

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

GABRIEL BAUER
BACHELOR THESIS

**INTEGRATION OF A
REFACTORING UI IN THE
SWEBLE HUB VISUAL EDITOR**

Submitted on 11 April 2017

Supervisor: Dipl.-Inf. Hannes Dohrn, Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 11 April 2017

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see <https://creativecommons.org/licenses/by/4.0/>

Erlangen, 11 April 2017

Abstract

Wikis are an essential part of modern internet in order to store knowledge. However, to date, they lack functions which assist on restructuring and transforming the contents of a wiki. The task of maintaining the contents and structure is therefore time-consuming and error-prone. This thesis designs and implements a user interface for restructurings in wikis build on the Sweble Hub software, which is a software similar to most wikis. In contrast to classic wikis, it is able to provide assistance on restructurings. With Sweble Hub and the user interface designed in this thesis, the wiki authors' efficiency is greatly improved.

Contents

1	Introduction	1
1.1	Transformations and Refactorings in Wikis	1
1.2	Goal of this Thesis	2
2	Related Work	4
2.1	Literature Review	4
2.2	Eclipse IDE Refactoring UI	5
3	Architecture and Design	8
3.1	Sweble Hub	8
3.1.1	Visual Editor	8
3.1.2	Collaborative Platform	9
3.2	Refactoring Types	11
3.3	Refactoring UI	12
3.3.1	Basic Workflow	12
3.3.2	Additional Functionalities	15
3.4	Backend	16
3.4.1	API Endpoints	17
3.4.2	Refactorings in a Collaborative Environment	18
3.5	Exemplary Implementations	19
4	Implementation	23
4.1	Technologies	23
4.2	Visual-Editor	24
4.2.1	Adding the Refactoring Button	24
4.2.2	Communication with the Refactoring UI	25
4.3	Frontend	26
4.3.1	Structural Overview	26
4.3.2	Refactoring UI Components	26
4.4	Backend	28
5	Discussion of Results	29

6	Future Work	30
7	Conclusions	31
	Appendices	32
Appendix A	API Endpoints	32
Appendix B	Integrating Further Refactorings	35
	References	37

1 Introduction

1.1 Transformations and Refactorings in Wikis

Wikis are a substantial component in the world of modern internet. The first wiki was made public in 1995¹. Since then, wikis have become more and more important with Wikipedia² being currently the fifth most popular website³. Nowadays wikis are also used in business organizations, for example to store documentation (Buffa, 2006). Even though wikis have evolved, they are lacking some crucial functions. Wikis do not provide automated transformations to restructure their contents. The authors still have to maintain the contents and structure manually. A good example of this time-consuming and error-prone task is the renaming of an article. All articles that contain a link to the renamed article have to be modified. The author has to find and modify every single article manually.

For instance, the most comprehensive Wikipedia (Wikipedia in English) currently contains around 5.349.000⁴ articles. As of 2011 the most internally referenced article was about the *geographic coordinate system*⁵. Nearly 660.000 articles contained a link to this article. Given that the article should be renamed, it would not be possible to modify this large number of articles manually.

One major barrier to automated transformations is the content format, in which the wiki articles are stored. Articles are normally written as pure text, which is structured with a markup language. A commonly used markup language is the wikitext, which is e.g. used in Wikipedia. The wikis then render the markup language as HTML, so that the browser is able to display the article. While nowadays some wikis offer a visual editor with WYSIWYG⁶ functionality, the articles are still stored as text. This makes it difficult for computer algorithms

¹<http://wiki.c2.com/?WikiHistory>

²<https://www.wikipedia.org/>

³<http://www.alexa.com/topsites>

⁴https://en.wikipedia.org/wiki/Main_Page

⁵https://en.wikipedia.org/wiki/Wikipedia:Most-referenced_articles

⁶What you see is what you get

to access a wiki's content.

A high-level representation of the contents is needed to make a wiki more machine-accessible. Dohrn and Riehle (2011b) designed the Wiki Object Model (**WOM**), a tree-based and wiki-independent content representation. With the help of the Sweble Wikitext Parser (Dohrn & Riehle, 2011a) the wikitext markup language can be parsed and converted into WOM. The WOM enables computer algorithms to better interact with the contents of a wiki. Therefore, it is possible to provide automated transformations in wikis (Dohrn & Riehle, 2013). In the previous example, a computer tool could apply the modifications to the articles that need to be changed. The author only has to specify the new article name and after the tool has performed all changes, review those changes. By this, the author's efficiency would greatly improve.

The term **refactoring** is mainly known from software development. It is used to describe a process of restructuring the source code of a software without changing its external functionality. A refactoring is often applied in order to enhance the maintainability of a software. It therefore improves the readability and decreases the complexity of the code. The term has also found its way in the context of wikis and refers to the transformation or restructuring in wikis.

In software development, it is possible to search for the program parts that should be refactored automatically. To put it simple, a refactoring tool is able to compute a mathematical measurement of how well the source code is structured (van Emden & Moonen, 2002). For example, refactoring tools can search for duplicate lines in the source code and suggest to combine both parts of the program. In wikis only a human is able to decide whether an article has to be refactored, since it is difficult to provide an automatic analysis. While source code can be described mathematically, this is complex for literature. An algorithm cannot decide yet if e.g. an article deals with two topics and should be split. Since this is an unsolved problem I will focus on an approach in which the author has to determine where a refactoring has to be applied. The refactoring tool is therefore only responsible for executing a given refactoring instead of detecting the need for a refactoring and then executing it.

1.2 Goal of this Thesis

This thesis is giving a review on literature focusing on refactoring tool support and human computer interactions as well as introduce an existing tool, used in software development. The main goal of this work is on designing a workflow for refactorings in wikis and the corresponding refactoring user interface (**UI**). The UI should guide an author through the steps of specifying a refactoring and

applying it. The UI has to be easy and comfortable to use for the authors. Three different examples will be implemented to demonstrate the possible workflows. The implementation of the refactoring UI has to be extendable with further refactorings.

The following chapters are structured as follows. Chapter 2 provides a short literature review on the topic of improving the usability of refactoring tools as well as a breakdown of an existing refactoring tool for software development. Chapter 3 briefly introduces the Sweble Hub software, one of the main focusses of this thesis, and the conceptual model of the refactoring UI. I will present the different types of refactorings, the workflow and functionalities of the refactoring UI and backend, which provides the data for the refactoring UI. Chapter 4 describes the implementation of the UI and its challenges. The results and the future work are presented in chapter 5 and 6. Chapter 7 gives the conclusions.

2 Related Work

2.1 Literature Review

While the process of restructuring source code exists since the development of programming languages, the concept of refactorings was introduced by Opdyke (1992) for modern programming languages. Since then, refactorings have been studied and further developed (Murphy-Hill, Parnin & Black, 2012). Nowadays, most development environments contain a refactoring tool, even though the range of supported refactorings can vary widely.

One of the first refactoring tools, described in a research paper, was created for the programming language Smalltalk (Roberts, Brant & Johnson, 1997). The authors documented that small refactorings can be easily implemented, while more complex refactorings can be composed of smaller ones. In their article, the authors described the development of a refactoring browser, which can guide the author through the process of extracting parts of a program that can be improved. Further, the authors had the following requirements, while implementing this tool. Refactorings should be fast - faster than doing it manually. And secondly, the refactoring UI should integrate well into the standard development tools.

A decade later, Mealy, Carrington, Strooper and Wyeth (2007) worked out some major points in order to improve the usability of refactoring tools. Since this publication deals exclusively with software tools, some approaches cannot be applied to wiki refactoring tools. The authors assumed that the users are computer experts. However, in case of Wikipedia, most articles are written by the community, which is represented by normal computer users rather than developers. A survey on Wikipedia from 2011 documented that the average author is probably "computer savvy but not necessarily a programmer"¹. Furthermore, Mealy et al. (2007) described that a refactoring can be broken down into three main steps. The first step deals with automated detection of program parts that need to be refactored, while the second step proposes the refactoring to the user. However, as I described in the introduction (see 1.1), the detection of potential refactorings

¹https://meta.wikimedia.org/wiki/Editor_Survey_2011/Executive_Summary#PROFILE

is difficult in wikis. The third step applies the refactoring, which is as well a main focus of this thesis. Most important, Mealy et al. (2007) provided a good summary of guidelines for refactoring tools. Among other suggestions, the authors considered it important to circumvent any errors. If users input data, the system should immediately check if the provided data is valid, instead of checking the data later and then returning an error. Another suggestion is to design the interface "minimal, simple to understand, organised, without redundancy [...] and aesthetically pleasing" (Mealy et al., 2007). Furthermore, it is important, that authors always get feedback about what is happening and that the UI responds fast and meaningful. The authors should be able to check the outcome of a refactoring before applying it to a wiki.

