FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Wirtschaftsinformatik

Coupling tag-based and hierarchical information organization

Felix Michel

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Wirtschaftsinformatik

Untersuchung der Kopplung von Schlagwort-basierter und hierarchischer Organisation von Informationen

Coupling tag-based and hierarchical information organization

Author:	Felix Michel
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Alexander Steinhoff
Submission Date:	July 15, 2012

Abstract

Since the digital information processing was introduced, hierarchical tree structures are used to store and represent data. In the recent years tagging became popular in the World Wide Web. Navigating tag-based provides a new, more dynamical perspective on content. While tags changed the web, the common file system concept is basically unchanged. Nevertheless, both concepts have advantages and disadvantages. TACKO combines the advantages of tags and the structured management of information in tree structures. The name TACKO, it is an acronym for tag-based content dependent knowledge organization. Even though, nowadays hierarchical file systems are the dominant organizational paradigm, the TACKO model does not support hierarchical file systems. Aim of this work is to develop and prototypically implement a TACKO Files model which provides the missing hierachical file system support. First of all, the classical hieratical structures are transformed into tags. Based on this import a hierarchical folder structure is exported. This export allows navigating tag-based on the classical hierarchical file system. Additionally tag-based create, rename, move and delete operations are supported.

Keywords: tags, facets, mulitfacet navigation, hierachical file system, TACKO, TACKO Files, tricia

Contents

1 Introduction and Motivation

This approach is based on a model named TACKO, it is an acronym for tag-based content dependent knowledge organization. The data model of TACKO was developed to combine the advantages of tags and the structured management of information in tree structures. This model is implemented in tricia[1], a web-based enterprise collaboration and information management platform.

The fact is, that currently the storage in tree structures is the dominant organizational paradigm. This work analyzes how content of tree structures can be transformed into the TACKO data model and how these data can be accessed via existing interfaces. Therefore several transforming algorithms are systematically developed and prototypical implemented. In order to provide access on the structured imported TACKO data, the SMB[2] interface provided by tricia is used. Two basically different navigation concepts are offered via SMB. The prototypically implementation offers also a web-based interface to illustrate the import algorithms. All navigation concepts are also accessible with the web-interface and provide additionally CRUD operations. Supported operations are create, rename, move and delete. All these described functionality is prototypically implemented in the TACKO Files plugin which extend the TACKO model.

1.1 Advantages and disadvantages of tagging systems and hierarchical file systems

The amount of digital documents is continuously growing day by day. Nowadays, it becomes more and more important to access data in a structured and effective fashion. Since operating systems exist the hierarchical storage and navigation concept is applied. Today hierarchical files systems are the most common way to store any kind of digital documents. Every average user is able to use hierarchical file system in the daily life and accept it as native concept. The theoretical approach of hierarchical file systems is to store documents in hierarchical structured folders. The user must decide where a document is placed correctly in this hierarchical order. Advantage of this concept are that it is well known and supported almost everywhere. On the other hand the design is static and does not support structure documents with metadata in an effective fashion.

[1]http://www.infoasset.de/ (accessed 22th of June 2012).
[2]Server Message Block protocol, it is a common communication protocol for mounting network devices.

In the recent years, tagging becomes popular in the world wide web. It enables new flexible navigation concepts. In general tags can be assigned to every kind of information. In this case the meta data of documents is represented by tags. Multiple tags can be assigned to one document, this offers the opportunity to provide document meta data in a flexible way. In contrast to folder trees, there exists no natural and effective way to access tagged contents in a structured way.

Coupling a tag-based systems with a hierarchical file systems provides the key advantages of both concepts and reduces the restrictions. Documents are hierarchical accessible based on tags. The TACKO model of tricia provides already accessing tagged resources in a structured way. Folders and documents are also taggable but the tags must be assigned manually. The TACKO Files model additionally accomplish consistency between the tags and the hierarchical file system. This model closes the gap between both concepts.

1.2 Structure of this thesis

First of all, hierarchical and tag-based fundamentals are covered in Chapter 2.1. Different common navigation concepts are illustrated. Based on the fundamentals, the important related work is explained in Chapter 2.2. All design related issues are treated in Chapter 3. This includes the different navigation concepts, native tag-based, multi facet tag-based and the TACKO hierarchical facet navigation. The user interface for create, rename, move and delete delete operations are described corresponding to the different navigation concepts. Furthermore the prototypical web-based TACKO Files user interface is presented. Design priorities are briefly described and justified. Moreover facet testcases are defined based on hierarchical file system with samples. In Chapter 4 all related algorithms are explained and visualized. In general this chapter illustrates the algorithms in the order of the first usage, beginning with the tag import. To access these tags, several algorithms provide fundamental functions. With the aid of these basic functions all export algorithms are described. According to the crud user interface mapping algorithms are described to initialize and execute these operations. The integration environment is briefly explained in Chapter 5.1. Chapter 5.2 outlines the basic TACKO Files architecture. The general plugin structure is present. Moreover mainly package dependencies within the TACKO Files plugin are illustrated. Consecutive to the architecture the most important implementation parts are described in Chapter 5.3. Finally the Chapter 6 illustrate a real data sample. The last Chapter 7 summarizes the result of this work and gives a short outlook.

2 Organisation structures for information

All related organization structures for this thesis are covered in this chapter. Basically it is divided into two subchapters. In the first, all important fundamentals are explained (see Chapter 2.1). Further the related work is illustrated (see Chapter 2.2).

2.1 Fundamentals

This Chapter describes fundamental hierarchical and tag-based knowledge. In the first part illustrates the basics of hierarchical file systems (see Chapter 2.1.1). Moreover the general tag-model is explained and different kind's tag-based navigation are illustrated (see Chapter 2.1.2). Additionally the TACKO concept is explained (see Chapter 2.1.3). Finally the presented fundamental concepts are briefly summarized (see Chapter 2.1.4).

2.1.1 Hierarchical-based navigation

This Chapter is written in according to the paper Going beyond the hierarchical file system [Arr03]. The hierarchical tree navigation is the most common way to access all kinds of digital information. All important operating systems provide a hierarchical tee view on folders and documents.

> "Back in the early days of computing, someone decided that the "natural" way to organize documents stored in electronic format was in folders. Users would create a tree of directories and subdirectories, and documents would go into those folders. The combination of a document's "path" and its file name uniquely identified a document, and constituted the only practical way to store that document's metadata." [Arr03]

Nowadays most users perceive this structure as intuitive and use it in the daily life. The theoretically perfect folder structure represents a taxonomy. A folder represents a category and contains subfolders which represent subcategories. The hierarchical file system path defines one possible sequence of categories, expressed with a folder and their subfolders. Searching for a certain document means incrementally adding more specific categories to the navigation path until the document is found. This navigation concept works pretty well with data which represent a taxonomy. "A taxonomy is a controlled vocabulary that establishes parent-child, or broader and narrow, relationships between terms. Taxonomies are typically hierarchical." [Smi08, p.72] Adding a new document means to find or create the corresponding path which represents the new documents meta information. E.g., a teaching assistant from the Technical University of Munich wants to add the slides of the first software engineering lecture for the summer term 2012.

Sample path: `\teaching\2012\software engineering\lecture01.pdf`

The sample illustrates a possible path which covers all corresponding document metadata in the path. A path contains also the document name and represents the most specific meta information. For the sample meta data, does not exist a really clear taxonomy there are existing more variants. This illustrated variant is the most common one but the year folder could be moved up or down within the hierarchical path. This ordering problem is really common in real data.

There is one hard restriction for all hierarchical file systems, a document can only be placed in one path. In the digital life with a continuously growing amount of data this is a not negligible disadvantage. Imagine within a hierarchical file system two paths `\it\hardware\` and `\it\software\` already exist. Adding a new hardware driver needs some semantic decisions to choose one of these locations. Independent from the decision which path is chosen, the path will not express all necessary metadata. Navigating in hierarchical file systems which contain such folders needs additional user knowledge. Copying the data in both folders is possible but leads to redundancy. One approach to improve the situation is using shortcuts, all modern operating system provide such an option in some way. In certain cases shortcuts can help but to provide considered shortcuts the must be updated after the linked resources is moved or deleted. Another deficit is deleting or moving the shortcut does not effect the corresponding resource. In general the single location problem is not solvable in hierarchical file systems.

2.1.2 Tag-based navigation

2.1.2.1 Tagging systems

Every tag-based model contains three major components (Figure 2.1). A resource describes the taggable content. The second component is the tags itself. In general, tags are textual labels, a word or a phrase. A tag is metadata which describes the resource content. Resources can be assigned with multiple tags, there is theoretically no limit. Keyword is a colloquial synonym for tag. Users assign tags to resources depending on several conditions. Content dependent expertise helps to tag the resource with exact keywords. Due to different domain specific knowledge, languages and goals, there are existing synonyms within each tag-based model. The resources location is transparent for the user and is normally only represented by the assigned tags. Navigation on a tag-based system means basically searching for all resources which has assigned the searched tag. The previous paragraph is summarized form the book Tagging: People-Powered Metadata for the Social Web [Smi08, p.4f.]

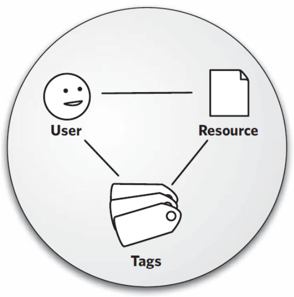


Figure 2.1: Basic tagging system [Smi08, p.4f.].

In the recent years, several social tagging systems became popular. Basically every social tagging system has shared resources which are tagged. Users tag resources with freely chosen vocabulary. Therefore no common taxonomy is exists within a the set of tags but

> ”the popular tags in social tagging systems have recently been termed folksonomy [...] a folk taxonomy of important and emerging concepts within the user group.” [MNdbD06]

A famous sample is Delicious[1], it use bookmarks which represent basically a certain url and a description as resources. Filicker[2] is another example, photos are used as resources. YouTube[3] and Last.fm[4] using also tags to structure their content. This paragraph is summarized from the Tagging Paper [MNdbD06]

[1]http://delicious.com/, (accessed 28th of June 2012)
[2]http://flicker.com/, (accessed 28th of June 2012)
[3]http://youtube.com/, (accessed 28th of June 2012)
[4]http://last.fm/, (accessed 28th of June 2012)

2.1.2.2 Tag-cloud navigation

Tag-clouds are a popular visualization and navigation concept of tags. Each tag represents a single unstructured textual label. There are several different methods for ordering, placing and highlighting them in the tag-cloud. Most tag-clouds are ordered lexicographically Figure 2.2 illustrate a sample. The font size is defined dependent on the frequency of a tag. Probably the most frequent tags are also the most important ones. In general the size can depend directly proportional to the frequence or linear. A maximum and minimum font size is defined for the tag labels. In real data samples a small amount of tags is used very often and a huge amount of tags are only used for some resources. If the size is proportional related to the frequence of a tag, most tags are visualized with the minimum font size and only a few with the maximum font size. In between there only less tags. Normally a logarithmic frequence dependence scales better. More tags are in between the minimum and maximum font size. This increase the legibility enormously. The tag-cloud is a compact representation of tags but there also some negative aspects. The word length of a tag influence the awareness. A longer tag label seems to be more important as a shorter. Further there is no semantic relation contained within the representation. Relation between tags are not expressed. E.g. the tag `operation system` and `windows` is placed and represented in the tag-cloud. In this case most resources which are tagged with `operation system` are also tagged with `windows`. Both tags are represented in a similar font size. This prevents a clear navigation structure and may confuse the user. This chapter is summarizes from the paper Tag Clouds: Data Analysis Tool or Social Signaller [HR07] and [Smi08, p.97ff.].

