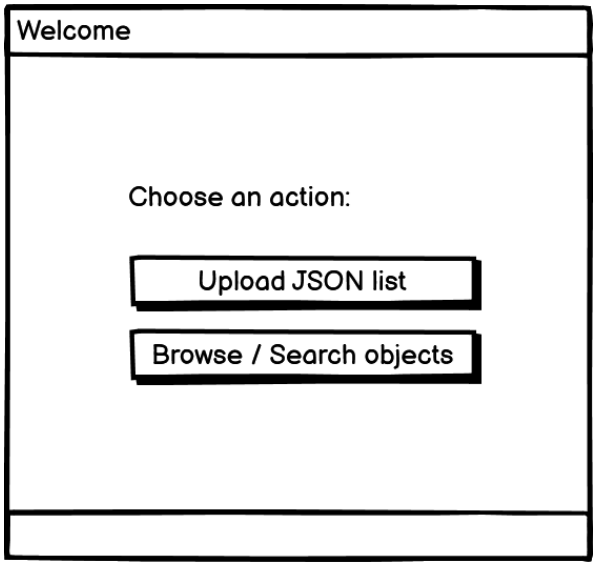

Figure 4.1: Wireframe – Welcome Window

If the user chooses to upload a JSON list, the Upload Window (see figure 4.2) is displayed. The user can then choose the path to upload the JSON list from. After the list has been uploaded, the Main Window is displayed.

The Main Window is shown in figure 4.3. It is divided into two parts: the left part is the list of the stored objects, the right part contains a search

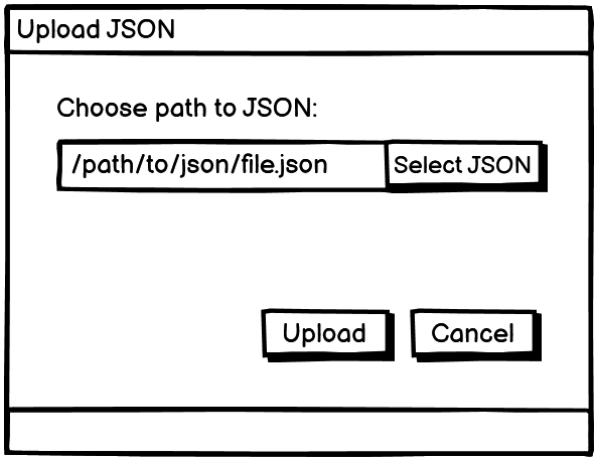

Figure 4.2: Wireframe – Upload Window

panel and a text panel where the contents of a selected or found object are displayed.

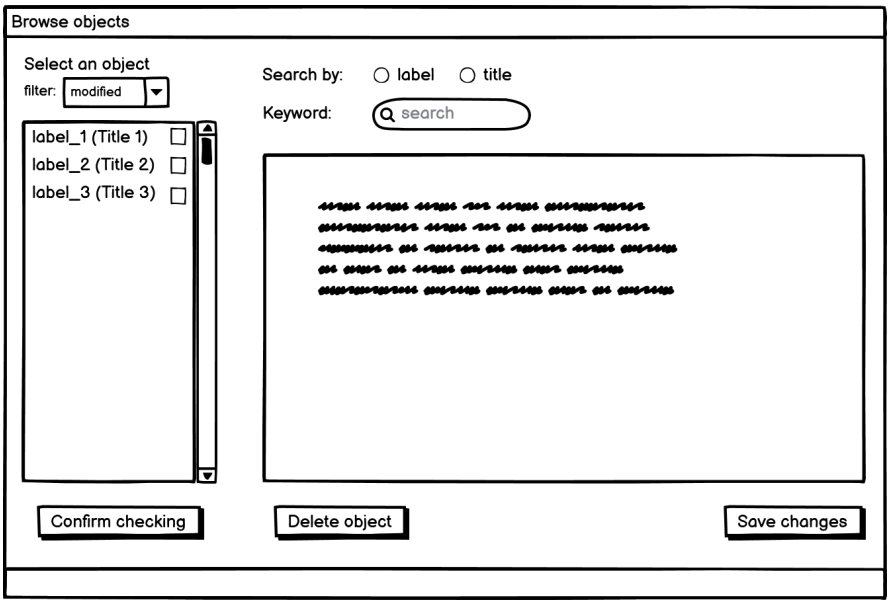

Figure 4.3: Wireframe – Main Window

The objects in the list are represented by a string `<label> (<title>)`, or just `<label>`, if the object's title is empty. They can be selected by clicking on them – then the object's contents are displayed in the text panel. If an object is selected in the list, the Delete Object button becomes enabled and the user can delete the selected object.

The state of an object is reflected by its color in the object list:

- black color means the object is *stored*;
- blue color means the object is *modified*;
- green color means the object is *checked*;
- gray color means the object is *deleted*.

If an object is in the *modified* state, a checkbox right from its label in the object list is enabled. The user selects the objects they have checked (meaning they have controlled and updated their relationships) by clicking on the checkbox. They confirm checking the object(s) by clicking the Confirm Checking button. After the user has finished controlling the objects, they click on Save Changes button.