A more efficient way of refactoring source code is described by Lee, Chen and Johnson (2013). The authors showed that the most commonly used refactorings are moving or extracting program parts. Instead of clicking through the refactoring dialogs, the developer could easily drag & drop the source code. While every standard text editor supports drag & drop, which is essentially copy & paste, the authors meant that the moved program code is appropriately modified. For example, in case the developer moves a string concatenation outside its method, a new function should be created. Repositioning elements inside a wiki's article could also be a commonly used refactoring. Therefore, drag & drop refactorings would be a convenient way in the context of wikis.

2.2 Eclipse IDE Refactoring UI

Since the Eclipse² IDE³ software includes a well-known refactoring tool, I analysed its structure and UI. The Eclipse IDE is an open source development environment available for different programming languages. It is mostly known for its Java IDE, which is the most commonly used IDE.⁴

An IDE is a software application which provides several tools in order to assist with software development. All tools needed to create software are directly integrated into the IDE. In general, an IDE consists of a source code editor, tools for building the software as well as a debugger. Modern IDEs also provide further tools like version control systems, tools for constructing UIs or a toolset for refactoring the source code. By using an IDE, the efficiency of software developers is

²<https://www.eclipse.org>

³integrated development environment

⁴<https://zeroturnaround.com/rebellabs/ides%2Dvs%2Dbuild%2Dtools%2Dhow%2Declipse%2Dintellij%2Didea%2Dnetbeans%2Dusers%2Dwork%2Dwith%2Dmaven%2Dant%2Dsbt%2Dgradle/>

greatly improved. Developers are able to perform all tasks inside one program and do not have to switch between different UIs.

The Eclipse IDE provides refactorings on code lines as well as on an entire file. In both cases the refactoring is started in the same way. After selecting the element, a file, function, variable, etc., the developer can start a new refactoring via the right click context menu. Depending on the selected element, the context menu shows the available refactorings (see figure 2.1).

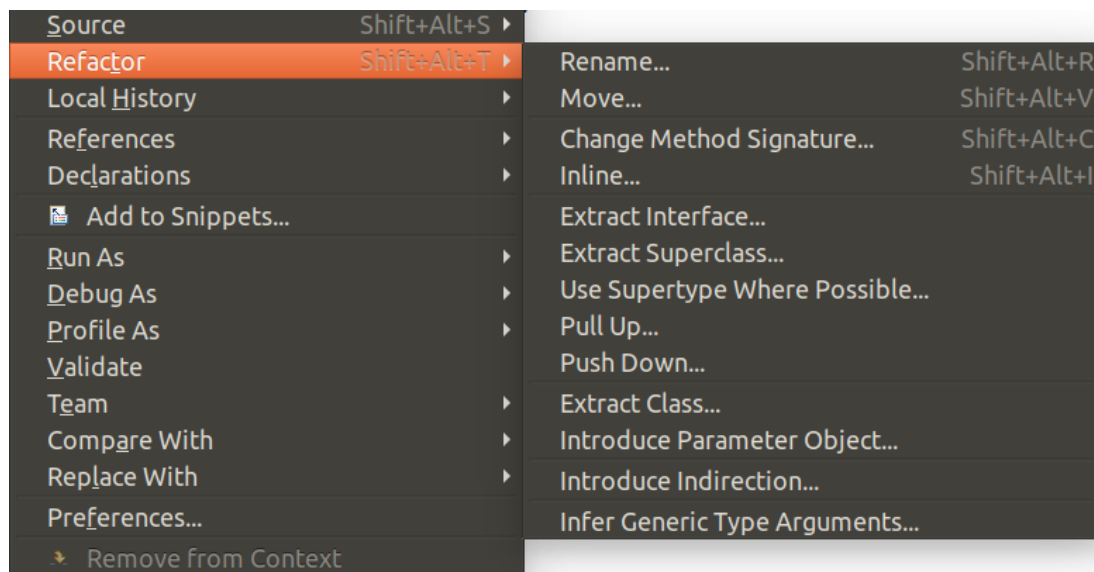


Figure 2.1: Screenshot from the Eclipse Java IDE showing the context menu after selecting a function.

After selecting a refactoring, a new window opens, in which the developer has to specify the parameters. The structure and content of this window varies for different refactorings. For example, in case the developer intends to move a function, the destination has to be specified (see figure 2.2). In order to do so, the developer can use a search box to find the right destination file. After specifying the destination, the author can either preview the changes or continue directly. In both cases the IDE validates if the refactoring is possible. While the validation is running, a progress bar shows its progress. If any problems are found during the check, the UI presents a list with the relevant source code lines (see figure 2.3). The developer can either cancel the refactoring, adjust the parameters or continue and ignore the problems.

As pointed out by Mealy et al. (2007), the system should prevent any errors. In case of the Eclipse IDE, instead of validating the users input immediately, the system only checks the parameters once the developer continues. Even though the validation cannot be computed instantly, it would be more convenient for the

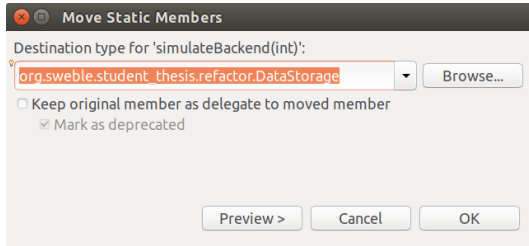


Figure 2.2: Screenshot of the window after the developer has selected to move a function.

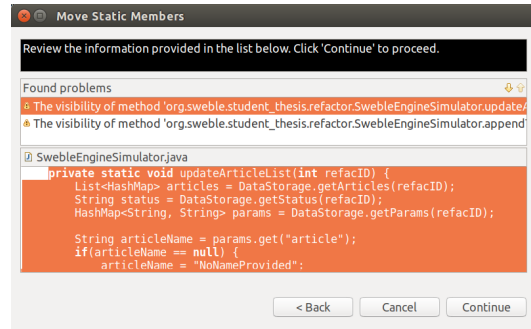


Figure 2.3: Error dialog of a refactor-ing in the Eclipse Java IDE.

user to get feedback directly. A solution to improve this functionality could be to use idle time and to check the input in the background.

A valuable feature that was integrated into the Eclipse IDE refactoring tool is the process of renaming elements. Instead of opening a dialog, the user is able to directly rename the element inside the editor.

3 Architecture and Design

In this section I present the Sweble Hub software, on which this thesis is focussing on, describe the three different categories of refactorings and the basic workflow. Furthermore, I show additional functionalities to the basic workflow and present three exemplary implementations.

3.1 Sweble Hub

The Sweble Hub is a software for managing knowledge. Its basic functionality is comparable to Wikipedia. It improves some main aspects of the underlying technology by using the WOM for storing its contents. Since the WOM enables algorithms to better interact with the contents, the Sweble Hub is able to assist on refactorings in a wiki (Dohrn & Riehle, 2013). With the ability to offer automated refactorings, the Sweble Hub has a main advantage compared to classic wikis. Refactorings that are executed by the software produce less errors and save the author's time.

Two aspects of the Sweble Hub are particularly important for this thesis. First, the visual editor which is the starting point of each refactoring and will be further extended. Second, the approach that is used in order to manage a collaborative usage of the platform.

3.1.1 Visual Editor

The Sweble Hub includes a visual editor (see figure 3.1), which is also used by almost all Wikipedias with the exception of some languages with special characters e.g. Chinese¹. Before the visual editor was introduced to Wikipedia, the editors had to write all articles in the wikitext markup language. The wikitext is a simplified or alternative way of structuring text instead of using pure HTML.

¹<https://www.mediawiki.org/wiki/VisualEditor>

Therefore, authors had to learn the syntax instead of concentrating only on the content of an article. The visual editor, on the other side, provides a user-friendly alternative, whose WYSIWYG functionality is comparable to a rich-text editor like OpenOffice or Microsoft Word. All formattings can be applied visually and are internally represented as wikitext. Furthermore, every element, as e.g. a paragraph or link, can contain a popup, which provides additional editing functionalities or information.

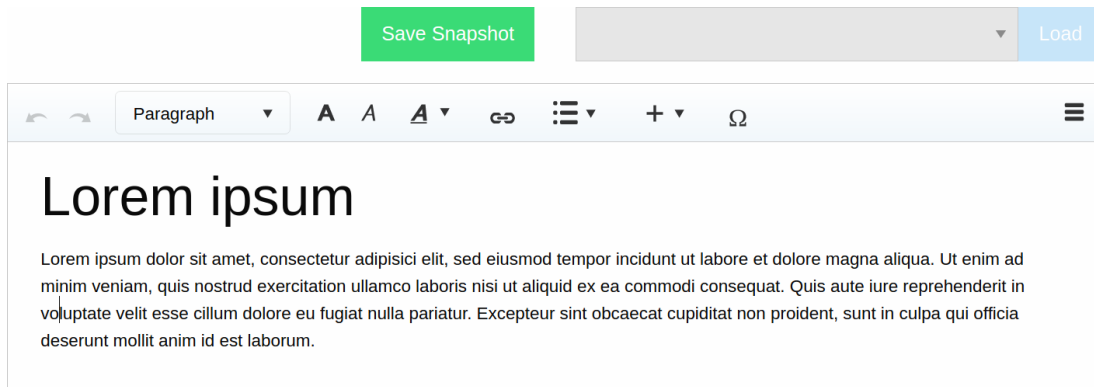


Figure 3.1: Screenshot of the Sweble Hub visual editor, which is used for easy editing of wiki articles.