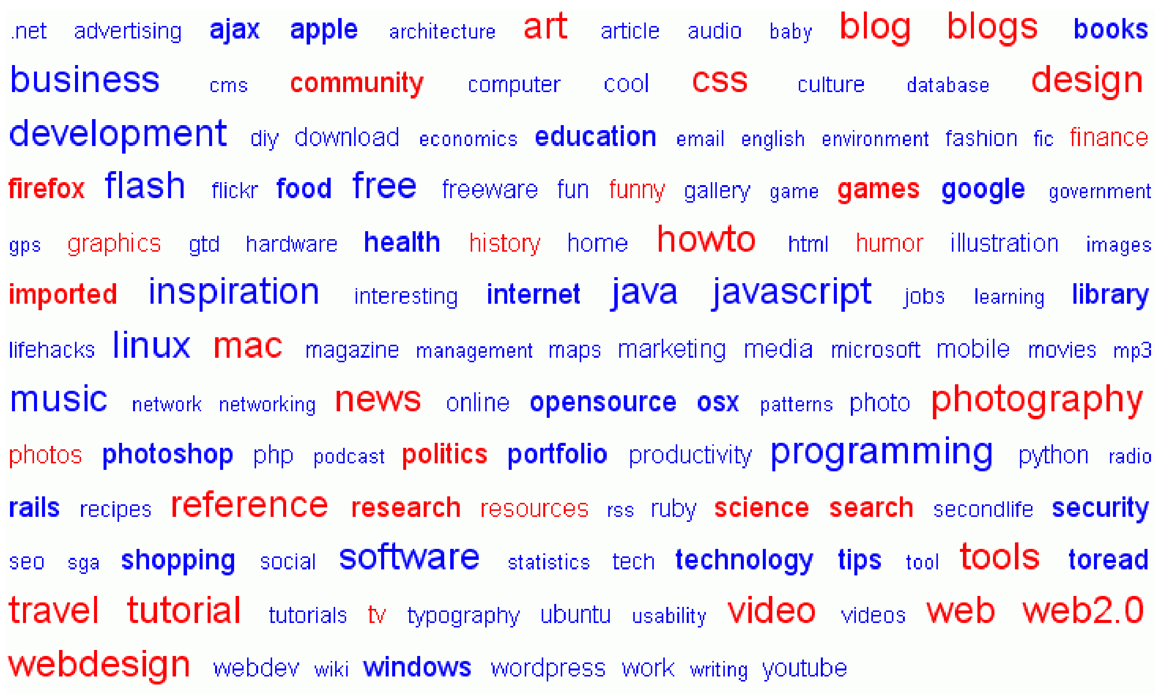

.net advertising ajax apple architecture art article audio baby blog blogs books
business cms community computer cool css culture database design
development diy download economics education email english environment fashion fic finance
firefox flash flickr food free freeware fun funny gallery game games google government
gps graphics gtd hardware health history home howto html humor illustration images
imported inspiration interesting internet java javascript jobs learning library
lifehacks linux mac magazine management maps marketing media microsoft mobile movies mp3
music network networking news online opensource osx patterns photo photography
photos photoshop php podcast politics portfolio productivity programming python radio
rails recipes reference research resources rss ruby science search secondlife security
seo sga shopping social software statistics tech technology tips tool tools toread
travel tutorial tutorials tv typography ubuntu usability video videos web web2.0
webdesign webdev wiki windows wordpress work writing youtube

Figure 2.2: Alphabetical ordered tag-cloud with most popular delicious tags [HR07].

2.1.2.3 Faceted search

Facet definition

"Facets refer to categories used to characterize information items in a collection." [Hea06, p.1] Every facet represents a category and contains the corresponding tags. The tags represented the information items. All facets are named to describe items of the category summarized with one word or phrase. E.g. the facet named `teaching` contains the tags `software engineering`, `databases` and `discrete structures`.

Parametric search

The parametric search visualizes facets and the user builds a search query based on this facets. All or certain facets are listed with the corresponding items. Users select items to add this item to the query. All facet "values selected within a single facet are combined using a logical OR, whereas constraints associated with different facets are combined using a locigal AND." [MTT$^+$09, p.21] The search is executed and presented to the user. With more selected items in different facets the probability strongly increase that no matching result exist. Maybe the documents are disjunct and split into two categories, both are selected in the query. It is not clear for users which item causes the empty search result. This paragraph is summarized form the book Synthesis Lectures on Information [MTT$^+$09, p.24ff.].

Faceted search

Basically the facet search is an improvement of the parametric search. The initial situation is similar to the parametric search, all facets are visible. Selecting facet values works incremental. After each new selection the resulting documents are updated. With each selection the search is more specific and the result set is probably smaller. Users can try selecting an item and evaluate the corresponding results. If the search contains the desired results, one more facet value can be applied as filter, otherwise it is possible to deselect the facet item and get the previous search result. In comparison to common searches this facet search provides meaningful structured browsing and discovering new content. This paragraph is summarized form the book Synthesis Lectures on Information [MTT$^+$09, p.24ff.].

2.1.3 TACKO

This paragraph summarizes the TACKO model described in the paper multifaceted context dependent knowledge organisation with TACKO [MNS12]. Based on the facet search concept the model provides additional functionality. The model uses freely chosen tags which are assigned to resources. Several different kinds of resources are taggable such as content of a wiki page or a document. Tags per resources are not limited. The resource search works with filters. Each filter contains a certain set of tags. Searching for a resource means to apply a filter on the set of all resources. The result of the filter contains only these resources which assign all tags of the filter. Additionally, complementary tags can be assigned to filters. These synthetic complementary tags exclude all resources which are contain this tag. Incremental adding filters reduces the result set. The model supports also expressing hierarchical tag relations in a certain context. Resulting tags of a filter represent a context. Relations like is `part of` are expressed with the defined subsumption relation. E.g. tag `car` subsumes `audi`, that means only resources in the context of `car` are to assigned the tag `audi` but never outside of this context. Subsumption relations are only possible in a context without complementary tags.

> ”When a user searches for a general tag, such as `researcher`, all resources being tagged with more specific tags, i.e. the tags subsumedby `researcher` shall be retrieved as well. Therefore, it has to be guaranteed that all resources tagged `postdoc` or `doctoral candidate` are tagged `researcher`, too.” [MNS12]

Using subsumption relations to express hierarchies has several advantages. A resource can be assigned with many tags and is part of different categories. Furthermore different categories are clearly separated as facets. Facets allow to organize categories, all tags of the same dimension form one facet. The venn diagram in Figure 2.3 illustrates two competing facets. In the general context, there are existing two facets. Horizontal the facet year is visualized, blue colored and vertical another facet is represented by the tags `projects` and `presentations`. Within the context `projects` exists one facet, green colored.

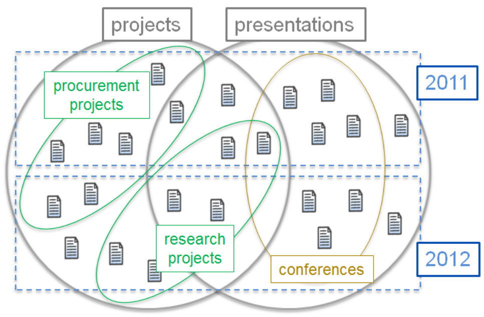


Figure 2.3: Context dependent tag-relations [MNS12]

The TACKO data model is illustrated with this sample data on Figure 2.4. To use a consistent notation for facets within the whole document, this representation differs from the original TACKO notation. The before defined filters, represent a context. Every context has a set of tags and contains one or more facets. A facet represents a category with a set of tags. All facet tags must be subsumed from all context tags.

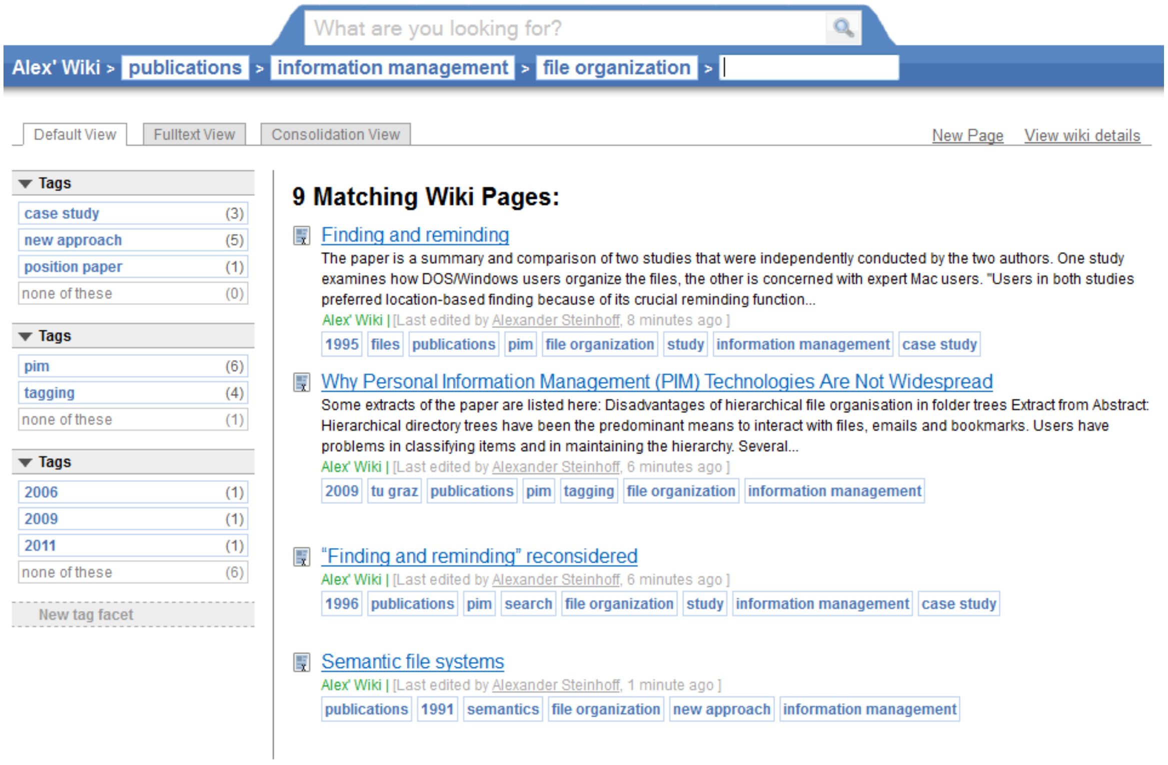
$$\begin{aligned}
&\textbf{Context} \rightarrow && \{\} \\
&\text{Facet}_1 \rightarrow && \{\text{projects, presentations}\} \\
&\text{Facet}_2 \rightarrow && \{2011, 2012\} \\
\\
&\textbf{Context} \rightarrow && \{\textbf{projects}\} \\
&\text{Facet}_1 \rightarrow && \{\text{research projects, procurement projects}\}
\end{aligned}$$

Figure 2.4: TACKO data model: Facets with corresponding context.