To search for an object, the user shall choose from the *label*/*title* options and enter the keyword they want to search for. If an object is found, it will be displayed in the text panel.

When the user clicks on the Delete Object button, a confirmation prompt (figure 4.4) is shown.

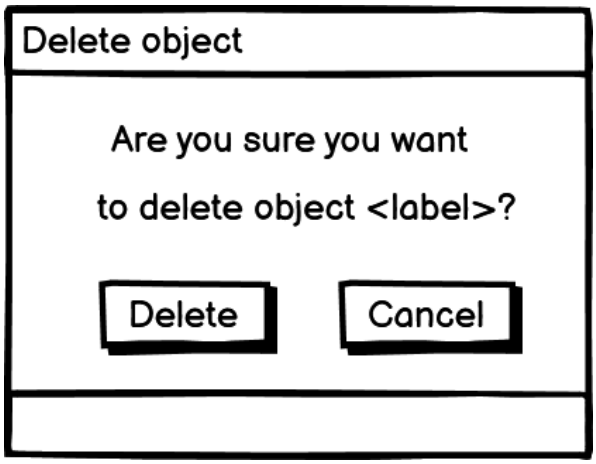

Figure 4.4: Wireframe – Delete Window

Chapter 5

Implementation

The chapter provides the details of how the tool, called RefTool, was implemented.

5.1 Implementation Language and Frameworks

The tool source code is written in Java using a JDK version 14. To define the tool's dependencies, build and deploy the tool, Gradle Build Tool v7.0 [12] was used.

Following plugins and dependencies are defined in the build.gradle file:

- `org.openjfx.javafxplugin` for the JavaFX plugin required to run the tool's UI [13].
- `com.google.code.gson` for the Gson library that is used to convert Java objects to JSON objects and vice versa [14].
- `junit` for the JUnit test framework which enables developers to create and run unit tests [15].

5.2 Architecture

The tool's architecture follows the Model-View-Controller design pattern. When the tool is launched, an instance of the Tool class extending the JavaFX Application class is created. In the overridden Application::init() method, the ToolModel is initialized and the data from the object's persistent storage are loaded.

Figure 5.1 shows the class diagram (generated by IntelliJ IDEA IDE; most of the dependencies are not shown for the sake of simplicity).