The visual editor itself was originally developed by the Wikimedia Foundation, the non-profit organization behind Wikipedia. Since the visual editor only exports its contents as wikitext markup language, it had to be extended for the Sweble Hub. Haase (2016) implemented the extension to work with WOM and integrated the visual editor into Sweble Hub. The visual editor now supports the entire WOM specification and can therefore export any formatting into WOM.

3.1.2 Collaborative Platform

In wikis, multiple users work simultaneously on the contents. Whenever two authors work on the same article at once, there is a chance that the changes from one author will accidentally overwrite the changes of another author. Therefore, the wiki software has to make sure that different changes cannot interfere with each other and no modifications are lost.

To overcome this issue, the Sweble Hub uses version control for all stored contents, which allows multiple users to edit, create and delete articles simultaneously. The used type of collaborative groupware is called multi-synchronous (Molli, Skaf-Molli, Oster & Jourdain, 2002). A well-known example for a service, that uses multi-synchronous groupware, is GitHub². It is a web-based version

²<https://github.com/>

control system, which is commonly used for open-source software projects.

On GitHub, each project has its own repository, in which the project files are stored. Changes to a project are always done as **commits**. A commit contains changes to *1 to n* files. Those changes are commonly dealing with the same topic, for example updating the project's description. Every single commit is stored in the repository and can be viewed in a version history. Furthermore, every repository consists of different **branches**, typically with one master branch. Each branch can contain a different set of commits. Branches can be created at any time, even from previous commits. Furthermore, different branches can be combined by merging them. Most merges can be achieved automatically. If both branches contain conflicting changes, the conflicting files have to be merged by a human.

This concept can be applied to wikis. It has some advantages compared to the current concept of most wikis. Changes that were made once, will not get lost, even if they are overwritten by further changes. All changes that were applied to the wiki can be viewed in a version history. Authors may make changes to multiple articles and submit them as one commit, which contains the complete set of changes. Reviewing those changes in context is much easier than reviewing the changes of every file individually.

Furthermore, volunteers can easily contribute to the wiki. On classic wikis, changes by voluntary authors are made directly to the wiki. However, they are not shown until they have been reviewed by the administration. If they get rejected, they are normally deleted after a while. In the multi-synchronous approach, a new branch can be created for each volunteer. The author can then apply the changes to the branch, which is independent from other branches. After the author made his changes, they can be merged back into the master branch. Therefore, changes in multiple branches cannot interfere with each other until they are merged. The administrators can review the changes and merge both branches. Even if the changes are not merged into the master branch, they are still stored in the author's branch. The changes on this branch can be improved and submitted for review again.

Another important advantage is the functionality to revert commits. All changes of one commit can be automatically reverted afterwards. Given that an author changed a large number of articles, the changes can be undone quickly. In contrast, undoing the changes manually and without a version control system is a huge task.

In summary, using a version control system in a wiki system has major advantages. However, the refactoring tool has to collaborate with this approach.

3.2 Refactoring Types

There are several possible refactorings that can be applied to a wiki. With a greater number of changed articles, the time that is needed to process the refactoring will increase. A refactoring is categorized mainly by the number of articles it will change. Each refactoring can be categorized into one of three categories; local, global-limited and global-unlimited refactorings.

Local refactorings

Local refactorings only apply to one article currently opened in the visual editor. An example for a local refactoring is the repositioning of a paragraph inside the same article. The backend does not need much time to prepare and apply this refactoring. Consequently, the changes can be applied immediately after the refactoring has been confirmed. The entire process of a local refactoring can be computed instantly. Normally, the author will not notice any delay.

Global-limited refactorings

If a refactoring changes more than one article, it is called a global refactoring. Furthermore, if the number of changed articles is still small, the refactoring is categorized as global-limited refactoring. This category includes refactorings that commonly change less than ten articles. Most refactorings of this category only change two articles, e.g. the extraction of a paragraph into another article. Extracting a paragraph cuts a paragraph from one article and pastes it into another article. Since this refactoring only affects two articles, the transformations can be applied relatively fast. The general workflow of a global-limited refactoring is the same as a local refactoring since the computation-time is similar.

Global-unlimited refactorings

If a refactoring affects a large number of articles, it is categorized as global-unlimited refactoring. This type of refactoring is the most complex one. An example refactoring that fits this category is the modification of an article name. If the name is changed, all other articles that have a link to the renamed article have to be changed, too. Predicting how the change will affect those articles could take some time, depending on the number of changed articles. As mentioned in section 1.1, in 2011 the internally most referenced article of Wikipedia was linked 660.000 times. The task of modifying this large number of articles needs time, even for a fast computer system.

Therefore, the general workflow of this type of refactoring has to be different. The author should not have to wait until the backend has finished preparing the

refactoring. So, the dialog has to close after all parameters have been provided by the user. The refactoring will then be processed in the background and can be resumed once it has finished its preparations.

3.3 Refactoring UI

The refactoring UI is built on top of the visual editor, which I described in section 3.1.1. The UI guides the author through the process of applying a refactoring in a wiki.

3.3.1 Basic Workflow

The refactoring UI has a predefined workflow, consisting of three major steps. As shown in figure 3.2 each step has interactions with the author. The workflow varies depending of the refactoring type. Nevertheless, they are based upon the same basic workflow.

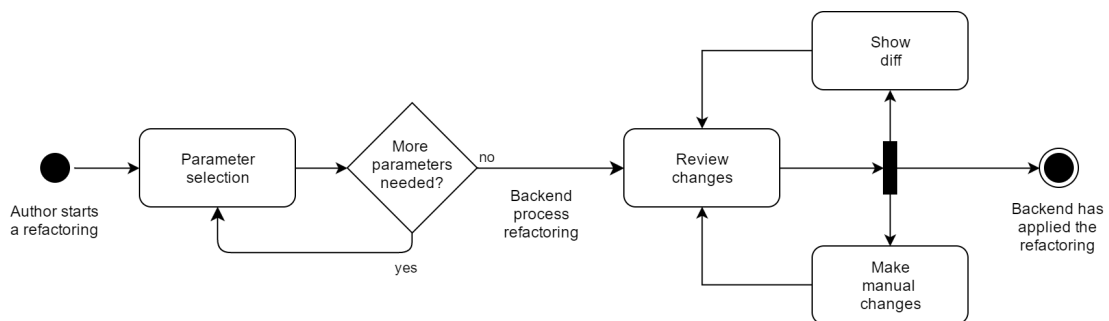


Figure 3.2: The basic workflow of a refactoring represented as an UML activity diagram.

1. Starting a refactoring

First, the author has to select the element which should be refactored. As shown in figure 3.3, the context menu opens, after an element has been chosen. The context menu only contains a refactoring button if there is at least one possible refactoring. After clicking the button, the actual refactoring UI opens in a modal window. The visual editor greys out and stays in the background, while the author navigates through all steps of the refactoring.

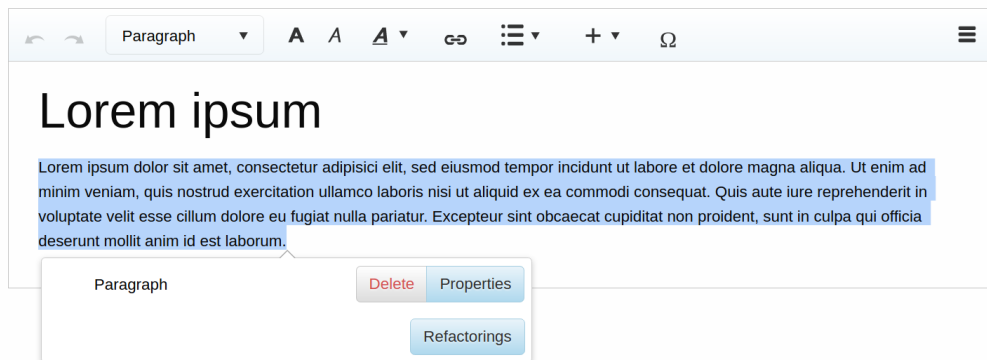


Figure 3.3: Screenshot of the Sweble Hub visual editor with the added refactoring button to the context menu.