Based on the explained data model, a multifaceted navigation is offered. The represented UI in Figure 2.5 is separated in several areas. The blue bar on top supports a full text search and illustrates the tags of the current context. These tags represent the navigation path but the order has no impact on the matching wiki page results. The input field at the end of the navigation path allows to specialize the context with an additional tag. Clicking on a tag in the path generalize the context, all tags on the right are removed from the current context. Facets are presented context related on the left.

> ”To determine the set of visible facets, all filters[5] being more general than the current filter are taken into account. If for one of these filters a facet is defined, it is displayed unless all resources matched by the current filter belong to the same categories with regard to this facet [...]” [MNS12].

That means facets can be inherited. This sample has three facets. Every facet offers another dimension to filter the results of the current context by adding more tags to the context. E.g. the second facet contains the tag `pim` and `tagging`, these tags are blue colored. There is one more label named `none of these`, colored in gray. Behind each facet tag a number shows the amount of matching results for this subcontext. Clicking on a facet tag extends the context with this tag and shows the corresponding resources.


What are you looking for?
Alex' Wiki > publications > information management > file organization >
Default View
Fulltext View
Consolidation View
New Page
View wiki details
New tag facet

Tags	
case study	(3)
new approach	(5)
position paper	(1)
none of these	(0)

Tags	
pim	(6)
tagging	(4)
none of these	(1)

Tags	
2006	(1)
2009	(1)
2011	(1)
none of these	(6)

9 Matching Wiki Pages:

Finding and reminding
The paper is a summary and comparison of two studies that were independently conducted by the two authors. One study examines how DOS/Windows users organize the files, the other is concerned with expert Mac users. "Users in both studies preferred location-based finding because of its crucial reminding function...
Alex' Wiki | [Last edited by Alexander Steinhoff, 8 minutes ago]
1995 | files | publications | pim | file organization | study | information management | case study

Why Personal Information Management (PIM) Technologies Are Not Widespread
Some extracts of the paper are listed here: Disadvantages of hierarchical file organisation in folder trees Extract from Abstract: Hierarchical directory trees have been the predominant means to interact with files, emails and bookmarks. Users have problems in classifying items and in maintaining the hierarchy. Several...
Alex' Wiki | [Last edited by Alexander Steinhoff, 6 minutes ago]
2009 | tu graz | publications | pim | tagging | file organization | information management

“Finding and reminding” reconsidered
Alex' Wiki | [Last edited by Alexander Steinhoff, 6 minutes ago]
1996 | publications | pim | search | file organization | study | information management | case study

Semantic file systems
Alex' Wiki | [Last edited by Alexander Steinhoff, 1 minute ago]
publications | 1991 | semantics | file organization | new approach | information management

Figure 2.5: Multifaceted navigation with TACKO [MNS12].

[5] A filter represent a context.

2.1.4 Summary

Hierarchical organization is easy to apply for data which represent a taxonomy. Real resources are not always assignable to exactly one category. This is the main limitation of hierarchical file systems. Tag based-models support to assign a resource with multiple tags. Resources are stored transparent for users and they are accessible with tags. Several different tag-based navigation concepts exist. Tag-clouds normally visualize tags dependent on the frequency. There is no really useful structure for navigating in an effective fashion. The parametric search in contrast provides a structured boolean search. In case of usability it is a problem that empty search results occur. Basically the concept of the facet search works very similar. The main difference is the incremental search concept which prevents the empty search results. TACKO is an advanced tag-based model which combines the facet navigation with a hierarchical navigation structure.

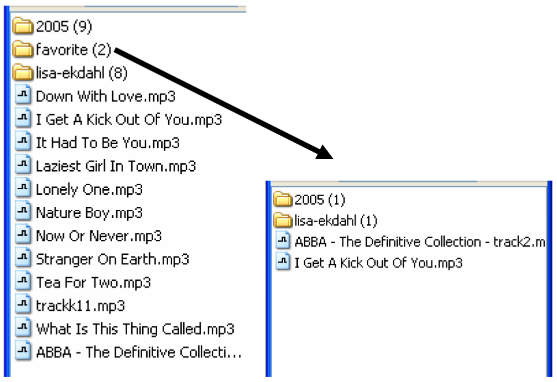
2.2 Related Work

This chapter describes all important related work corresponding to map tag-based systems to hierarchical file systems. All fundamental definitions are in the previous Chapter 2.1.

2.2.1 Tag semantic for hierarchical file systems

The following paragraph summarizes the paper Tag Semantics for Hierarchical File Systems [SB06]. All monospace written phrases in this subchapter are directly cited interface names from the previous mentioned paper. Approach of this paper is to map a tag-based system to a hierarchical source file system. Hierarchical file systems bind resources to a single location. That means a resource is only accessible with exactly one path. This single location restriction leads to several problems. All folders and subfolders must represent a strictly hierarchical relation to ensure the maximum specific browsing concept. Further orthogonal dimensions are not meaningful representable, they must be expressed serialized in the path. With this serialization an ordering issue occurs. A path should express hierarchical ordered categories, this is not possible for orthogonal categories. Aim is to combine the advantages of tagging with the advantages of a hierarchical file system.

The design in general defines that folders represent tags. Figure 2.6 illustrates a browsing sample. In general all tags in the tagging system are listed in the root folder together with all existing files. Selecting certain tags filters the resulting resources. E.g. the tag favorite in the left part of the sample is selected by clicking. The arrow points to the resulting view.



```
2005 (9)
favorite (2)
lisa-ekdahl (8)
Down With Love.mp3
I Get A Kick Out Of You.mp3
It Had To Be You.mp3
Laziest Girl In Town.mp3
Lonely One.mp3
Nature Boy.mp3
Now Or Never.mp3
Stranger On Earth.mp3
Tea For Two.mp3
trackk11.mp3
What Is This Thing Called.mp3
ABBA - The Definitive Collecti...
```

```
2005 (1)
lisa-ekdahl (1)
ABBA - The Definitive Collection - track2.m
I Get A Kick Out Of You.mp3
```

Figure 2.6: TagFS browsing a hierarchical file system tag-based [SB06].

Further, mapping tags to hierarchical file systems is structured into three main parts. There are two methods offered to browse this view. The first method **`view(`**`location`**`)`** queries for all files which are tagged with the tags of the location. The query is conjunctive and permuted tags result in the same resource. To represent the resources in a meaningful structure subfolders which represent tags are accessible with the method **`subfolders(`**`location`**`)`**. Empty folders in the hierarchical source file system are not considered.

There are also methods to modify the existing tags. To remove a tag from a file the method **delete(**`location, file`**)** is offered. This "[...] removes the last tag from the specified file." [SB06] The method name is a bit confusing. "In order to really delete a file, it has to be tagged with delete." [SB06] Another method provides copy tags, **copy(**`location source, location target, file`**)**. This means to "[...] assign additional tags from the target folder to the file in question." [SB06] Additionally, a move operation is provided. It works according to the delete and copy methods.

The third kind of method is used to add new files. All folders names contained in the path of new files are used to tag the file. Due to the design, all filenames within the system must be unique. This allows listing all files without conflicts on the root level of the tag-based view. Files which cause conflicts are renamed.

In summary, this model maps a tag-based view with corresponding operations on top of a hierarchical file system. On the root level of the tag based view all resources are represented. Imagine the system contains a huge amount of files. A common hierarchical file system contains normally more than a few folders, that means a lot of tags are listed on the root level. Humans are only able to select one item out of seven in an effective way. Therefore, navigation design is not as powerful as desired for real samples.

2.2.2 Hybrid approach to construct tag hierarchies

This chapter summarizes the paper A Hybrid Approach to Constructing Tag Hierarchies [GS10]. "Folksonomy tags do not generally have any associated structure. [...] The intention of this paper is to generate a semantic structure of tags in the folksonomy that can later support semantic information access."[GS10] This approach assumes that all tagged resources are textual. All assigned tags t_i of a resource are represented by a tag vector $V = [t_1, t_2, ..., t_n]$. A second vector expresses the $weight$ w_i of every tag t_i corresponding to tag vector: $T = [w_1, w_2, ..., w_n]$. The $weight$ is dependent on the textual resource content. The amount of every tag within a resource represents the $weight$ w_i. The vector T is used to evaluate the structure. "The first step in the process is to run association rule mining on the set of tags, using the Apriori algorithm [...]" [GS10]. This results in a frequent itemset which represents the association rules. An association rule expresses if a resource assigns a tag T_i there is a certain probability that the resource assigns also another tag T_j. The tag T_j is defined as the consequence tag of premise tag the T_i. E.g. a resource is tagged with the tag `audi`, there is a high probability that the resource also assigns the tag `car`. This associations are evaluated with a confidence function (Figure 2.7). The confidence express "[...] the percentage of observations that contain the premise and that also contain the consequence [...]" [GS10].

$$confidence(T_i \rightarrow T_j) = P(T_j|T_i) = \frac{supportCount(T_i \cup T_j)}{supportCount(T_i)}$$

Figure 2.7: Confidence function [GS10].

This association rules represent the initial hierarchical relations, "[...] the premise of the rule may be viewed as a child of the consequence." [GS10] Finally the "[...] construction of the hierarchy starts with an empty root node. Next we and all consequences which do not appear as premises of any rule. These are added as direct children of the root node. For each first level child, all tags that appear as premises of rules with the first level child as consequence are added as children. This process continues until there are no more children to add, or an attempt is made to add a tag that already exists in the path from the root to the current node." [GS10] The structure verification is done in an additionally step, this is not summarized here.

In general this concept is powerful. It generates a semantic hierarchical structure based on tags with a relation to the textual resources. A disadvantage is that all resources must be textual.

2.2.3 Study - Don't take My Folders Away

This Chapter is a briefly summary from the study Don't take My Folders Away [JPGB05]. A group of 14 people was participated in the study. The first question was, why thy create folders. All participants answered: "in order to get back to my files" [JPGB05]. The main question of this study was:

> "Suppose that you could find your personal information using a simple search rather than your current folders...Can we take away your folders? Why or why not?" [JPGB05]

Out of the 14 participants 13 answered with no. A main reason was the participants do not trust only a search. Moreover they want to have the control that all files which are related to each other, grouped in one folder. The third reason is related to the visibility and understandability of the structure. On the other hand the following statement illustrated the need for an advanced navigation or search concept:

> "All of the participants said they would be happy to have search utility that helped them to find their personal information better." [JPGB05]

2.2.4 Summary

This chapter summarizes the previous chapters briefly and refers more related work. Most important for this work is the tag semantic approach (see Chapter 2.2.1). The conceptual idea of mapping a tag-bases view on a hierarchical file system is also used in this thesis. Apparently, tag-based navigation is very limited and not applicable on real data samples. The tag import stores the path related tags amorphous. Additional structural information included in the path is lost. The other approach illustrated in Chapter 2.2.2 generates hierarchical relations based on tags. In general the concept is based only on textual files and not usable for the most common binary file formats. Nevertheless, this approach was helpful for developing the facet import algorithm.