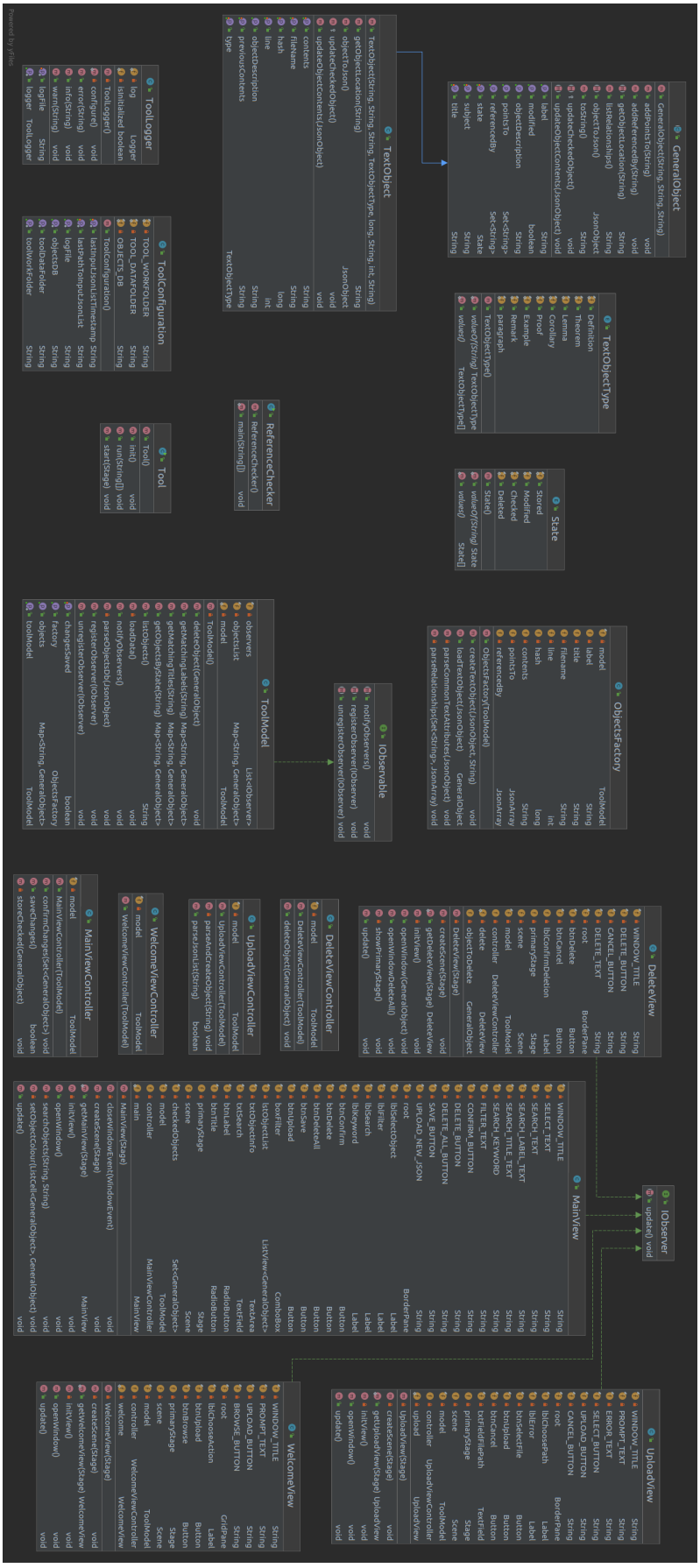

Figure 5.1: Class diagram

5.2.1 Model

The ToolModel class is a singleton implementing an IObservable interface to notify classes in the View layer (that implement an IObserver interface) about the changes in the Model – these notifications follow the Observer behavioral design pattern. The ToolModel is responsible for transforming instances of GeneralObject class children into JSON objects and backwards, loading the JSON objects into and from the tool's persistent storage, saving user changes and deleting objects. It also provides an API to list objects, perform searching by label and title in the list of objects or filter objects by state.

GeneralObject is an abstract class representing all the stored objects. In the current tool prototype, its only child is the TextObject class which stores objects parsed from WooWoo-generated JSON files. GeneralObject defines methods to get an object description or location, update object's contents or transform object's attributes into JSON data – all based on the object's real class. It is further responsible for listing object's relationships and checking the state of an object.

5.2.2 View

There are four View classes in the tool that correspond with the application windows as they were proposed in the User Interface design: WelcomeView, UploadView, MainView and DeleteView. The View classes define the scene layout using JavaFX UI elements. The UI elements handle user actions by passing them to Controller classes.

The MainView class, which serves to display object data to the user, responds to changes in the Model by updating the UI elements representing the objects.

5.2.3 Controller

Each View class has a Controller class associated with it. When a user action is registered and an action on the tool model's data needs to be performed, this is handled by a Controller.

UploadViewController validates the input of the Path-to-Json-File bar and passes data to a factory that creates an object.

MainViewController triggers object's state changes and handles saving the changes.

DeleteViewController calls Model methods that delete the selected object or all objects based on which user action was performed.

5.3 Objects Factory

The ObjectsFactory class serves to implement the Factory creational design pattern. It provides methods to create instances of GeneralObject child classes from the data saved in the tool's persistent storage and from the data parsed from a JSON input file.

5.4 Configuration and Logging

The ToolConfiguration class is used to store tool configuration data, like the path to the file with saved objects and the log file.

ToolLogger is a class that sets up a `java.util.logging.Logger` instance and provides simple methods to log Info, Warning and Error messages into the log file.

5.5 Persisted Objects File Format

The tool's "database" is a JSON file with a root element representing the data that was saved by the user. It contains three elements:

- a JSON object *jsonPath* with the path to the last uploaded JSON input file;
- a JSON object *timestamp* with the timestamp from the last uploaded JSON input file;
- a JSON array *objects* with saved object data.

Comparing to the JSON objects of the JSON input file, the tool's internal JSON objects several additional values:

- *class*, which is the object's class in the tool implementation;
- *subject*, which is the study subject that the object is parsed from;
- *state*, which is the object's state upon saving the changes;
- *previousContent*, which is the object's previous content. It is an empty string unless the object is *modified*.

An example snippet from the file follows:

```
{
  "jsonPath": "/home/user/bi-zma.json",
  "timestamp": "2021-04-30 14:37:14 +0200",
  "objects": [
    {
      "class": "TextObject",
      "subject": "Základy matematické analýzy (BI-ZMA)",
      "label": "Theorem.22.-4268866053640503722",
      "title": "Vlastnosti obecné mocniny",
      "state": "Stored",
      "textObjectType": "Theorem",
      "filename": "03-rady.woo",
      "line": 777,
      "hash": 4268866053640503722,
      "content": "(Text of the theorem)",
      "previousContent": "",
      "points_to": [],
      "referenced_by": []
    },
    {
      "class": "TextObject",
      "subject": "Základy matematické analýzy (BI-ZMA)",
      "label": "def_limita_posloupnosti",
      "title": "Limita posloupnosti",
      "state": "Deleted",
      "textObjectType": "Definition",
      "filename": "02-posloupnosti.woo",
      "line": 407,
      "hash": 4045706248800058517,
      "content": "(Text of the definition)",
      "previousContent": "",
      "points_to": [],
      "referenced_by": [
        "veta_jednoznacnost_limity",
        "paragraph.491.3439574107469921445",
        "paragraph.490.-4185148331242362374",
        "Proof.6.2283156093901435788",
        "paragraph.337.-2325975530685237404",
        "paragraph.226.-3807412196376826373"
      ]
    }
  ]
}
```

5.6 Graphical User Interface

This section describes the UI control elements and their purpose. It can also serve as a user guide that explains how to work with the RefTool.