Another position for the refactoring button, would have been the top bar inside the editor, which also offers options like e.g. formatting bold, italic or underlined text. While this position is easier and faster to reach for the author, it would have been inconsistent with the structure of the visual editor. The top bar offers formatting options as well as buttons to insert specific elements. All element options and actions are placed within the context menu. Therefore, I placed the button to start a refactoring inside the context menu, too.

2. Selecting the parameters

After the main refactoring UI has opened, the author has to specify the parameters for the refactoring. In order to provide a better structure this step was split into two sub-steps.

(a) Type selection

The author has to specify which refactoring should be applied. Therefore, the user has to select a type of refactoring. The possible types, that are shown in the select box, depend on the element that was selected previously. For example, a link can be renamed or moved, but a paragraph can be moved but not renamed. Thus, the type selection box does not offer the option to rename a paragraph.

This UI was mainly inspired by the Eclipse IDE right-click context menu. Compared to the Eclipse IDE, the refactoring UI was extended with a subtype. The subtype input field will only appear if the type needs clarification. For example, if the author selects the type *move* on a paragraph, the subtype field is displayed. The options for the subtype are to either move the paragraph up, down or to move it into another article. Once the author selected the type and if necessary the subtype, the continue button is enabled.

(b) Additional parameters (*optional*)

Since there are commonly more parameters needed, the author has to provide them in this second sub-step. However, not all refactorings have additional parameters, e.g. moving a paragraph upward inside its article. Therefore, this step is optional, depending on the refactoring. An example for a refactoring needing additional parameters is the extraction of a paragraph into a new article. The backend has to know the name of the new article. It is important to note that all inputs of the author should be validated immediately, as described in section 2. Therefore, the system has to check whether an article with the same name already exists.

3. Reviewing the changes

After the author provided all parameters, the backend computes the results. The UI shows a list of changed articles. The author can exclude or include articles from the refactoring by checking or unchecking a checkbox. Furthermore, for each article the UI offers a diff view to review the changes in detail and a visual editor (see figure 3.4) to adjust articles manually. Currently, the diff view, which is shown in figure 3.5, is only a mock and has to be replaced. Another small function, that was integrated into the review process, is the possibility to display a warning (or error) message. The system can produce a message for each article individually and a warning icon is then displayed behind the article name. Since the Sweble Hub frontend does not include a mouse-hover popup functionality currently, the message is displayed only above the visual editor. After the changes have been reviewed and confirmed by the author, the backend applies the refactoring and the editor reloads with its updated content.

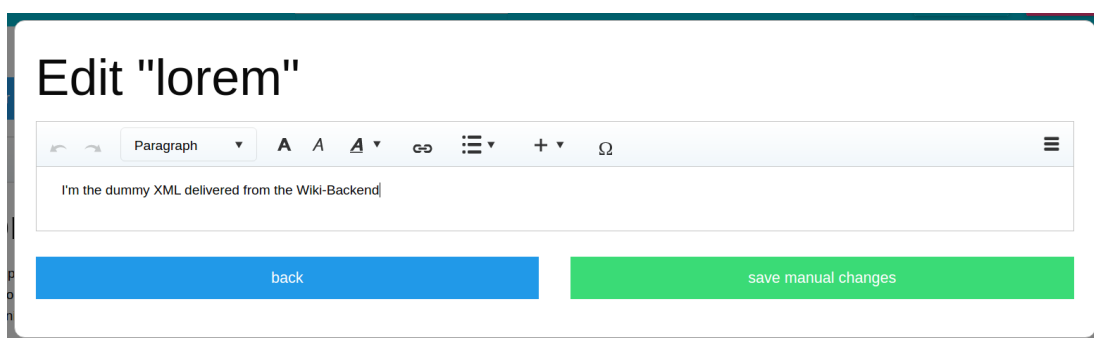


Figure 3.4: Screenshot of the visual editor to perform manual changes on an article within the review process.

While the author is selecting the parameters, he can go back through all steps and adjust parameters. As I will explain later, the refactoring is created on

the backend after all parameters have been selected. Therefore, the parameters cannot be changed anymore. Instead, the author has to cancel the refactoring and start a new refactoring with adjusted parameters.

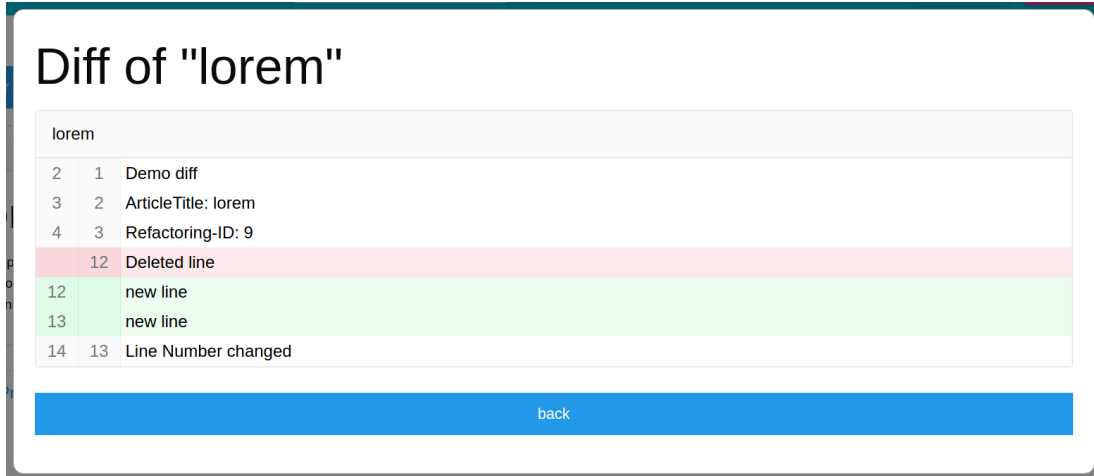


Figure 3.5: Screenshot of the refactoring UI showing the demo diff view of an article.

3.3.2 Additional Functionalities

The basic workflow was further extended with additional functionalities. This is necessary due to the fact that a global-unlimited refactoring has - as pointed out in section 3.2 - extended requirements and to enrich and simplify the refactoring's workflow.

Pausing refactorings

Since global-unlimited refactorings may change many articles, reviewing all changes and if necessary modifying some of them can take time. The UI therefore offers the possibility to save the refactoring for later. This functionality is provided for all types of refactorings. The author has to name the refactoring and can save it anytime during the review process (see figure 3.6). By default, the refactoring is named after its internal ID, for example *refactoring#0*. After the author named the refactoring, the dialog closes and the author returns to the visual editor. In contrast, the selection of parameters does not include a function to save the current state, because it should be possible to provide all parameters in only a few moments.

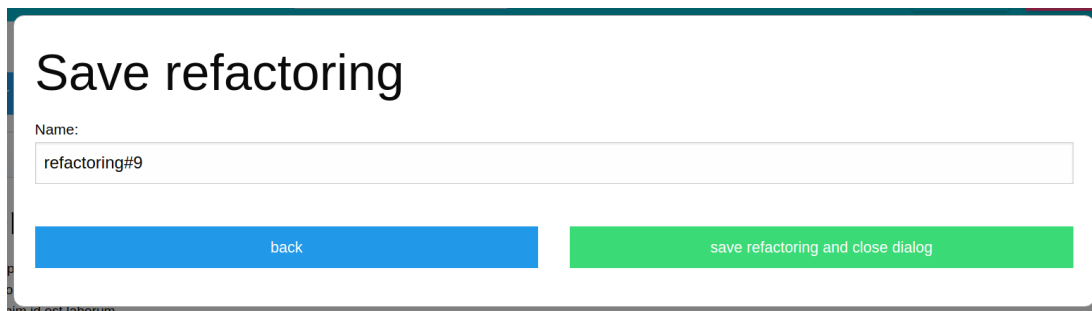


Figure 3.6: Screenshot of the dialog to save refactorings.

Addition for global-unlimited refactorings

In case of global-unlimited refactorings, the backend needs time to perform the refactoring. The processing takes place between step two, selecting the parameters, and step three, reviewing the changes. Therefore, the workflow must be paused after step two, so that the author can continue on other tasks and does not have to wait. The refactoring dialog closes with a message that the refactoring will be performed in the background. Once the refactoring has finished the author can resume and continue with step three, as described in the following paragraph.

List of refactorings

The refactorings from the two above described functionalities are both collected in a list. The author can follow the process of a refactoring, cancel each refactoring or resume saved or finished refactorings in this list. The list is rendered on a independent webpage and not on top of the visual editor. Furthermore, if the author resumes a refactoring, it will not open in a modal windows. Instead, it is also displayed on a independent webpage. After finishing the refactoring, the author is redirected back to the list of refactorings.