The paper "Can You Retrieve a File on the Computer in your First Attempt? Think to a New File Manager for Multiple Categorization of Your Personal Information" [SAZ12] and "Supporting Multiple Categorization using Conceptual File Management" [ASB11], illustrate a new approach to solve the single location issue (see Chapter 2.1.1) in hierarchical file systems.

Finally the study in Chapter 2.2.3 illustrates the need for an combined approach. The participants want to have the possibility to define a structure and accessing these in different ways.

3 Design

This chapter covers all design related issues. The general context is described and the relation between the TACKO model and the TACKO Files extension is explained. Design priorities are defined and explained (see Chapter 3.1). All different tag-based navigation concepts are illustrated (see Chapter 3.2). Related to the tag-based navigation concept CRUD operations such as create, rename, move and delete are illustrated (see Chapter 3.3). Moreover the web-based TACKO Files user interface is presented (see Chapter 3.4). The relation between this user interface and the navigation concepts with the corresponding CRUD operations is illustrated. Further visionary user interface scenarios are briefly described (see Chapter 3.5). Finally facet import test cases presented and discussed (see Chapter 3.6). These test cases are really important, they define the behavior of the complete navigation concept.

All further algorithms and user interfaces are based on a common hierarchical file system. A tag-based view is mapped on top of the source file system. That means all tag-based operations must be transformed into hierarchical file system operations. Each operation is mapped to hierarchical operations and executed on the source system. Figure 3.1 illustrates the general context. In the first step, the hierarchical source file system is imported into the TACKO model. There are two different import functions, a simple import and a facet import function. The simple one imports tags based on the names of the directory path. Importing the facets is more advanced and considers the hierarchical structure. All imports are represented in the TACKO data model. Based on this data model different navigation concepts are offered. By default, the TACKO web interface provides a hierarchical multifaceted navigation. Furthermore the TACKO Files model offers a hierarchical native or a multifaceted[1] tag export. Export tags are used as synonym for presenting a navigation concept. Additionally it is possible to export the native or multifaceted view also with a postfix which counts the documents per tag. All export options can be accessed via the web-interface or a mountable network device. In the Figure 3.1 two possible export concepts are presented. In general there are existing $2 * 2 * 2 = 8$ different combinations. Based on the web-interface, create, rename, move and delete operations are provided corresponding to the navigation concept. A context menu offers different options depending on the resource. Related to the operation a dialog shows all necessary information. Executing any CRUD operation means to modify the hierarchical source file system. After the execution the consistency between the hierarchical file system and the tags respectively the facets is no more given. The tags and facets are updated to a consistent state.

[1]The multifaceted tag-based navigation a synonym for group by tag-based navigation.

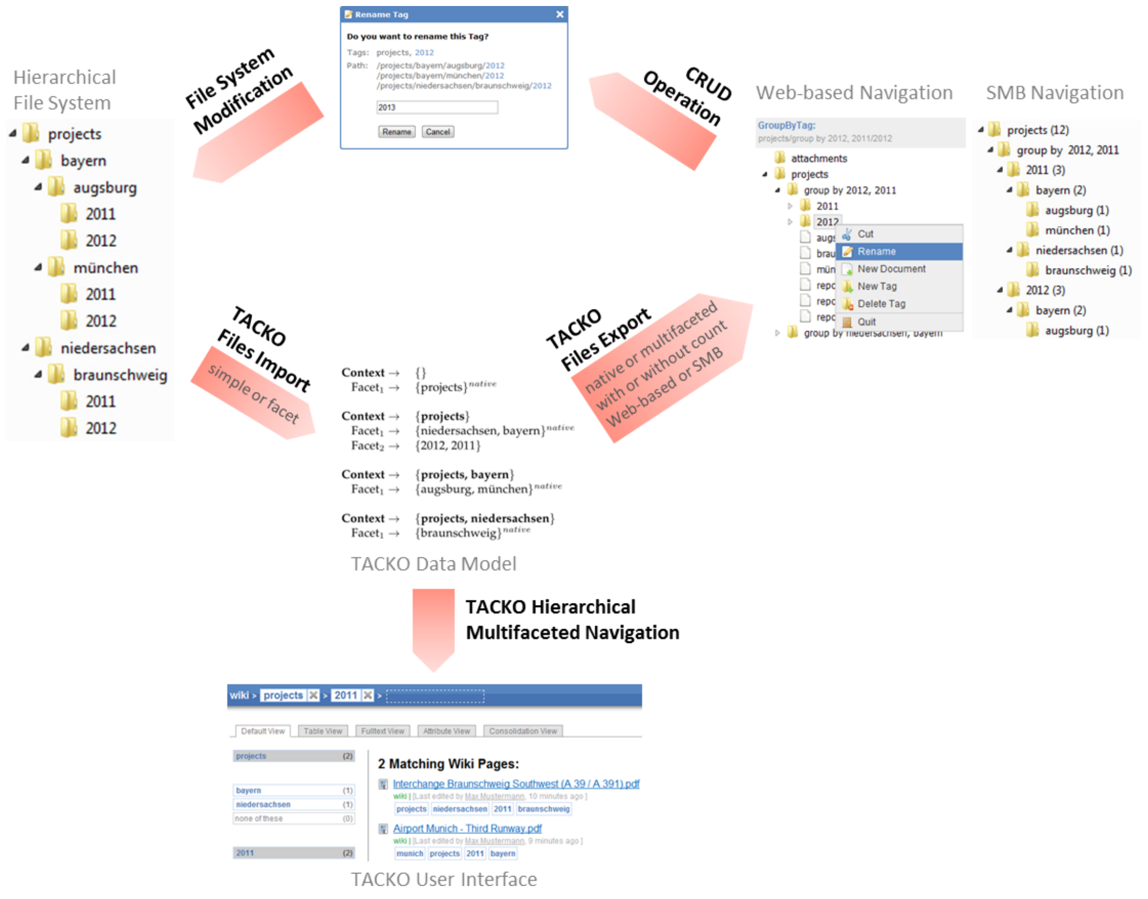

Hierarchical File System
File System Modification
Rename Tag
Do you want to rename this Tag?
Rename
Cancel
CRUD Operation
Web-based Navigation
SMB Navigation
TACKO Files Import
simple or facet
TACKO Files Export
native or multifaceted with or without count Web-based or SMB
Context → {}
Facet₁ → {projects}^native
Context → {projects}
Facet₁ → {niedersachsen, bayern}^native
Facet₂ → {2012, 2011}
Context → {projects, bayern}
Facet₁ → {augsburg, münchen}^native
Context → {projects, niedersachsen}
Facet₁ → {braunschweig}^native
TACKO Data Model
TACKO Hierarchical Multifaceted Navigation
2 Matching Wiki Pages:
TACKO User Interface

Figure 3.1: General context: Hierarchal file system - TACKO - TACKO Files.

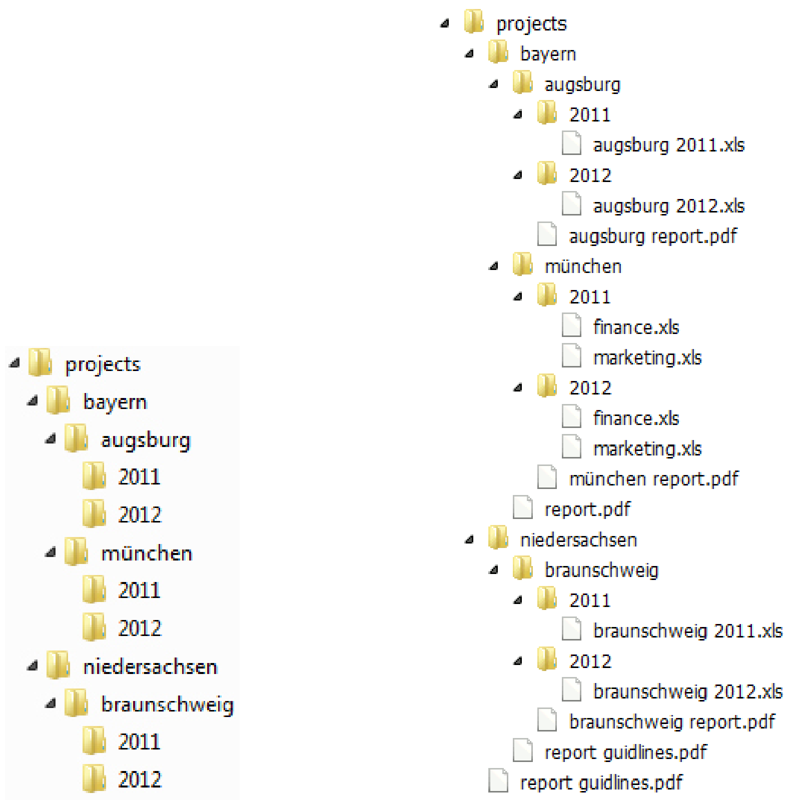
3.1 Design priorities

Offering different hierarchical tag-based navigation concepts (see Chapter 3.2) is the primary requirement. Further a possibility to compare the common hierarchical file system navigation with the tag-based navigations is provided (see Chapter 3.4). With this comparable view, all transforming algorithms can be easily visually analyzed. This provides evaluating algorithm changes and enables to conclude theories. Usability of tag-based navigation views and crud operations are important but for a better algorithm understanding, more information as necessary is provided. All other not usability optimized views only help to understand or initialize the system and they are not necessary for a productive system. Performance optimization is secondary and not the main part of this work.

3.2 Navigation Concepts

The TACKO Files design offers two basic user interfaces for tag-based navigation. Basically the TACKO data model is used as input data (see Chapter 2.1.3). The native tag-based navigation concept (see Chapter 3.2.1) is basically similar to the hierarchical file system navigation concept. Furthermore, multifaceted (see Chapter 3.2.2) tag-based navigation provides more advanced navigation information. Both kinds offer an extended view with counted documents per tag. All described navigation concepts can be mounted as a network device and accessed with a common hierarchical file system browser. E.g., in windows this file browser is named explorer. Moreover the TACKO model offers a web-based hierarchical multifaceted navigation concept (see Chapter 3.2.3). Due to technical conditions, a prototypical web-based interface (see Chapter 3.3) is additionally offered to illustrate the create, rename, move and delete operations concept.

Figure 3.2 represents the example hierarchical source file system for all following subchapters. The default view via network device shows only the folders within the navigation tree. All documents and subfolders of the current selected path are placed in a separated frame. The web-based interface provides the hierarchical view with all documents.



```
projects
  bayern
    augsburg
      2011
      2012
    münchen
      2011
      2012
  niedersachsen
    braunschweig
      2011
      2012
```

```
projects
  bayern
    augsburg
      2011
        augsburg 2011.xls
      2012
        augsburg 2012.xls
      augsburg report.pdf
    münchen
      2011
        finance.xls
        marketing.xls
      2012
        finance.xls
        marketing.xls
      münchen report.pdf
    report.pdf
  niedersachsen
    braunschweig
      2011
        braunschweig 2011.xls
      2012
        braunschweig 2012.xls
      braunschweig report.pdf
    report guidlines.pdf
  report guidlines.pdf
```

(a) without documents via SMB (b) with documents via web-interface

Figure 3.2: Hierarchical source file system.