5.6.1 Welcome Screen

When the tool is started, the user sees the Welcome screen (figure 5.2). It prompts the user to choose an action that they want to perform: either upload a new file with a JSON relationship list ("Upload JSON list" button) or go to the main screen to work with the tool's persisted objects ("Browse / Search objects" button).

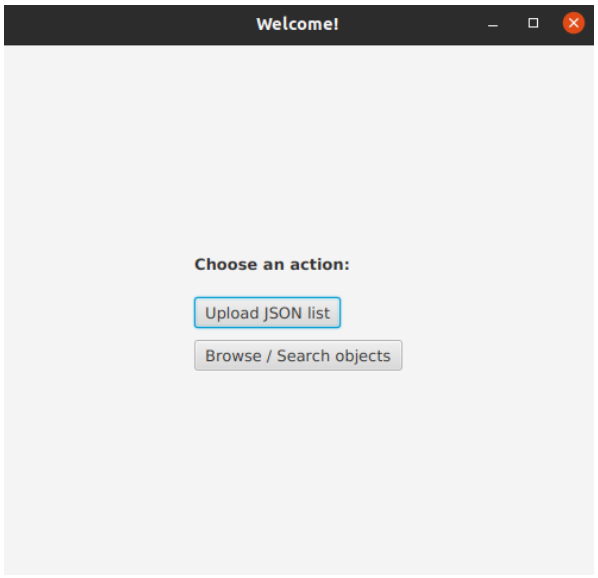

Figure 5.2: Welcome screen

5.6.2 Upload Screen

If the user clicks on the "Upload JSON list" button, the Upload screen (figure 5.3) is displayed. The screen provides a text field for entering the path to a JSON file to be uploaded and a button to select the file from the file system ("Select JSON"). To upload the selected file, the user should click the "Upload" button. The tool parses the given JSON file and transits to the Main screen.

If the "Cancel" button is clicked, the application returns to the Welcome screen.

If an invalid path is chosen or the selected file is not a JSON file, an error message (figure 5.4) is displayed.

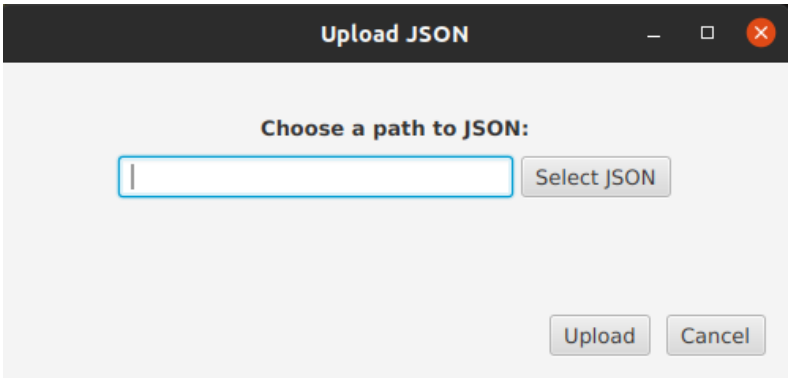

Figure 5.3: Upload screen

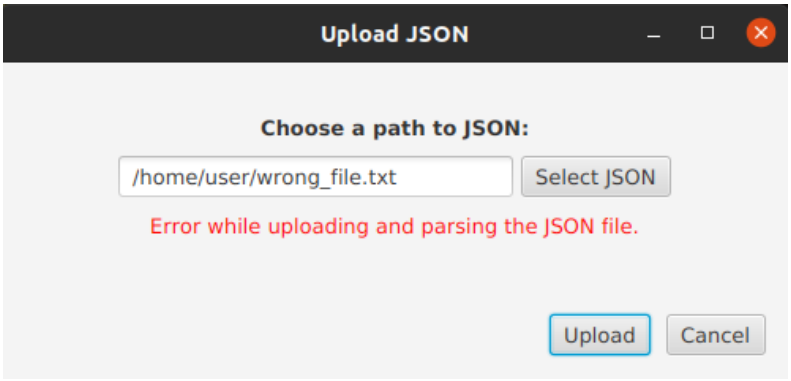

Figure 5.4: Upload screen – invalid file

5.6.3 Main Screen

The Main screen (figure 5.5) is divided into four areas: the top panel, the object list on the left side, the object field on the right side and the bottom panel with buttons.

The top panel serves to filter and search objects. On the left side, there is a "filter" drop-down menu that enables the user to filter the object list by state. On the right side, there is a search bar with radio buttons to choose searching by label or by title and a text field to enter a keyword to search for.

The left panel with the object list displays all objects stored in the tool's "database" in the format "label (title)" or "label" (if no title is present). The user can click on the object to display its contents in the object field. The object states are distinguished by colour: black objects are in the *stored* (neutral) state, blue objects are *modified*, green objects are *checked* and gray objects are *deleted*.