3.4 Backend

The refactoring UI uses a client-server architecture, where the backend handles most of the logic. The backend already existed before I started the work on this thesis. It provides demo articles for the visual editor from a local Git repository. I further extended the backend to serve the data for the refactoring UI.

Furthermore, the backend includes the Sweble Wikitext Parser, which converts articles stored as wikitext into WOM. With the Sweble Wikitext Parser all articles from Wikipedia can be opened in the visual editor and be refactored with the

UI, as described in this thesis. However, the backend is currently lacking the implementation that actually performs the refactorings. All provided data for the refactoring UI are for demo purposes only. Nevertheless, I implemented some logic into the backend to show the functionality of the refactoring UI. For example, after a refactoring has been applied, the backend appends a line with the refactoring’s parameters to the article in which the refactoring was started.

An exception to the server-sided logic is the initial parameter selection. Inside the visual editor, each element (e.g. paragraph, link, image, ...) has an option to either enable or disable the refactoring button, because not every element can be refactored currently. The visual editor does not communicate with the backend directly. Instead, the articles are passed by the frontend into the visual editor. Therefore, the starting button of a refactoring is hard-coded, because it would have been rather complex to integrate a server-sided control mechanism into the visual editor. Since the refactoring definitions should be positioned at a consistent place, they were all implemented into the frontend. Therefore, no communication with the backend is needed at this step. As soon as the author provided all parameters, the refactoring is created on the backend and the parameters are transferred afterwards. Any further steps will now be handled by the backend. The client will only display the information in a correct manner.

Furthermore, the backend provides different status codes to the frontend, as shown in table 3.1. The frontend then shows the right UI depending on the status code. Since the parameter selection is not managed by the backend, the frontend begins fetching the status from the backend afterwards.

Status	Client content	Backend
started	parameter selection	client-side only
progressing	closes dialog	prepares the refactoring
awaiting-confirmation	review changes	-
committed	closes dialog	finished the refactoring

Table 3.1: All status codes and their frontend behaviour.

3.4.1 API Endpoints

The frontend communicates with the backend over an application programming interface (**API**). The frontend sends HTTP requests to specific API endpoints. Any endpoint has its own function and returns different data. Furthermore, the frontend sends its request with a HTTP method, e.g. GET, PUT, POST, etc.. The sent HTTP method indicates the type of request. Three HTTP methods were used in the refactoring’s API. If the frontend needs data, it sends a GET

request. If the frontend sends data to the backend, it will use a PUT request. The third used HTTP method POST is used for creating a new refactoring.

Path	Method	Description
/refactorings/create	POST	Creates a new refactoring
/refactorings/list	GET	List all refactorings
/refactorings/{ID}/status	GET	Returns the status
/refactorings/{ID}/status	PUT	Set the status
/refactorings/{ID}/parameters	PUT	Set the parameters
.../articles	GET	Returns all changed articles
.../articles/{article}	GET	Returns the XML of an article
.../articles/{article}	PUT	Set the content of an article
.../articles/{article}/changes	GET	Return diff

Table 3.2: Overview of all API endpoints implemented for the refactoring UI. The second part's path is /refactorings/{ID}/review/...

I had to choose different endpoints in order to design a clear and simple API. All implemented endpoints are shown in table 3.2. The first five endpoints are used in general for starting and managing a refactoring. The second part of the endpoints was necessary to provide all the functionalities for the review process. A detailed, technical description is listed in appendix A.

3.4.2 Refactorings in a Collaborative Environment

As described above, the Sweble Hub uses a version control system for its contents. The refactoring backend has to cooperate with the version control system. If an author starts refactoring, other changes may interfere with the refactoring. This is especially true for global-unlimited refactorings, which take time in order to prepare and process. If another conflicting commit is applied before a refactoring has finished, the refactoring has to be restarted or the wiki is in an inconsistent state.

One way of dealing with such problems could be a directly integrated UI for repeating the refactoring inside the refactoring UI. While this appeared to be a good and easy solution, it turned out that this could lead to a nearly infinite loop of restarting refactorings. Since the backend has to prepare the refactoring, some time may elapse between starting and finishing the refactoring. Therefore, other commits can interfere with the refactoring again.

Another way of eliminating this problem is to create a new, temporary branch, where the state of the wiki will not change. Each time a refactoring is started the backend creates a new branch, on which the refactoring is applied. Therefore, the

process of applying a refactoring stays simple, with no possibility for conflicts to arise. All changes of a refactoring will be summarized as one commit automatically. However, the changes from the automated refactoring and the optional changes from the author are split into two separate commits. Thus, the manual changes can be undone easily.

Nevertheless, the branch has to be merged back into the original branch, which may not be the master branch, but a branch from an author. While most smaller refactorings can probably be merged automatically, bigger ones are more likely to produce conflicts. If it is not possible to merge both branches automatically, the user has to merge it manually. Since a UI for resolving conflicts manually was beyond the scope of this thesis, this workaround was not further elaborated.

However, a general problem is not solved with this approach. If an author inserts new content to the wiki while a refactoring is running, this may lead to an inconsistent state. Since the refactoring only applies to the commit at the time it was started, any further changes are not considered by the refactoring. Given that a refactoring, that renames an article, is running at the same time and an author inserts a link to this article, the new inserted link will not be changed. After the refactoring has finished and was merged, the wiki will contain an invalid link to the renamed article. Therefore, further work is needed on how to deal with inconsistencies from refactorings.

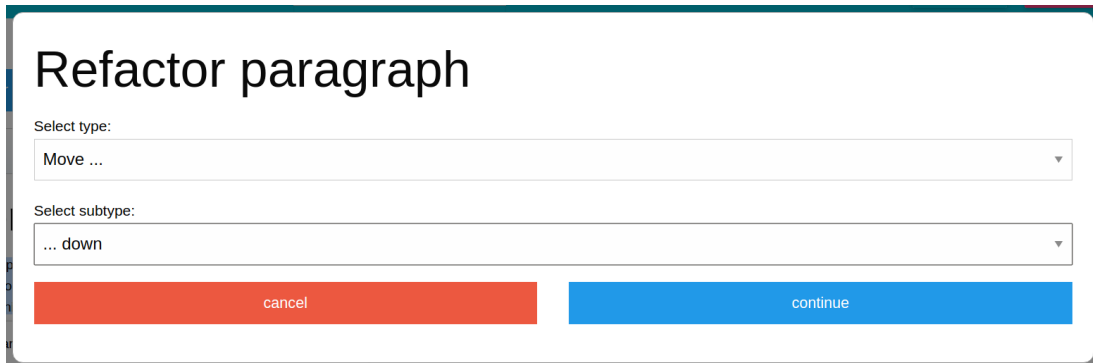
3.5 Exemplary Implementations

In the following, I present one example for each of the three categories to illustrate the different workflows.

Local refactoring

An example for a local refactoring is the repositioning of a paragraph inside its article by e.g. moving a paragraph upwards or downwards. The refactoring is started from within the visual editor through the refactorings button inside the element's context menu, as previously shown in figure 3.3. Afterwards, the refactoring dialog opens and shows the parameters selection. Since this refactoring does not have any additional parameters, the author only selects the type 'Move ...' and subtype e.g. '... down' (see figure 3.7). Directly afterwards, the UI shows the list of changed articles, which looks similar to figure 3.9. In this case, only one article was changed. After the author accepted the changes, the visual editor will reload with the updated content. Since the backend is missing the actual refactoring implementation, only a line with the refactoring's parameters is inserted into the article. In this example the appended line shows " *Refactored with*

params: {article=lorem, type=move, subtype=down, nodeType=paragraph}”.



Refactor paragraph

Select type:
Move ...