The hierarchical source file system (Figure 3.2) is imported into the TACKO data model (Figure 3.3). Based on this data model the navigation concepts are exported.

Context $\rightarrow$ $\{\}$
Facet$_1$ $\rightarrow$ $\{\text{projects}\}^{native}$

Context $\rightarrow$ $\{$**projects**$\}$
Facet$_1$ $\rightarrow$ $\{\text{bayern, niedersachsen}\}^{native}$
Facet$_2$ $\rightarrow$ $\{2011, 2012\}$

Context $\rightarrow$ $\{$**projects, bayern**$\}$
Facet$_1$ $\rightarrow$ $\{\text{augsburg, münchen}\}^{native}$

Context $\rightarrow$ $\{$**projects, niedersachsen**$\}$
Facet$_1$ $\rightarrow$ $\{\text{braunschweig}\}^{native}$

Figure 3.3: TACKO data model corresponding to the hierarchical source file system.

3.2.1 Native tag-based navigation

Native tag-based navigation is closely related to navigation on a hierarchical file system. Hierarchical organization is the most common manner in the digital life, for personal use as well as in companies. Each average user knows intuitively how to browse in a windows file explorer. Everyone who understands the concept of hierarchical navigation should be able to use this native tag-based navigation concept.

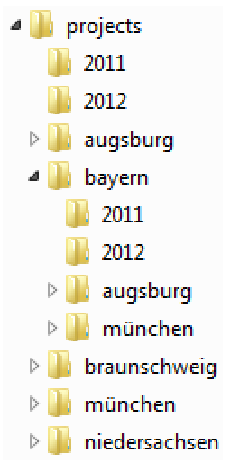
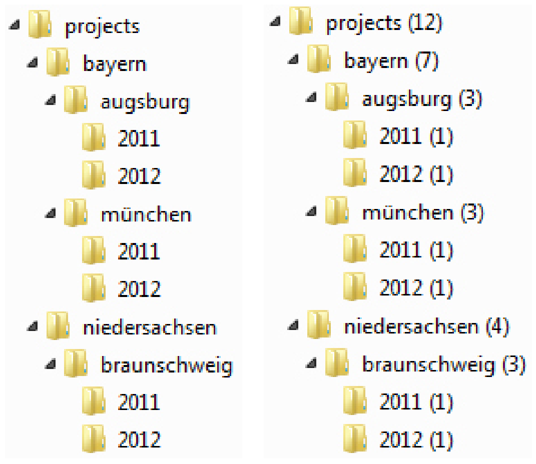


Figure 3.4: Simple tag navigation.

In general folder symbols in the tag-based navigation concept, represents a tag. Figure 3.4 illustrates a simple tag navigation. On the root level all existing tags are listed. Selecting a tag filters the current representation and shows all resources which are additionally assign the selected tag. E.g. clicking on the tag `bayern` shows the tags `2011`, `2011`, `augsburg` and `münchen`. The amount of sibling tags explodes with an increasing amount of folders. This would end up in a messy data structure in comparison to a hierarchical file system. Imagine all of your files and folders are listed on one hierarchical level. There is no chance to navigate in a meaningful fashion. This sample illustrates that a simply tag-based navigation is not powerful enough to use it with a huge amount of tags and documents. Approach of this work is to define and develop a better navigation concept.

Related to the imported hierarchical source file system Figure 3.5(a) presents the native tag-based navigation interface. This interface is based on the TACKO data mode. Each folder in a path represents a tag, all path tags together describe a context. A context is a set of tags. E.g. the path `\projects\bayern\` represents the context {`projects, bayern`}. To each context one or more corresponding facets are bound. A facet represents basically a set of sibling folders. In this case {`bayern, münchen`} is one facet. In other words, the context {`projects, bayern`} has one facet {`augsburg, münchen`}.

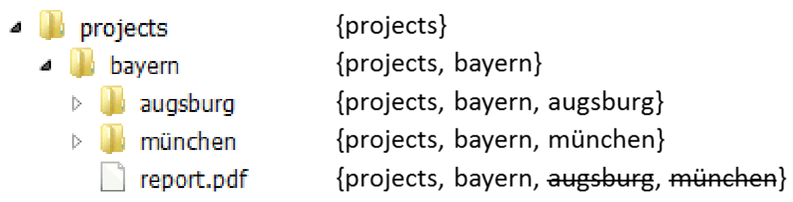
Clicking on a tag, represented by a folder, means to search for a certain context. The searched context must contain all tags corresponding to the path. The first facet of the context is represented as child tags. E.g., clicking on `projects` leads to a search for a context which contains all path tags. In this case the path contains only one tag: `projects`. One context is found and the corresponding first facet {`bayern, münchen`} is presented as child tags of the context. In the following, another example is described. By clicking on the folder `augsburg` the context {`projects, bayern, augsburg`} is searched. The corresponding first facet {`2011, 2012`} is appended as a child to the folder `augsburg`.



(a) default (b) with count option

Figure 3.5: Native tag-based view via SMB.

Within a tag-based folders, all documents, which are tagged with all tags of the context are listed. Further these documents must not contain the additional tags of the sub contexts (Figure 3.6). E.g. the document `report.pdf` is tagged with `projects` and `bayern`, the corresponding context is {`projects, bayern`}. Sub contexts are {`projects, bayern, augsburg`} and {`projects, bayern, münchen`}. In this case `augsburg` and `münchen` are additional sub context tags. All documents which are placed in exactly the same context {`projects, bayern`} must not contain the tags `augsburg` and `münchen`. These tags are defined as not path tags. In other words, all documents within a tag folder must contain the path names as tags and must not contain the child tag folder names as tags. When a document contains such a tag, it is placed in a sub context.



{projects}
{projects, bayern}
{projects, bayern, augsburg}
{projects, bayern, münchen}
{projects, bayern, ~~augsburg~~, ~~münchen~~}

(a) sample. (b) corresponding tags.

Figure 3.6: Document placement sample.

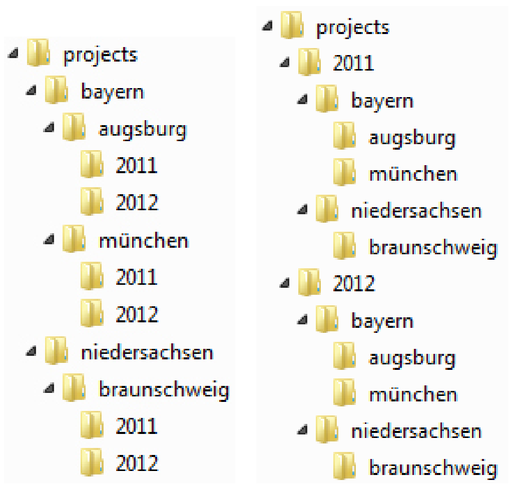
By comparing the Figures 3.5(a) and 3.5(b), it is easy to recognize that both representations are equal with exception of the post numbering. A number in brackets expresses how many documents are contained in this context.This is context dependent. E.g., the tag `augsburg` is assigned to three documents in the context {`projects, bayern`}.

The native tag-based navigation concept (Figure 3.5(a)) compared to the hierarchical source file system (Figure 3.2) has no structural difference. This is desirable to provide a well-known navigation behavior. However this navigation concept is much more powerful than the common hierarchical concepts. The hierarchal navigation structure is dynamically generated, this allows to adapt additional functionality. Counting the amount of documents is one possible example. Furthermore it is conceivable to append other kinds of tagged resources dynamically to the generated structure. Such a concept allows to integrate all tagged resources into this structure such web-pages. A link could represent a tagged web-page in a navigation view. To consider different usecases, options could be offered to append only certain types of resources.

3.2.2 Multifaceted tag-based navigation

Multifaceted tag-based navigation extends the native tag-based navigation (see Chapter 3.2.1) concept. The name group by navigation is a synonym for multifaceted based navigation. As input the TACKO data model is used similar to the native navigation concept and the hierarchical source file system is also similar (Figure 3.2).

Two folders, `2011` and `2012` are contained in several directories. This is some kind of an ordering problem, which category is more general `bayern` and `niedersachsen` or `2011` and `2012`. In general this issue is not solvable. It is possible to change the order (Figure 3.7(b)) but after reordering the issue still exists. Multifaceted tag-based navigation can solve this issue. The sample shows there are two competing categories, year and regions, for the context `projects`. Categories are represented by facets. Each Context can contain many facets. Competing categories are named orthogonal facets.



```
projects
  bayern
    augsburg
      2011
      2012
    münchen
      2011
      2012
  niedersachsen
    braunschweig
      2011
      2012
```

(a) source

```
projects
  2011
    bayern
      augsburg
      münchen
    niedersachsen
      braunschweig
  2012
    bayern
      augsburg
      münchen
    niedersachsen
      braunschweig
```

(b) reordered source

Figure 3.7: Reordered hierarchical source file system.

In the sample, the context `projects` contains a facet {`bayern, niedersachsen`} and a second facet {`2011,2012`} based on the hierarchical source files system (Figure 3.2). The first facet is the native one that means this facet exists as native folders in the context path. Tags of the second facet are existing in multiple sources (Figure 3.8). In the hierarchical source file system there are three physical paths are related to the synthetic facet tag `2011`. That means all documents of these directories are represented by the tag `2011`. The tag `2012` exists also in multiple physical paths.

Context $\rightarrow$ {**projects**}
Facet$_1$ $\rightarrow$ {bayern, niedersachsen}^{native}

```
              bayern→  \projects\
       niedersachsen→  \projects\
```

Facet$_2$ $\rightarrow$ {2011, 2012}

```
                2011→  \projects\bayern\augsburg\
                       \projects\bayern\münchen\
                       \projects\niedersachsen\braunschweig\
                2012→  \projects\bayern\augsburg\
                       \projects\bayern\münchen\
                       \projects\niedersachsen\braunschweig\
```

Figure 3.8: Facettag path mapping.

Wherever documents are placed physically, the multifaceted navigation concept makes it transparent for the user and provides a facet-based navigation. Figure 3.9 illustrates the hierarchical source file system (Figure 3.2) sample as simplified multifaceted view. The context projects contains two folders beginning with the name `group by`. These are synthetic folders, which do not represent a single tag. Each `group by` folder represents one facet. All facet tags are appended to the `group by` prefix, separated with a comma.