If an object is in the *modified* state and the user wants to mark it as *checked*, a checkbox on the left side of the object should be clicked. If the user

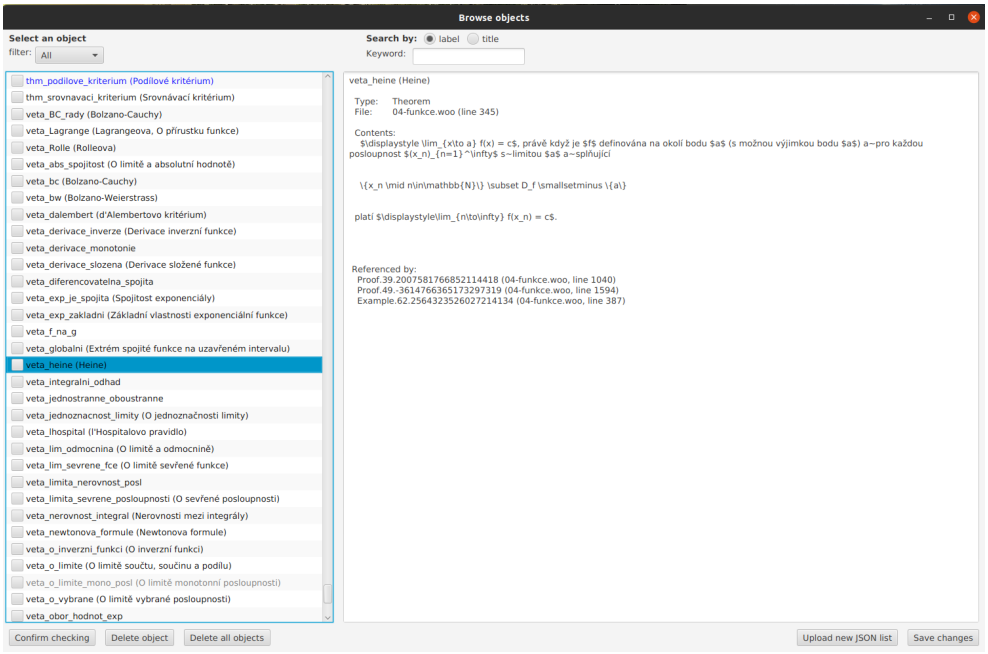

Figure 5.5: Main screen with uploaded data

has checked several objects, they mark all these objects with checkbox ticks. *Modified* objects that have been checked transit to the *checked* state after the user clicks the "Confirm checking" button.

If an object is selected, the "Delete object" button becomes displayed. This button serves to notify the tool that the object has been removed from the source texts and shall no longer be stored. There are two options: if the deleted object has no "referenced by" relationships, the object is deleted from the tool's "database" right away, otherwise it transits to the *deleted* state. This is to notify the user that there might be obsolete references to the removed object in the source texts that should be checked.

The "Delete all objects" button clears the whole object list.

If the user clicks the "Upload new JSON list" button, the Upload screen is displayed and the user can upload a new JSON file.

When the user finishes working with the tool, they should save their changes by clicking the "Save changes" button – otherwise, they would not be written into the "database". If the user wants to close the program without saving the changes, a warning message is displayed.

5.6.4 Delete Screen

The Delete screen (figure 5.6)is to confirm the action when the user clicks on the "Delete object" or "Delete all objects" button. If the user cancels the

action, the application returns to the Main screen.

Figure 5.6: Delete screen

Chapter 6

Testing

The implemented tool was tested to identify issues and bugs. Following test types were performed[4]:

- Compatibility testing to verify that the application can run on different operating systems;
- Unit testing to verify the correct behaviour of the implemented methods;
- System testing to verify that the tool meets the defined requirements;
- Usability testing to verify that the tool's UI is easy to operate.

Test setup, procedures and results of the executed tests are described in the following sections.

6.1 Compatibility Testing

The tool requires a Java SDK of version at least 14 and a JavaFX distribution to run the UI components. If the following requirements are met, the tool shall be able to run on any desktop operating system.

The tool was run on the following OS versions:

- Ubuntu 20.04.2;
- Linux Mint 19 Cinnamon;
- Windows 10 Home Edition.

On each system, the environment variable `JAVA_HOME` was set to point to a Java SDK that meets the tool requirements. Used JDKs:

[4]For a precise test types definition, refer to [16].

- OpenJDK 15 [17]
- JavaFX 11.0.2 [13]

To run the tool, the launch script distributed together with the tool binaries (*reftool* for Unix-like systems and *reftool.bat* for Windows OS) was executed.

6.1.1 Results

The tool was successfully launched on both Linux operating systems. The launching on Windows OS resulted in an exception caused by inability to find a Java class used by JavaFX. To fix the issue, a JDK distributed together with JavaFX was used instead of the initial OpenJDK 15 + JavaFX 11 setup: Zulu OpenJDK Java package "JDK FX" of version 15 [18]. After performing this change, the launch script was able to start the tool.

Except for the mentioned Windows JavaFX issue, no anomalies were found. Compatibility test passed.

6.2 Unit Testing

Unit tests to verify the expected method behaviour were created and run with usage of the JUnit [15] framework.

The tests cover:

- the tool configuration setup;
- the ToolModel class methods used to list objects and to filter objects by state, label and title;
- the GeneralObject and TextObject class methods used to create objects, get their contents and modify them.

6.2.1 Results

All tests passed, as the execution summary in figure 6.1 shows.