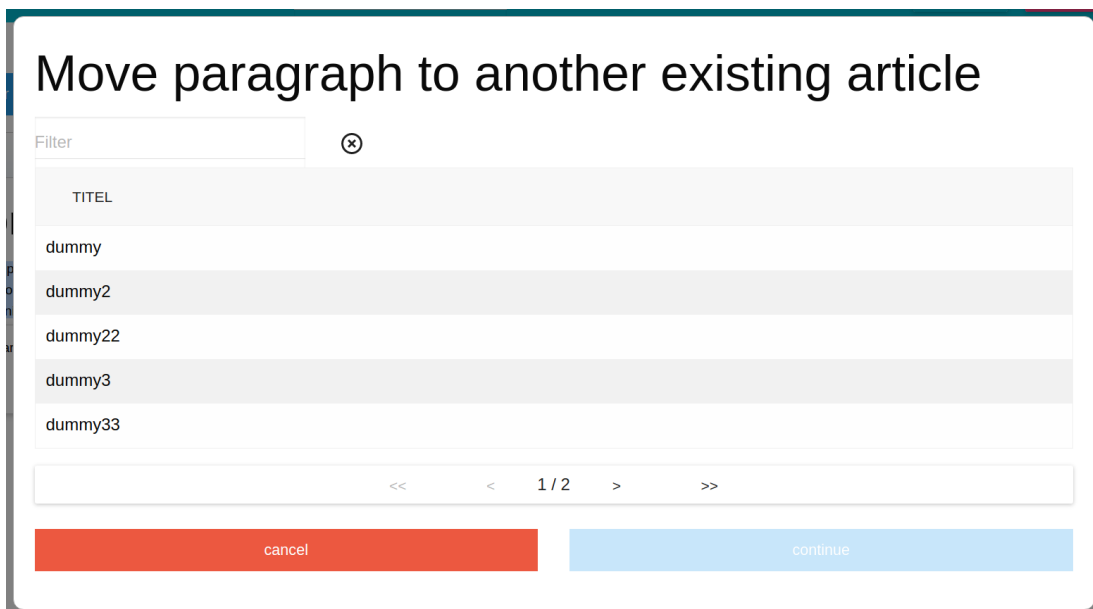
Select subtype:
... down

cancel continue

Figure 3.7: Screenshot of the parameter selection after selecting a paragraph.

Global-limited refactoring

In general, the workflow of a global-limited refactoring is similar to a local refactoring. As an example for a global-limited refactoring I chose to implement the extraction of an paragraph into another (existing) article. The refactoring is started inside the visual editor and afterwards, the type *'Move ...'* and subtype *'... into another article'* have to be selected. In contrast to the first example, the system needs to know in which article the author wants to move the paragraph



Move paragraph to another existing article

Filter ⊗

TITEL
dummy
dummy2
dummy22
dummy3
dummy33

<< < 1 / 2 > >>

cancel continue

Figure 3.8: The second step of the parameter selection, when extracting a paragraph into another article.

(see figure 3.8). Therefore, the UI shows the second step of the parameter selection. In this case, an article selection is shown. After the author has selected the target article, the review dialog is displayed. As shown in figure 3.9, both articles, that will be modified, are listed in the review dialog. As in the above example, the visual editor reloads after the confirmation of the changes.

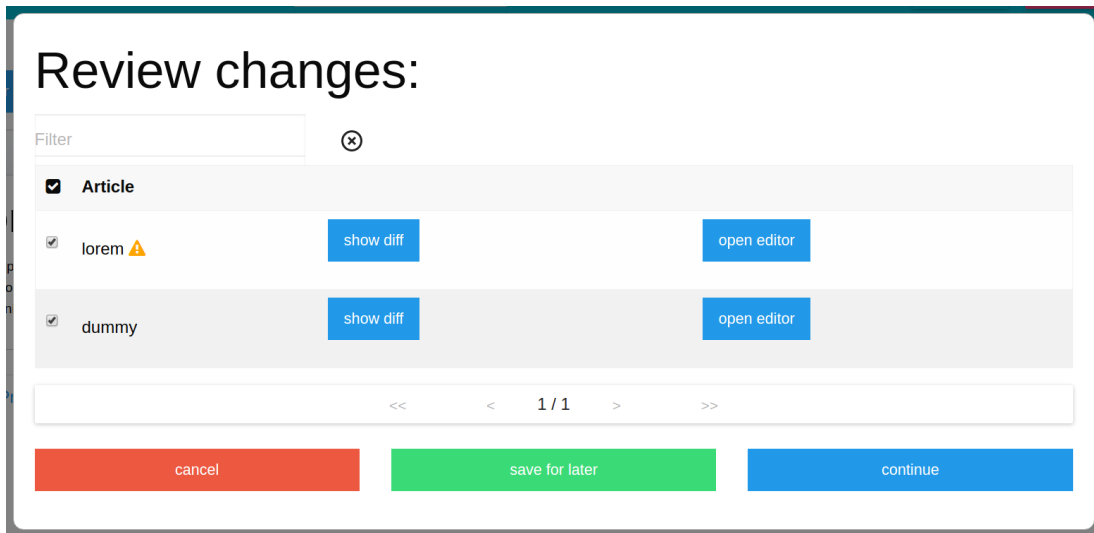


Figure 3.9: Screenshot of the review dialog of a global-limited refactoring.

Global-unlimited refactoring

Renaming an article represents an example for a global-unlimited refactoring. All articles that contain a link to the renamed article have to be modified. The workflow differs from the two above examples. The refactoring is started via the button above the visual editor. After the author has selected the parameter to rename the article, the UI shows an input field to provide the new name. The new name is validated immediately. Following, the dialog closes and the visual editor becomes visible again.

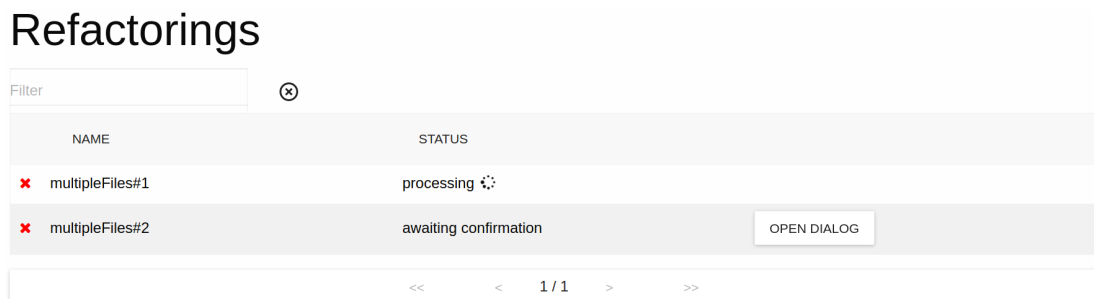


Figure 3.10: Screenshot of the list of running refactorings.

The progress of the refactoring is then shown in the list of running refactorings (see figure 3.10). If the backend finished preparing the refactoring, the author can resume with step three. The dialog is displayed on its own webpage. After the author has confirmed the changes, the refactoring list is shown again and the refactoring is listed as committed.

4 Implementation

This chapter gives a brief overview of how I implemented the refactoring UI into the Sweble Hub. This work is based on the state of the frontend after the visual editor has been integrated by Haase (2016).

4.1 Technologies

The Sweble Hub uses several technologies and programming languages. The top most layer of the software stack, the frontend is programmed in React, a JavaScript library. In React, all frontend elements are programmed as components. A complex UI is created by combining multiple smaller components into one bigger component, with each component having its own state and logic. In order to create a complex frontend, React is combined with Redux, which is also a JavaScript library. Redux provides a global storage, e.g. for the application state.

To achieve a global look & feel, the frontend framework Foundation is used. Thus, the styling of all HTML elements can be changed easily and the framework is mobile friendly. All refactoring UIs are developed with the help of Foundation elements. The refactoring UI integrates well into other UI elements, which also use Foundation. Furthermore, I used icons from Font Awesome, a CSS-based icon toolkit.

The visual editor is programmed mainly with the help of the JavaScript library jQuery. The Sweble Hub visual editor consists of the original visual editor from Wikimedia and the *WOM3* extension to collaborate with the WOM format. All code changes are done exclusively to the *WOM3*-module. To structure the visual editor, it uses a namespace inside the global domain. While both parts share the same namespace, classes can be distinguished easily. *WOM3* files begin with 'Wom3', e.g. *ve.ui.Wom3LinkContextItem*.

Further, the backend is programmed exclusively in Java. In order to offer the REST-API, the framework JAX-RS is used. The entire backend is build with the

automated build system Marvin.

4.2 Visual-Editor

I mainly implemented the UI as React components rather than as part of the visual editor. Therefore, the look & feel is the same as the general frontend and it is easier to maintain. However, the initial step - starting a refactoring - had to be implemented directly into the visual editor.

4.2.1 Adding the Refactoring Button

As I described in section 3.3.1, the context menu of an element may contain a refactoring button. Whether a refactoring button is shown depends on if refactorings are defined for the element. Currently, not all elements have refactorings defined. Therefore, each type of element has the hard coded option to either enable or disable the refactoring button.

For each element the visual editor offers a *contextItem*, which I referred to as context menu previously. The *contextItem* is located in the *ve.ui* namespace and defines the popup appearing after an element has been selected. Any *contextItem* inherits the *ve.ui.LinearContextItem* class. I implemented an extension for this class called *Wom3LinearContextItem*. It inherits the *LinearContextItem* and is responsible for adding the refactoring button. The *Wom3LinearContextItem* offers a simple variable, *hasRefactorings*, to its child to either enable or disable the button.