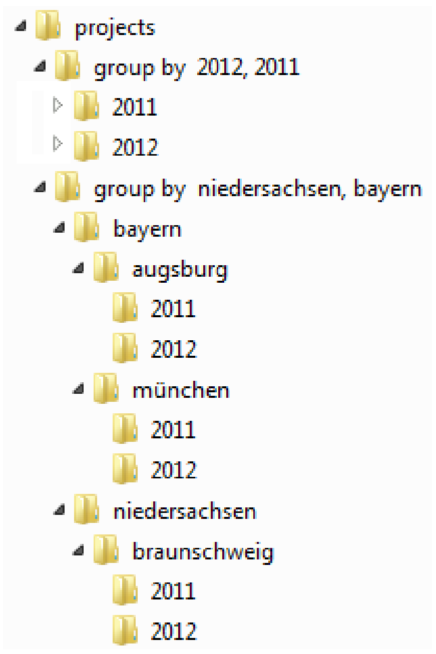
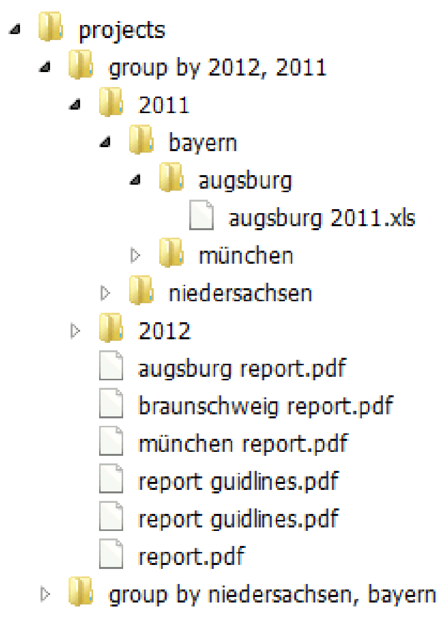


Figure 3.9: Simple multifaceted view without inherited facets.

By clicking on them the context does not change, only another representation for this context is selected. E.g. the current context is {`projects`}, clicking on the `group by 2011, 2012` folder changes the path from `\projects\` to `\projects\group by 2011, 2012\` and the context is still {`projects`}. The name prefix {`group by`} is reserved for selecting a context facet. Tags with this prefix are not allowed. Clicking on a facet tag within the `group by` folder, changes the context like expected. E.g. the current path is `\projects\group by 2011, 2012\`, clicking on 2012 changes the current context from {`projects`} to {`projects, 2012`}.

Figure 3.10 represents a hierarchical source file system and all corresponding tags for each path. This sample illustrates clearly how the document placement works in the multifaceted navigation concept. E.g. all documents within the tag path `\projects\group by 2011, 2012\` are tagged with `projects` and not with `2011` and `2012`.



(a) sample.	(b) corresponding sample tags.
projects	{projects}
group by 2012, 2011	{projects}
2011	{projects, 2011}
bayern	{projects, 2011, bayern}
augsburg	{projects, 2011, bayern, augsburg}
augsburg 2011.xls	{projects, 2011, bayern, augsburg}
münchen	{projects, 2011, bayern, münchen}
niedersachsen	{projects, 2011, niedersachsen}
2012	{projects, 2012}
augsburg report.pdf	{projects, ~~2011~~, ~~2012~~}
braunschweig report.pdf	{projects, ~~2011~~, ~~2012~~}
münchen report.pdf	{projects, ~~2011~~, ~~2012~~}
report guidlines.pdf	{projects, ~~2011~~, ~~2012~~}
report guidlines.pdf	{projects, ~~2011~~, ~~2012~~}
report.pdf	{projects, ~~2011~~, ~~2012~~}
group by niedersachsen, bayern	{projects}

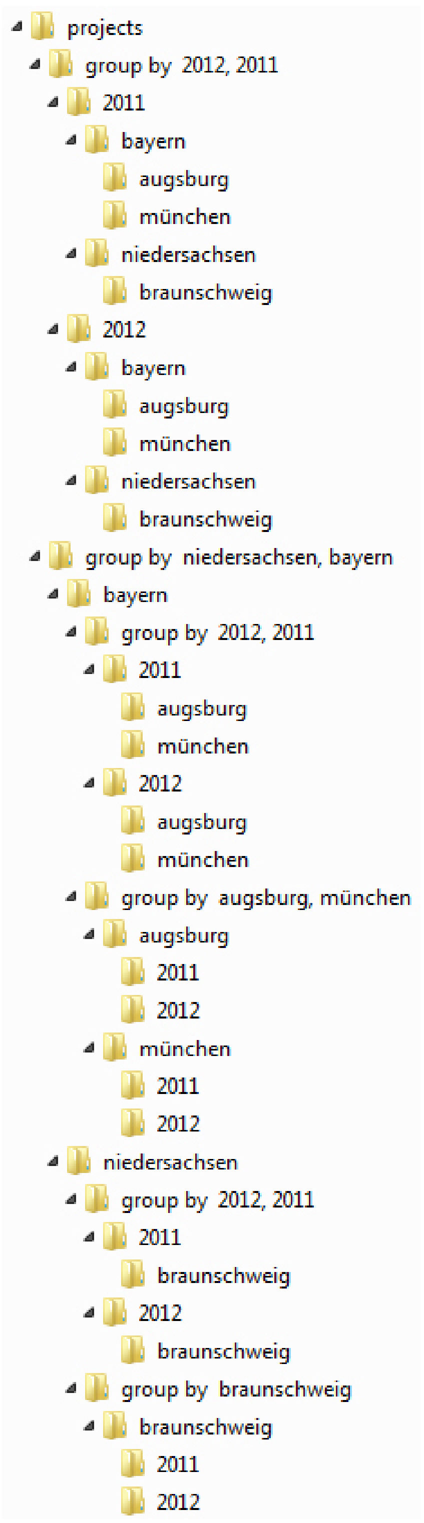
Figure 3.10: Document placement sample.

Facet based navigation is an advanced navigation concept. For one resource several navigation paths are possible. The simplified sample (Figure 3.9) does not consider inherited facets. Inherited facets are explained in Chapter refsec:basicTacko). The real navigation example is illustrated in Figure 3.12(a). E.g. searching for all documents which are tagged with `münchen` . Physically it exists only once in the path `\projects\bayern\`, the sample (Figure 3.11) illustrates all possible navigation paths.

```
münchen→   \projects\group by 2012, 2011\2011\bayern\
           \projects\group by 2012, 2011\2012\bayern\
           \projects\group by niedersachsen, bayern\bayern\group by 2012, 2011\2011\
           \projects\group by niedersachsen, bayern\bayern\group by 2012, 2011\2012\
           \projects\group by niedersachsen, bayern\bayern\group by augsburg, münchen\
```

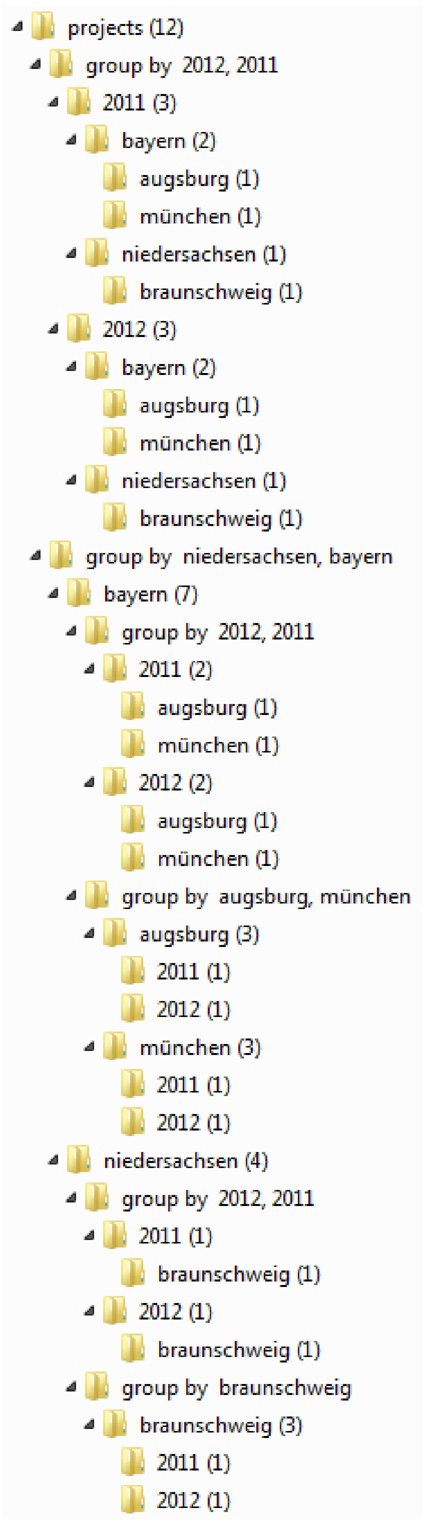
Figure 3.11: All possible navigation paths for the tag müchen.

The multifaceted tag-based navigation concept provides also a count option, similar to the native tag-based count option (Figure 3.12(b)). `group by` folders are an exception and have no count postfix. The parent path tags are counted, a `group by` folder only changes the representation but not the context. Only the navigation concept is influenced but not the amount of resources. The count is always equal to the count of the parent tag.



```
projects
  group by 2012, 2011
    2011
      bayern
        augsburg
        münchen
      niedersachsen
        braunschweig
    2012
      bayern
        augsburg
        münchen
      niedersachsen
        braunschweig
  group by niedersachsen, bayern
    bayern
      group by 2012, 2011
        2011
          augsburg
          münchen
        2012
          augsburg
          münchen
      group by augsburg, münchen
        augsburg
          2011
          2012
        münchen
          2011
          2012
    niedersachsen
      group by 2012, 2011
        2011
          braunschweig
        2012
          braunschweig
      group by braunschweig
        braunschweig
          2011
          2012
```

(a) default



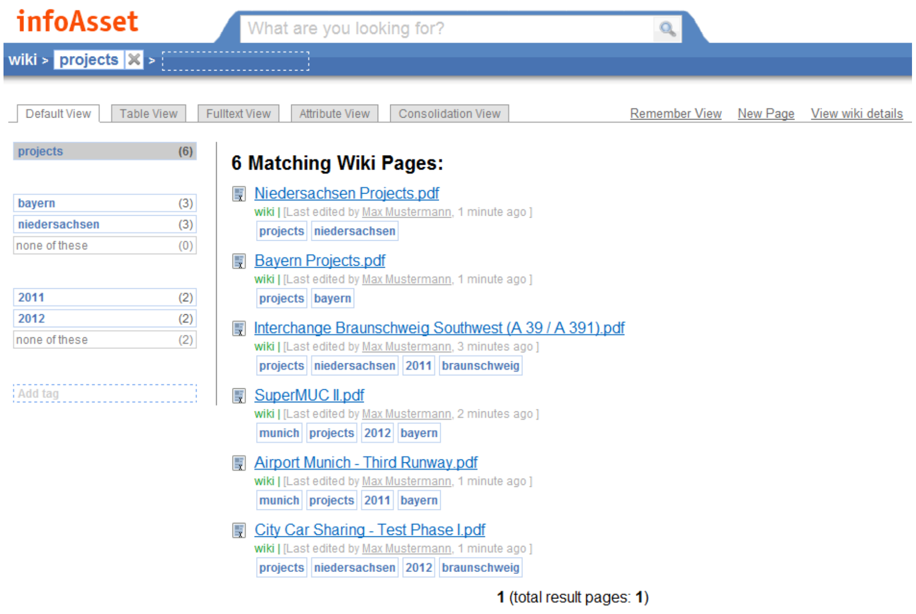
```
projects (12)
  group by 2012, 2011
    2011 (3)
      bayern (2)
        augsburg (1)
        münchen (1)
      niedersachsen (1)
        braunschweig (1)
    2012 (3)
      bayern (2)
        augsburg (1)
        münchen (1)
      niedersachsen (1)
        braunschweig (1)
  group by niedersachsen, bayern
    bayern (7)
      group by 2012, 2011
        2011 (2)
          augsburg (1)
          münchen (1)
        2012 (2)
          augsburg (1)
          münchen (1)
      group by augsburg, münchen
        augsburg (3)
          2011 (1)
          2012 (1)
        münchen (3)
          2011 (1)
          2012 (1)
    niedersachsen (4)
      group by 2012, 2011
        2011 (1)
          braunschweig (1)
        2012 (1)
          braunschweig (1)
      group by braunschweig
        braunschweig (3)
          2011 (1)
          2012 (1)
```

(b) with count option

Figure 3.12: Multifaceted tag-based view via SMB.