6.3 System Testing

Verification that the tool meets the requirements defined in sections 3.2.2 and 3.2.3 was performed.

Test Summary

7 tests	0 failures	0 ignored	0.496s duration	100% successful

Packages Classes

Package	Tests	Failures	Ignored	Duration	Success rate
cz.cvut.fit.referencechecker	1	0	0	0.015s	100%
cz.cvut.fit.referencechecker.model	3	0	0	0.420s	100%
cz.cvut.fit.referencechecker.model.objects	3	0	0	0.061s	100%

Figure 6.1: Unit testing results

6.3.1 Functional Requirements Coverage

Func Test Case 01: Input Data Format
- Started the tool and uploaded a JSON relationship list generated as the WooWoo program output.
- Verified that the tool displays the objects parsed from the JSON relationship list.
- Saved changes and exited the tool.
- Verified that a file $(HOME)/reftool/tooldata/objects.json was created and contains the parsed objects.

Func Test Case 02: Object Persistence
- Started the tool and uploaded a new JSON relationship list with the same contents as in TC 01 except for one newly added object.
- Verified that each object has a label presented to the user in the object's list.
- Verified that the newly added object is displayed by the tool.
- Verified that the previously added objects are not duplicated.
- Saved changes and exited the tool.
- Verified that the contents of the reftool/tooldata/objects.json were updated with the new object.

Func Test Case 03: Object Presentation
- Started the tool and navigated to the Main Window.
- Verified that the objects displayed in the tool's object list correspond in their contents with the ones stored in reftool/tooldata/objects.json.

Func Test Case 04: Object Search
- Selected "label" radio button in the search panel and typed "Definition" into the keyword bar.
- Verified that the tool filters the displayed objects to the ones that contain the "Definition" word in their label.

- Typed "thm" into the keyword bar.
- Verified that the tool filters the displayed objects to the ones that contain the "thm" string in their label.
- Selected "title" radio button in the search panel and typed "li" into the keyword bar.
- Verified that the tool filters the displayed objects to the ones that contain the "li" string in their title.

Func Test Case 05: Object Versions Comparison
- Uploaded a new JSON file based on the original JSON relationship list, but with several objects having their hash value changed.
- Verified that the objects with the updated hash changed their colour to blue.
- Set filter to display the Modified objects.
- Verified that only the objects with the updated hash are displayed.

Func Test Case 06: Changes Presentation
- Selected one of the Modified objects that has a non-empty "points_to" list.
- Verified that the "points_to" list is displayed and contains information about the objects pointing to the selected object.
- Selected one of the Modified objects that has a non-empty "referenced_by" list.
- Verified that the "referenced_by" list is displayed and contains information about the objects pointing to the selected object.

Func Test Case 07: Changes Confirmation
- Selected an object from the Modified list.
- Clicked the checkbox button next to the object to indicate that the object was checked.
- Clicked the "Confirm checking" button to confirm checking the object.
- Verified that the object moved from "Modified" to "Checked" state.

Func Test Case 08: Object Deletion
- Selected an object without "referenced by" relationships from the objects list.
- Clicked on the "Delete object" button.
- Confirmed the deletion.
- Verified that the object is no longer in the database.
- Selected an object with "referenced by" relationships from the objects list.
- Clicked on the "Delete object" button.
- Confirmed the deletion.
- Verified that the object was moved to the Deleted state.

6.3.2 Non-Functional Requirements Coverage

Non-Func Test Case 01: Target Platform
- The coverage of this requirement was performed as part of the Compatibility testing.

Non-Func Test Case 02: User Interface
- Verified that upon launching the tool, an instance of the class extending the JavaFX Application class [13] is created and the tool's UI is displayed.

Non-Func Test Case 03: Logging
- Verified that the tool logs messages of the Info, Warning and Error level into the tool's log file reftool/ref_tool_log.log

6.3.3 Results

All tests verifying the coverage of the functional and non-functional requirements passed.

6.4 Usability Testing

To test the usability of the tool and its interface, Ing. T. Kalvoda, Ph.D. and doc. Ing. Š. Starosta, Ph.D. were asked to execute a simple usage scenario. The test scenario steps are as follows:

1. Run the RefTool.
2. Upload bi-zma.json into the RefTool.
3. Display the object with the label "Definition.30.".
4. Search for the object with the label "thm_odhad_zdola".
5. Search for the object(s) that have title containing the word "funkce".
6. Upload bi-zma_upd.json into the RefTool.
7. Display objects in the Modified state.
8. Mark any Modified object as Checked and confirm checking.
9. Display objects in the Checked state.
10. Delete the object "Example.1.".
11. Delete the object "defi_inverze".
12. Display objects in the Deleted state.

13. Delete all objects.

14. Save changes and exit the RefTool.

The tool was distributed to the testers together with *bi-zma.json* and *bi-zma_upd.json* files. The *bi-zma.json* file contains the original version of the JSON list generated by the WooWoo program from BI-ZMA subject scripts; *bi-zma_upd.json* is an updated version of the original file, which has four objects with modified hash – this should simulate a scenario when the user changes these objects, re-generates the JSON relationship list and uploads it into the tool.