In order to add a refactoring button to a *contextItem*, e.g. *ve.ui.Wom3LinkContextItem* for links, it has to inherit the new class *ve.ui.Wom3LinearContextItem*. After completion, the refactoring button can be enabled via the *hasRefactorings* variable, as shown in listing 4.1.

```
1 00.inheritClass(ve.ui.Wom3LinkContextItem, ve.ui.  
    Wom3LinearContextItem);  
2 // ...  
3 ve.ui.Wom3LinkContextItem.static.hasRefactorings = true;
```

Listing 4.1: Lines that need to be changed or inserted in order to enable the refactoring button on links.

4.2.2 Communication with the Refactoring UI

After the user clicks the refactoring button, the visual editor has to send an event to the React component. Even though the visual editor as well as the frontend use a JavaScript library, it is difficult to communicate directly, since both elements use different approaches in their dataflows. The visual editor uses a namespace inside the global domain. Different classes can communicate easily. The react frontend has a hierarchical structure of its components similar to object-oriented programming. Only with the addition of Redux, communication between hierarchical not directly related components becomes possible.

The click event is triggered in the *ve.ui.Wom3LinkContextItem* class and has to be passed to the refactor component. There are two options in order to connect both lower-level elements. One possibility is to pass the event through different classes, which requires to modify many files. The other option is to communicate through the global namespace, which is not generally recommended in React. Since there will be only one refactor component, I decided to implement it this way. The connection is defined inside the refactor component. As shown in listing 4.2, I defined a new sub-namespace *ve.globalWom3* and bound a local React function to *ve.globalWom3.refactorHandler*. In order to suppress warnings from ESLint, which is used to check if the code meets the style guidelines, it is disabled and re-enabled afterwards.

```
1 /* eslint-disable */
2 ve.globalWom3 = {
3     //this.startRefactor is a method inside the react component
4     refactorHandler: this.startRefactor.bind(this)
5 };
6 /* eslint-enable */
```

Listing 4.2: Code that binds a component's function to the visual editors namespace.

Inside the visual editor, the click event can simply be passed to the function defined in *ve.globalWom3.refactorHandler*, as shown in listing 4.3. Since the backend currently does not contain the actual refactor implementation, only the name of the element type is transferred as parameter. The parameters have to be extended with an element identifier so that the backend knows which element should be refactored.

```
1 ve.globalWom3.refactorHandler(this.constructor.static.name);
```

Listing 4.3: The method-call within the click event handler of the *ve.ui.Wom3LinearContextItem* which is used to pass the event to the refactoring frontend.

4.3 Frontend

All further refactoring UI elements were implemented using React and Redux.

In general, all components with lists (article selection, refactoring list and the list in the review component) are equipped with pagination, filter and sort mechanisms. Each component contains a loading animation (*LoadingDataDecorator*), which is provided by the Sweble Hub frontend.

4.3.1 Structural Overview

I split the refactoring UI into multiple components for a better maintainability. As shown in listing 4.4, there are three smaller components (*refactorings/components/**), two main components (*refactorings/**) as well as a demo implementation (*refactorings/demo*). The three smaller components are independent on their own and can therefore be better tested and reused elsewhere.

```
refactorings
  components
    diff
    parameters
      components
    review
  demo
  dialog
  list
```

Listing 4.4: Directory structure of the React components, which are located at `frontend/app/client/src/`.

4.3.2 Refactoring UI Components

Parameter selection `./components/parameters`

The parameters component handles the gathering of the parameters. Inside the directory is a *settings.js* file, which contains the basic configuration for each type of element, e.g. paragraph or wom3Link for links. The type (and subtype) selection is specified in this JSON config. New refactorings can therefore be added easily. Further steps of the parameter selection have to be specified manually inside the main *Parameters.js* component. Sub-components for further steps should be placed inside the *parameters/components* directory. A documentation how to integrate further refactorings can be found in appendix B.

Review list `./components/review`

After the author specified all parameters and the backend has finished preparing the refactoring, the dialog shows a list of changes. This component also implements the function to save the refactoring, displays the diff view for a specific article, in- or excludes an article from the refactoring as well as opens the visual editor for manual changes.

Diff view `./components/diff`

This component contains a demo implementation of the diff view. It fetches the data from the backend and displays it. It has to be replaced with the real implementation of the diff view.

Main dialog `./dialog`

The dialog component is the main component in the refactoring UI. I implemented all the general logic regarding the refactoring dialog into this component. It connects the parameter and review components and is responsible for fetching the status code from the backend as well as determining which component should be displayed. The component is used for starting and for resuming a refactoring. Therefore, this component includes the *ve.globalWom3.refactorHandler* function, from which the visual editor sends the refactoring start event. Furthermore, the component includes the logic to decide whether to display the dialog in a modal windows. In summary, this component is mainly used as content-router and in order to glue all subcomponents together.

List of running refactorings `./list`

When refactorings are manually saved or prepared in the background, they are listed with the help of this component. It fetches all running refactorings and displays them.

Demo implementation `./demo`

This directory contains two small components, which combine the visual editor with the refactoring dialog as well as the refactoring list with the refactoring dialog.

API Client

Most of the components have to communicate with the backend. I created a singleton class, which provides all methods for the communication. It is located at *frontend/app/client/src/api-clients* and uses the JavaScript library *SuperAgent*

for all ajax requests.

4.4 Backend

While the backend consists of the fully implemented API, it only serves demo data right now. To demonstrate all frontend functionalities some logic had to be implemented into the backend. I implemented logic for pagination, filtering and sorting as well as to change the status code of a refactoring after a user input. However, since the functional implementation will be replaced, the logic was only provisionally programmed. Only the API definitions will be adopted.

In the JAX-RS framework, all API endpoints are represented as independent class. To separate the new refactoring endpoints from existing endpoints for the visual editor, I created a new package named *org.sweble.student_thesis.refactor*. Only the *EditorBackendApplication* class, which is located in the *org.sweble.student_thesis.v3* package and initialises the API, was extended to integrate the new endpoints. To store the refactorings data in the backend, I used multiple *HashMaps*. Therefore, the refactoring data are not stored persistently and are reset once the backend is restarted.

The endpoint for creating a refactoring returns the ID of the new refactoring as part of the location HTTP header. However, browsers remove the location header from the HTTP response¹. In order for the frontend to get the location header, the backend has to instruct the browsers to expose this header. As shown in listing 4.5, the backend therefore needs to send the additional HTTP Header *Access-Control-Expose-Headers: Location*.

```
1 UriBuilder builder = uriInfo.getBaseUriBuilder();
2 builder.path("refactoring/" + Integer.toString(refacID));
3 return Response.created(builder.build()).header("Access-Control-Expose-Headers", "Location").build();
```

Listing 4.5: Excerpt of the Java code, which responds the refactoring ID as location header and exposes this header to the JavaScript code.

¹<https://github.com/visionmedia/superagent/issues/770>

5 Discussion of Results

The goal of this thesis was to design a refactoring workflow with working implementations that is easy to extend and integrates well. New refactorings can be added fast, since the first step of selecting the type and optional subtype is defined in a JSON configuration file. Even further steps in the parameter selection can be added with minor changes (see appendix B). Furthermore, one example per refactoring type was implemented to illustrate the refactoring UI.

As presented in section 2, drag & drop could replace some commonly used refactorings. Repositioning a paragraph inside an article could be a common refactoring performed in a wiki. The feature of drag & drop could be easy and intuitive for authors. However, since drag & drop is handled by browsers and not by a script on the webpage, it is probably rather complex to achieve such a functionality for web components. Therefore, it was not further considered in this thesis.

A suggestion for the design of refactoring tools was to provide fast refactorings (Mealy et al., 2007). The simplest refactoring, moving a paragraph inside its article, takes more time than copying and pasting the paragraph manually. Therefore, this particular refactoring is rather a proof of concept. Another suggestion was to offer shortcuts for power-users (Mealy et al., 2007). Currently, the refactoring does not offer shortcuts. A possible shortcut could be a checkbox in the parameter selection to confirm the changes automatically and to skip the review dialog.

Furthermore, the author should be aware of the current position in the refactoring dialog. Since the workflow always consists of three major steps (starting the refactoring, providing the parameters and reviewing the changes), it is easy to follow the progress. Another suggestion from the literature is to prevent errors (Mealy et al., 2007). The refactoring UI checks the user input immediately and therefore prohibits errors.

In summary, the requirements for designing and implementing the refactoring UI from section 1.2 were achieved. Further, most of the suggestions from section 2 were implemented into the refactoring UI.

6 Future Work

Future work is necessary to implement more refactorings as well as integrate the actual refactoring algorithm. With both additions authors could use the refactoring UI in a productive environment. Small improvements could be implemented to the refactoring UI, for example shortcuts for power-users. Furthermore, the type and optional subtype of a refactoring should be validated. Currently, the last paragraph of an article shows the option to move it downwards. Thus, for the last paragraph of an article, the subtype down and for the first paragraph the subtype up should be hidden.