3.2.3 Multifaceted tag-based navigation with TACKO

Additionally to the navigation concepts described in Chapter 3.2.1 and 3.2.2 , Figure 3.13 illustrates how the navigation with TACKO in combination with the documents imported from the hierarchical file system could look like. The basics of this navigation are explained in Chapter 2.1.3. The concept is strongly related to the TACKO Files multifaceted tag-based navigation. Basically the hierarchical file system is the same as in the previous sample (Figure 3.2). All screens are separated in three different areas. On top is a blue navigation bar with the tag path which expresses the current context. Selected facet-tags and the facets of the current context are visualized on the left. Facets with a gray background are in the context. Each facet of the current context is represented as own block of facet-tags. Centered, the context corresponding documents are listed. Subfigure 3.13(b) represents the context, {`projects, bayern`}. The right Subfigure 3.13(c) illustrates the synthetic context {`projects, 2011`}.



infoAsset
What are you looking for?
wiki > projects >
Default View | Table View | Fulltext View | Attribute View | Consolidation View — Remember View | New Page | View wiki details

projects (6)

bayern (3)
niedersachsen (3)
none of these (0)

2011 (2)
2012 (2)
none of these (2)

Add tag

6 Matching Wiki Pages:

Niedersachsen Projects.pdf
wiki | [Last edited by Max Mustermann, 1 minute ago]
projects niedersachsen

Bayern Projects.pdf
wiki | [Last edited by Max Mustermann, 1 minute ago]
projects bayern

Interchange Braunschweig Southwest (A 39 / A 391).pdf
wiki | [Last edited by Max Mustermann, 3 minutes ago]
projects niedersachsen 2011 braunschweig

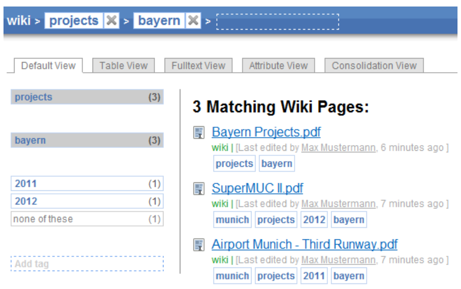
SuperMUC II.pdf
wiki | [Last edited by Max Mustermann, 2 minutes ago]
munich projects 2012 bayern

Airport Munich - Third Runway.pdf
wiki | [Last edited by Max Mustermann, 1 minute ago]
munich projects 2011 bayern

City Car Sharing - Test Phase I.pdf
wiki | [Last edited by Max Mustermann, 1 minute ago]
projects niedersachsen 2012 braunschweig

1 (total result pages: **1**)

(a) Tags: projects



wiki > projects > bayern >
Default View | Table View | Fulltext View | Attribute View | Consolidation View

projects (3)

bayern (3)

2011 (1)
2012 (1)
none of these (1)

Add tag

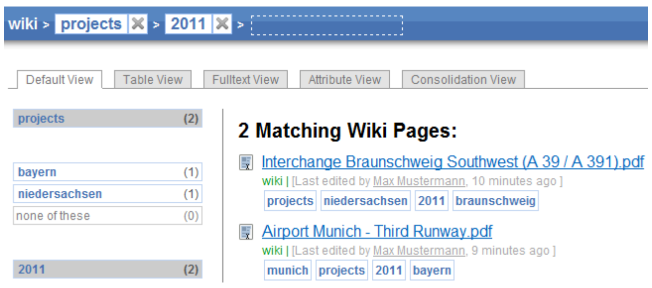
3 Matching Wiki Pages:

Bayern Projects.pdf
wiki | [Last edited by Max Mustermann, 6 minutes ago]
projects bayern

SuperMUC II.pdf
wiki | [Last edited by Max Mustermann, 7 minutes ago]
munich projects 2012 bayern

Airport Munich - Third Runway.pdf
wiki | [Last edited by Max Mustermann, 7 minutes ago]
munich projects 2011 bayern

(b) Tags: projects, bayern



wiki > projects > 2011 >
Default View | Table View | Fulltext View | Attribute View | Consolidation View

projects (2)

bayern (1)
niedersachsen (1)
none of these (0)

2011 (2)

2 Matching Wiki Pages:

Interchange Braunschweig Southwest (A 39 / A 391).pdf
wiki | [Last edited by Max Mustermann, 10 minutes ago]
projects niedersachsen 2011 braunschweig

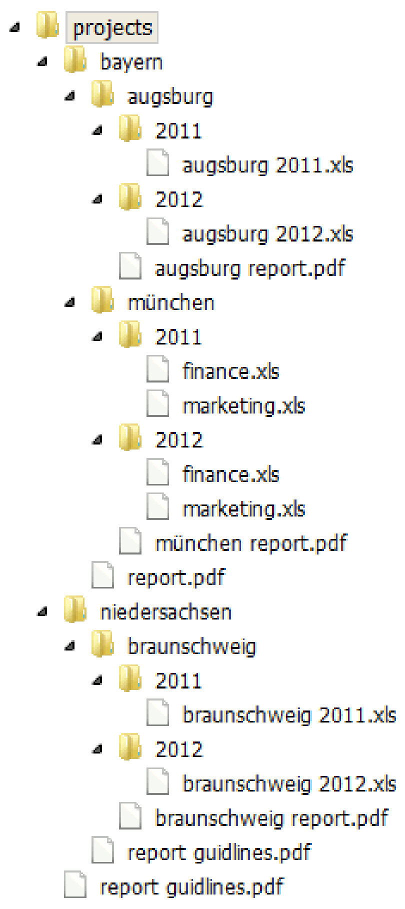
Airport Munich - Third Runway.pdf
wiki | [Last edited by Max Mustermann, 9 minutes ago]
munich projects 2011 bayern

(c) Tags: projects, 2011

Figure 3.13: Multifaceted tag-based navigation with TACKO.

3.3 CRUD Operations

This chapter covers all possible tag-based operations. For each navigation concept (see Chapter 3.2.1, 3.2.2), native tag-based, native tag-based with count option, multifaceted and multifaceted tag-based navigation with count option, there are four different operations possible, create, rename, move and delete. Each Operation is divided in a directory and document operation. This are $2_{views} * 2_{count\ option} * 4_{operations} * 2_{dir\ or\ doc} = 32$ possible operations. The cases can be simplified, the count option does not really affect the concept of operations ($2_{views} * 4_{operations} * 2_{dir\ or\ doc} = 16$). Before each operation, the count numbers are removed. From the user point of view there are only 8 different ones ($4_{operations} * 2_{dir\ or\ doc} = 8$). In the following subchapter these operations are explained with samples. Not all cases are covered but the basic ideas and most important ones are illustrated. All CRUD operation samples are based on the hierarchical source file system (Figure 3.2). Documents within the directories are not shown in the windows based view only the current selected directory. The prototypical hierarchical view represents also files in the directory tree. Figure 3.14 illustrates the hierarchical source file system example with its documents.



```
projects
  bayern
    augsburg
      2011
        augsburg 2011.xls
      2012
        augsburg 2012.xls
      augsburg report.pdf
    münchen
      2011
        finance.xls
        marketing.xls
      2012
        finance.xls
        marketing.xls
      münchen report.pdf
    report.pdf
  niedersachsen
    braunschweig
      2011
        braunschweig 2011.xls
      2012
        braunschweig 2012.xls
      braunschweig report.pdf
    report guidlines.pdf
  report guidlines.pdf
```

Figure 3.14: CRUD operations hierarchical source file system via web-interface.

All operations are accessible via a right click on the tag or document in the tag-based navigation tree. Depending on the selected item, several actions are visible in the context menu. Clicking on the action item initializes the operation and shows a dialog. Figure 3.15(b) illustrates a create new document dialog. The dialog concept uses common dialog patterns. On top of each dialog is a status bar which contains the operation name, on the left an icon corresponding to the operation and on the right a cross which triggers the close event. The dialog question is placed on top of the dialog content. In dependence of the selected operation, different interactions are possible. Warnings and errors are placed in a red box below the dialog question. In case of an error, the execution button on the bottom is disabled.

This is a prototypically implementation and the aim of these crud operations is to get a better understanding how the tag-based mapping works. Each operation is mapped to the hierarchical source file system. All dialogs provide the execution information tag-based and based on the mapped hierarchical file system. In a productive system would be better to provide only tag-based information and show some extra mapping information when an error occurs to improve the usability. This is a trade-of decision, in a prototypically system it is more important to understand the concepts. Therefore all mapping information is provided in the dialogs.

According to the Usability Engineering reference book [Rud06][p.54], similar tasks should be handled in a similarly way. Therefore, each dialog uses the same notation to illustrate the operation corresponding information. Under the question the context tags are placed in gray. All tags which are excluded are crossed out. Below, the associated hierarchical file system path is presented. In some cases the path does physically not exist in the hierarchical source files system (Figure 3.17(b)). In this case the path represents the path which is created if the operation is executed. This kind of mapping problems does only existing in combination with the multifaceted navigation concept. Documents or tags which are affected by the operation marked in blue.

The following subchapter illustrates the different kinds of operations. Most of the operations are represented with special scenarios which illustrates the different issues. An average user is normally not able to cause so many special cases. Each chapter is structured by native tag-based operations and multifaceted tag-based operations. A document and one tag operation are illustrated. Each sample contains at least a tag-based tree navigation picture with the context menu and a dialog.

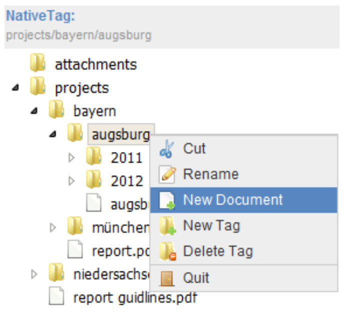
3.3.1 Create Operation

A create operation offers the possibility to add a new tag or a document. Each create operation dialog has an input field which represents a new tag or document name. In case of this prototypical implementation it is only possible to create a dummy file with the input name. The real file handling is only some kind of an implementation issue. On dialog initialization, the tag context is known a corresponding hierarchical path is also known. All in the existing directory names and document names in this hierarchical path are not allowed as new name. The input field is evaluated, triggered by an on key event. If a name contains a special char or an existing name, a warning message is shown, the input field is marked with a red border and the create button is deactivated.