After performing the test scenario steps, the testers were asked to answer following questions:

A) Evaluate on scale 1–5 (1 = excellent, 5 = bad/faulty) and give the reason for the mark:

1. Simplicity of the interface (is it intuitive?);

2. Tool usage convenience (uploading files, searching, deleting);

3. Displayed object information (clarity, completeness).

B) Provide any remarks you might have on how to improve the tool.

The answers to the questionnaire are cited below.

Ing. T. Kalvoda, Ph.D.:
A1)
Given Mark: 3
Commentary: I was a little bit unsure about what each button (at the bottom of the window) does, and I understood the meaning of the colours in retrospect. The "state" of an object could be written somewhere in words.

A2)
Given Mark: 2
Commentary: It seemed good to me. The only thing, when uploading the second file, it could remember the directory from where the first file was uploaded.

A3)
Given Mark: 3
Commentary: I would add the information about the state of the object (I could maybe miss it), and I would probably graphically modify the description in the right part of the window a little bit (for example some parts in bold – title, label?), to distinguish them from the remaining plain content.

It would also be fine to display the changes, I didn't notice them there. Meaning how the new and the old version differ.

B)
(No commentary provided.)

doc. Ing. Š. Starosta, Ph.D.:
A1)
Given Mark: 2
Commentary: Names of some buttons are slightly confusing and do not correspond with what one would intuitively expect. For example, "Upload JSON list" confuses with the word "list". "Deleted" are not all deleted objects, but only the ones that are referenced. In the main window, the data items are not graphically distinguished, so for example "Previous contents" may be "hidden" in the preceding "Contents", that may also have empty lines.

A2)
Given Mark: 1
Commentary: I didn't know, how exactly the tool usage is intended to be, so I wasn't sure about the purpose of some behaviour; however, the whole functionality didn't have any issues. A checkbox is next to every object; I didn't have trouble understanding that it is a control element, but I can also imagine that someone would not notice it.

A3)
Given Mark: 2
Commentary: After the file is uploaded, there is no sign of its version + version of the previous data state which is being compared against.

B)
It would be useful to show data versions (previous and current) and highlight item titles within the object data text field (i.e. graphically separate Type, File, Contents, Previous contents, etc.).

6.4.1 Results

The testers did not seem to have any significant troubles with executing the given test scenario. The user interface can be evaluated as overall intuitive, with the exception of some control elements – like for example "Delete" button, which has a specific functionality based on objects' contents, or checkboxes, which serve solely to mark the objects that have been checked (but not to select objects to perform a display/delete action). Such issues may be eliminated by providing a user guide for the tool, where the purpose of each control element

will be explained.

The identified usability issues are following:

- Version identification of the previous/current JSON input file. This can be fixed by adding the timestamp value from the JSON input file into the UI (in the tested tool version, it is logged into the log file).
- Displayed object's contents graphical presentation. In the current version, it is in plain text. A fix is to add a graphical markup to visually separate data items from each other.
- After the user has confirmed checking, the list refreshes to display all objects, which seems to be an action that the user does not expect. A fix is to display the list with the filter that was set before the Confirm action.

Conclusion

The aim of the developed tool prototype was to provide a solution to improve the evolvability of study materials created at the Department of Applied Mathematics at FIT CTU. The original idea that led to the topic of this thesis was inspired by the Normalized Systems theory and the possibility of its application in the document management domain.

The current state of the art of how the study materials are managed causes inconsistency in different study materials types, which are primarily study texts, presentation slides, online exercises and video recordings of lectures. There are many references between different "objects" (meaning definitions, theorems, exercises, paragraphs of text, figures, etc.) within one type of a study material as well as across all types. For example, a definition presented in a text script may be referred by a theorem in the same script, used by an online exercise and mentioned in a video lecture at the same time. Such references may be taken as *cross-cutting concerns* as they are defined by the NST and they cause so-called *ripple effects*: a change in a definition may cause the necessity of updating theorems, exercises and other entities that refer to this definition.

An obstacle for improving the state of the art and providing a way how to increase the evolvability of study materials was the fact that there are established ways and tools for creating and managing study texts (like WooWoo [9] and LaTeX), and the teachers would most probably not want to learn how to handle another tool, so the possible solution was to provide a means to track the ripple effects instead of avoiding them.

The analysis of the problem and the tool requirements were defined in cooperation with Ing. Tomáš Kalvoda, Ph.D. and doc. Ing. Štěpán Starosta, Ph.D. from the Department of Applied Mathematics. The solution is based on the modular structure of source files from which study texts and presentations are generated: definitions, theorems, tables, text paragraphs, etc. are treated as separate modules, or "objects". The WooWoo program, which is responsible for generating PDF files with text scripts, can track the references between

such objects and generate a list of objects containing their "relationships", i.e. objects that the particular object points to and objects that the particular object is referenced by.

The developed tool (*RefTool*) is a UI desktop application that takes this list of objects as input and provides information about each object and its relationships in a human-readable way. When the user performs changes to an object, the tool detects a change and marks the object as modified. This is to signal the user that he should check the relationships of such an object to verify that the change did not cause any ripple effects, or, if it did, to perform necessary changes on affected objects.