Since each refactoring creates a new temporary branch, both branches, the temporary and the original branch, have to be merged afterwards. A 'merge UI' is missing to resolve conflicts, which may arise if both branches contain changes to the same articles. As workaround, branches can be merged with existing software (e.g. KDiff3¹) or manually with an editor and a terminal. However, as described previously, after merging the branches inconsistencies can occur, such as e.g. dead links. Therefore, future work is needed on the merge UI as well as to validate options to prevent such inconsistencies.

¹<http://kdiff3.sourceforge.net/>

7 Conclusions

In this thesis, I have summarized UI-goals for refactoring tools, designed a workflow for refactorings in wikis and implemented it on top of the Sweble Hub software. The UI guides the authors through the process of applying a refactoring in a wiki. The dialog consists of three steps, in which the authors start the refactoring, provide the necessary parameters and review the changes. The refactoring UI is extendable, can be integrated easily and is able to handle all sorts of refactoring types. Furthermore, it follows the summarized suggestions, for e.g. to prevent errors by immediately checking the user's inputs.

Currently, wiki authors have to manually restructure and transform wikis, since the used software can only offer limited assistance in this process. Refactorings are therefore time-consuming and error-prone. With the framework of the Sweble Hub software, it is possible to provide assisted refactorings. Especially in bigger wikis, the author's efficiency is increased. With the refactoring UI designed in this thesis, everybody is able to start and perform refactorings in a wiki.

Appendix A API Endpoints

POST /refactorings/create

Request query: which=\${node-type}

Response code: 201 CREATED

GET /refactorings/list

Request query: length=\${paginationMaxLength}&offset=\${paginationOffset}

Request query (optional): filter=\${filter}&sort=\${sortByRow}[:desc]&direction=[next, prev, last]

Response code: 200 OK

Response type: JSON

Example response:

```
1 {
2   "itemsTotal":2,
3   "paginationOffset":0,
4   "items":[
5     {
6       "name":"multipleFiles#1",
7       "refacId":1,
8       "id":0,
9       "status":"processing"
10    },
11    {
12      "name":"multipleFiles#2",
13      "refacId":2,
14      "id":1,
15      "status":"awaiting-confirmation"
16    }
17  ]
18 }
```

GET /refactorings/\${ID}/status

Response code: 200 OK

Response type: JSON

Possible responses:

```
1 //One of the following status codes:
2 [awaiting-confirmation, processing, committed]
```

PUT /refactorings/{ID}/status

Request query: ?which=[cancel, apply-refactoring]

Response code: 200 OK

PUT /refactorings/{ID}/parameters

Request body: JSON

Response code: 200 OK

Example request:

```
1 {
2   "type": "move",
3   "subtype": "up",
4   "article": "lorem"
5   // ...
6 }
```

GET /refactorings/{ID}/review/articles

Request query: length=\${paginationMaxLength}&offset=\${paginationOffset}

Request query (optional): filter=\${filter}&sort=\${sortByRow}[:desc]&direction=[next, prev, last]

Response code: 200 OK

Response type: JSON

Example response:

```
1 {
2   "itemsTotal": 1,
3   "paginationOffset": 0,
4   "items": [
5     {
6       "name": "lorem",
7       "include": "true",
8       "warning": "This is a demo warning that may appear when
9         the server is unsure with a decision.",
10      "id": 0
11    }
12  ]
}
```

GET /refactorings/{ID}/review/articles/{articleName}

Response code: 200 OK

Response type: XML

PUT /refactorings/{ID}/review/articles/{articleName}

Request body: XML

Response code: 200 OK

GET /refactorings/{ID}/review/articles/{articleName}/changes

Response code: 200 OK

Response type: JSON (Diff view)

PUT /refactorings/{ID}/review/articles/{articleName}/status

Request body: JSON

Response code: 200 OK

Possible requests:

```
1 //One of the following:  
2 [exclude, include]
```

Appendix B Integrating Further Refactorings

This appendix gives a brief overview of the necessary steps to add new refactorings to the frontend.

Enabling the refactoring button

The steps are presented in section 4.2.1 and in listing 4.1.

Defining the type (and subtype)

The refactoring type and its optional subtypes have to be defined in the JSON configuration file, which is located at `/refactorings/parameters/settings.json`. The current configuration is shown in listing 7.1.

```
1 {
2   paragraph: { //Refactorings on paragraphs:
3     move: { //Type 'move'
4       label: 'Move ...',
5       options: { //Subtypes
6         up: {
7           label: '... up',
8         },
9         down: {
10          label: '... down',
11        },
12        article: {
13          label: '... to another existing article',
14          steps: 2, //One additional step
15        }
16      }
17    },
18  },
19  wom3Link: {
20    rename: {
21      label: 'Rename linked article',
22    }
23  },
24  article: {
25    rename: {
26      label: 'Rename article',
27    }
28  }
29 }
```

Listing 7.1: JSON configuration for the types and subtypes of a refactoring.

The JSON configuration contains all elements (paragraph, wom3Link, etc.), which can currently be refactored. The exact naming of an element type can be viewed in the error message after the refactoring was started on an element without defined refactorings. The key, that is used in the JSON file, of the types or subtypes will be sent to the backend, for e.g. 'move' when moving a paragraph. Each type or subtype has to contain a label, optional is the key 'steps'. It is used for additional steps in the parameter selection.

Implementing further steps in the parameter selection

As described in 7.2, additional steps are added to the *Parameters.js*, which is located under *refactorings/components/parameters/*. The 'content-routing' is currently done with simple if-statements. It is important to note, that the number of steps has to be defined in the config file first. The new parameters can be saved via a function that is passed to the sub component (line 78). The *NavigationButtons* display the buttons in order to continue or go backwards.

```

71 //Second step
72 if(step === 1) {
73   if(params.type === 'move' && params.subtype === 'article') {
74     return (
75       <div>
76         <h1>Move {nodeType} to another existing article</h1>
77         <ArticleSelection
78           onArticleSelect={({article} => dispatch(actionSetParam('
79             targetArticle', article)))}
80         />
81         <br/>
82         <NavigationButtons enabled={params.targetArticle !==
83           undefined} handleAction={handleAction} />
84       </div>
85     );
86   }
87 }

```

Listing 7.2: Excerpt of the *Parameters.js*:71, in which additional steps in the parameter selection are shown.

References

- Buffa, M. (2006). Intranet wikis. In *Proceedings of the intrawebs workshop 2006 at the 15th international world wide web conference* (Vol. 6).
- Dohrn, H. & Riehle, D. (2011a). Design and implementation of the sweble wikitext parser: Unlocking the structured data of wikipedia. In *Proceedings of the 7th international symposium on wikis and open collaboration* (pp. 72–81). WikiSym '11. Mountain View, California: ACM. doi:10.1145/2038558.2038571
- Dohrn, H. & Riehle, D. (2011b). *Wom: An object model for wikitext* (tech. rep. No. CS-2011-05). University of Erlangen, Dept. of Computer Science.
- Dohrn, H. & Riehle, D. (2013). Design and implementation of wiki content transformations and refactorings. In *Proceedings of the 9th international symposium on open collaboration (2:1–2:10)*. WikiSym '13. Hong Kong, China: ACM. doi:10.1145/2491055.2491057
- Haase, M. (2016). *Integration und erweiterung eines visuellen editors in sweble hub*. Master thesis FAU Erlangen-Nuernberg, 98pp.
- Lee, Y. Y., Chen, N. & Johnson, R. E. (2013). Drag-and-drop refactoring: Intuitive and efficient program transformation. In *Proceedings of the 2013 international conference on software engineering* (pp. 23–32). ICSE '13. San Francisco, CA, USA: IEEE Press.
- Mealy, E., Carrington, D., Strooper, P. & Wyeth, P. (2007). Improving usability of software refactoring tools. In *Software engineering conference, 2007. aswec 2007. 18th australian* (pp. 307–318). doi:10.1109/ASWEC.2007.24
- Molli, P., Skaf-Molli, H., Oster, G. & Jourdain, S. (2002). Sams: Synchronous, asynchronous, multi-synchronous environments. In *The 7th international conference on computer supported cooperative work in design* (pp. 80–84). doi:10.1109/CSCWD.2002.1047653
- Murphy-Hill, E., Parnin, C. & Black, A. P. (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1), 5–18. doi:10.1109/TSE.2011.41
- Opdyke, W. F. (1992). *Refactoring object-oriented frameworks* (Doctoral dissertation, University of Illinois at Urbana-Champaign).
- Roberts, D., Brant, J. & Johnson, R. (1997). A refactoring tool for smalltalk. *Urbana*, 51, 61801.

- van Emden, E. & Moonen, L. (2002). Java quality assurance by detecting code smells. In *Ninth working conference on reverse engineering, 2002. proceedings.* (pp. 97–106). doi:10.1109/WCRE.2002.1173068