Because the tool is a prototype, the current functionality is limited to the level of text scripts and a single subject. However, the tool was designed in a way that it could be extended to support other formats, such as online exercises and videos, and track references between different study subjects. An essential condition for this is to create a RefTool extension (an application or a framework) that would generate a list of objects similar to the one generated by the WooWoo program for text scripts from web pages with exercises and video playlists.

The RefTool was tested in terms of meeting the defined requirements and usability. No major issues were found, but the presentation of object data requires a better graphical adaptation (currently it is displayed in plain text), which shall be a part of future tool upgrades.

To make a conclusion, the principles of the NST, which lead to evolvability improvement, have not been strictly applied: although the developed tool is working with the modularity principle, it does not enforce encapsulating single modules of text documents and does not solve the problem of ripple effects. Instead, it creates a possibility to track the emerging ripple effects, so that users (authors of study materials) can eliminate inconsistencies in document contents. A possible improvement would be the automation of the process when the user checks modules that have been identified by the tool as affected by a change in another module. This could be done by using Artificial Intelligence or Natural Language Processing algorithms.

Bibliography

[1] Mannaert, H.; Verelst, J.; et al. *Normalized Systems Theory. From Foundations for Evolvable Software Toward a General Theory for Evolvable Design.* Koppa, 2016, ISBN 9789077160091.

[2] Borches, P.; Bonnema, G. On the Origin of Evolvable Systems: Evolvability or Extinction. In *Proceedings of the TMCE*, volume 2, April 2008, [Cited 2021-04-30]. Available from: https://www.researchgate.net/publication/229012738_On_the_Origin_of_Evolvable_Systems_Evolvability_or_Extinction

[3] Rowe, D.; Leaney, J. Evaluating evolvability of computer based systems architectures – an ontological approach. In *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*, March 1997, [Cited 2021-04-30]. Available from: https://ieeexplore.ieee.org/document/581903

[4] Oorts, G. *Design of modular structures for evolvable and versatile document management based on normalized systems theory.* Dissertation thesis, 2019, [Cited 2021-05-04]. Available from: https://repository.uantwerpen.be/docstore/d:irua:1845

[5] Oorts, G.; Mannaert, H.; et al. Exploring Design Aspects of Modular and Evolvable Document Management. April 2017, ISBN 978-3-319-57954-2, pp. 126–140, [Cited 2021-05-04]. Available from: https://www.researchgate.net/publication/316205800_Exploring_Design_Aspects_of_Modular_and_Evolvable_Document_Management

[6] Suchánek, M.; Pergl, R. Evolvable Documents – an Initial Conceptualization. 2018, [Cited 2021-05-04]. Available from: https://www.thinkmind.org/download.php?articleid=patterns_2018_4_10_78002

[7] Knaisl, V. Proposing an Architecture of an Intelligent Evolvable Document Generation System Based on the Normalized Systems Theory. In *Enterprise and Organizational Modeling and Simulation*, 2019, pp. 70–81, [Cited 2021-05-04]. Available from: https://link.springer.com/chapter/10.1007%2F978-3-030-35646-0_6

[8] Kalvoda, T.; Klouda, K. MARAST. 2012, [Cited 2021-04-30]. Available from: https://marast.fit.cvut.cz

[9] Kalvoda, T. WooWoo Specs, May 2021, [Cited 2021-05-04]. Available from: https://kam.fit.cvut.cz/deploy/woowoo-specs/woowoo-specs.pdf

[10] ECMA-404. The JSON data interchange syntax. December 2017, [Cited 2021-04-30]. Available from: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

[11] Unified Modeling Language. December 2017, [Cited 2021-04-30]. Available from: https://www.omg.org/spec/UML/2.5.1/PDF

[12] Gradle Build Tool. [Cited 2021-05-04]. Available from: https://gradle.org

[13] Gluon. JavaFX. [Cited 2021-05-04]. Available from: https://gluonhq.com/products/javafx/

[14] Gson. [Cited 2021-05-04]. Available from: https://github.com/google/gson

[15] JUnit. JUnit 4. [Cited 2021-05-04]. Available from: https://junit.org/junit4/

[16] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1990: pp. 1–84, [Cited 2021-05-04]. Available from: https://ieeexplore.ieee.org/document/159342

[17] OpenJDK. JDK 15. [Cited 2021-05-04]. Available from: https://openjdk.java.net/projects/jdk/15/

[18] Azul. Zulu Builds for OpenJDK. [Cited 2021-05-04]. Available from: https://www.azul.com/downloads/zulu-community/?package=jdk

Appendix A

Acronyms

FIT CTU Faculty of Information Technology of Czech Technical University

JDK Java Development Kit

JVM Java Virtual Machine

NST Normalized Systems Theory

OS Operation System

SDK Software Development Kit

UI User Interface

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `distributions` ..... the directory with executables
- `sources` ..... the directory of source codes
  - `src` ..... the directory of the Java source code
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `MasterThesis.pdf` ..... the thesis text in PDF format