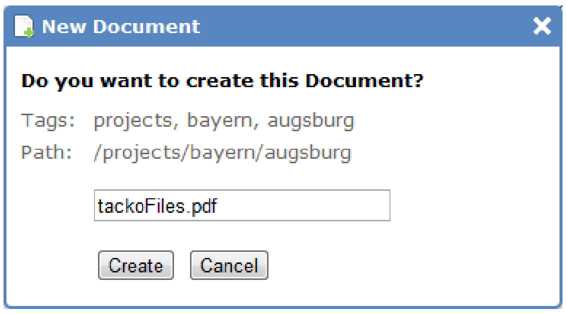
3.3.1.1 Native tag-based create operations

Native tag-based create new document operation

Figure 3.15 illustrates a basic use case, create new document. Right click on the tag path `\projects\bayern\augsburg\` opens a context menu. Several options are offered, a left click on `New Document` opens the corresponding dialog. The label path in the dialog show the hierarchical path corresponding to the tags. In this case they are directly assigned. The new filename is `tackoFiles.pdf`.



(a) create context menu



Do you want to create this Document?

Tags: projects, bayern, augsburg
Path: /projects/bayern/augsburg

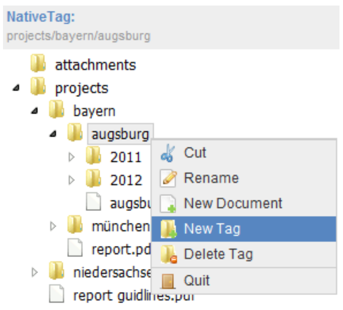
tackoFiles.pdf

(b) create dialog

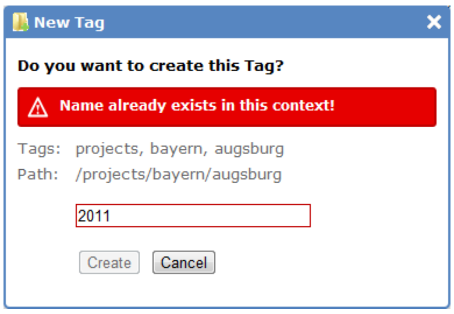
Figure 3.15: Native tag-based create new document operation.

Native tag-based create new tag operation

The second native tag-based sample illustrates a new directory create operation (Figure 3.16). The red box contains an error `Name already exist in this context!`. On the native tag-based navigation tree, it is visible that in this context already a tag named `2011` exists.



(a) create context menu



Do you want to create this Tag?

Name already exists in this context!

Tags: projects, bayern, augsburg
Path: /projects/bayern/augsburg

2011

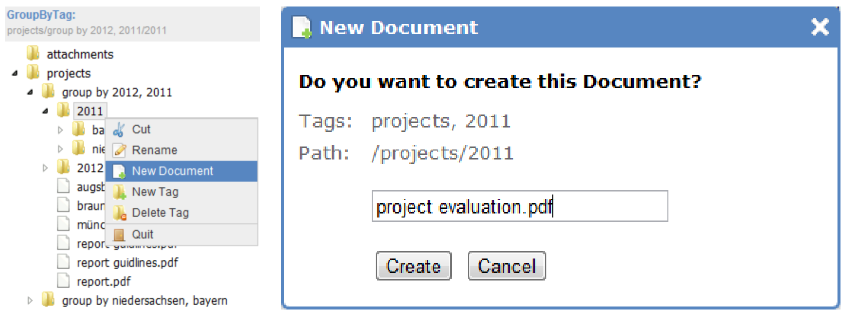
(b) create dialog

Figure 3.16: Native tag-based create new tag operation.

3.3.1.2 Multifaceted tag-based create new document operations

Multifaceted tag-based create new document operation

Figure 3.17 illustrates a multifaceted tag-based create new document operation. The operation context is {`projects, 2011`} and the corresponding path is \\`projects`\\`group by 2011,2012`\\`2011`\\. This facet {`2011, 2012`} represented with the group by is a synthetic one. This means the selected path does physically not exist in the hierarchical source file system (Figure 3.14). Create the document `project evaluation.pdf` means to create a physical related directory and then the document. The new created directory named `2011` represents the path \\`projects`\\`2011`\\. In general a new folder in the hierarchical source file system can affect the facet finding algorithm. In this case, the facet tag `2011` is now a native. For the facet tag `2012` no directly related physical path exists, that means this tag is sill synthetic in this context. Therefore the facet {`2011, 2012`} is also still a synthetic facet.



Do you want to create this Document?

Tags: projects, 2011

Path: /projects/2011

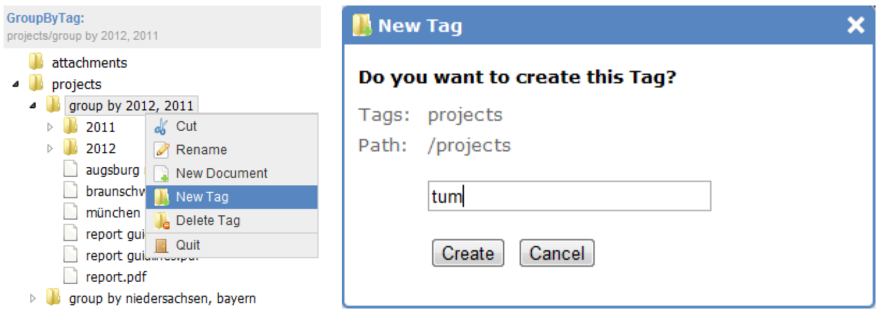
project evaluation.pdf

(a) create context menu (b) create dialog

Figure 3.17: Multifaceted tag-based create new document operation.

Multifaceted tag-based create new tag operation

The tag-based create new document operation in Figure 3.18 represents another special case. The context menu is selected on the path `\projects\group by 2011, 2012\`. According to chapter 3.2.2, a `group by` folder does not affect the context. The operation is executed in the context `{projects}`.It is also conceivable to prevent operations on all `group by` folders. This user interface design allows to perform operations, there are two different reasons. The navigation concept defines that a `group by` folder only change the representation. Moreover the aim is to integrate this navigation concept into the mountable network device. There it is not possible to modify the context menu in a meaningful way.



Do you want to create this Tag?

Tags: projects
Path: /projects

tum

(a) create context menu (b) create dialog

Figure 3.18: Multifaceted tag-based create new tag operation.

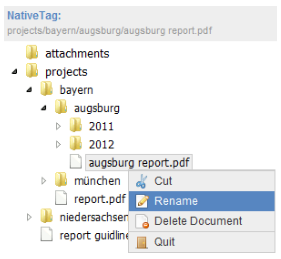
3.3.2 Rename Operation

Rename operations rename tags or documents, the name input validation is similar to the create operation (see Chapter 3.3.1).

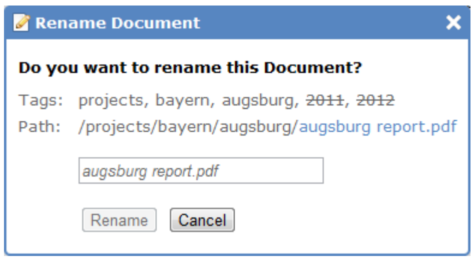
3.3.2.1 Native tag-based rename operations

Native tag-based rename document operation

Figure 3.19 illustrates a native tag-based rename document operation. This dialog shows the initial state, the rename input field is initialized with the old name in italic letters. The first on click event removes it and after changing a letter, the rename button is enabled. Path sections which are affected by this operation are marked blue. `2011` and `2012` are not path tags (see Chapter 3.2.1).



(a) rename context menu



Rename Document

Do you want to rename this Document?

Tags: projects, bayern, augsburg, ~~2011~~, ~~2012~~

Path: /projects/bayern/augsburg/augsburg report.pdf

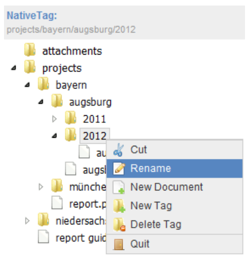
augsburg report.pdf

(b) rename dialog

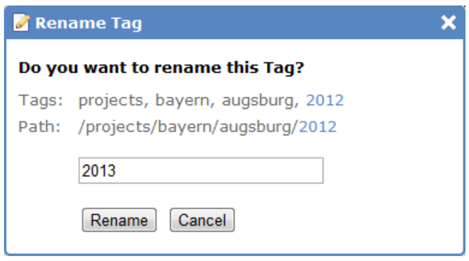
Figure 3.19: Native tag-based rename document operation.

Native tag-based rename tag operation

This is a simple tag-based rename tag operation. The context {`projects, bayern, augsburg`} are directly corresponding to the path `\projects\bayern\augsburg\` of the hierarchical source file system. This operation renames the tag `2012` into `2013`. Other subdirectories in the hierarchical source files system contains still the folders `2011` and `2012`. Therefore, the original facet is extended and results in {`2011, 2012, 2013`}.



(a) rename context menu



Rename Tag

Do you want to rename this Tag?

Tags: projects, bayern, augsburg, 2012

Path: /projects/bayern/augsburg/2012

2013

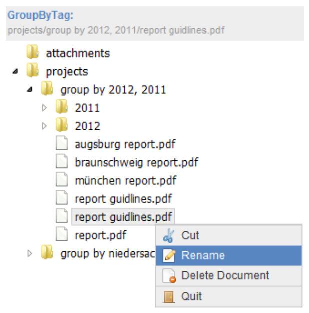
(b) rename dialog

Figure 3.20: Native tag-based rename tag operation.

3.3.2.2 Multifaceted tag-based rename operations

Multifaceted tag-based rename document operation

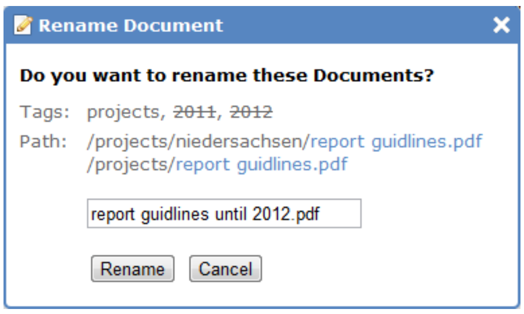
This paragraph illustrates a multifaceted rename document operation. The context {`projects`} contains two files with an identical name and different paths in the hierarchical source file system (Figure 3.14)). Rename `report guidliness.pdf` in this context means to rename the file `\projects\niedersachsen\report guidlines.pdf\` and also `\projects\report guidlines.pdf\`. The new renamed file name is `report guidelines until 2012.pdf`. Causal for this case is the synthetic representation of this context with the facet {`2011, 2012`}.



GroupByTag:
projects/group by 2012, 2011/report guidlines.pdf

```
attachments
projects
  group by 2012, 2011
    2011
    2012
    augsburg report.pdf
    braunschweig report.pdf
    münchen report.pdf
    report guidlines.pdf
    report guidlines.pdf
    report.pdf
  group by niedersac
```

(a) rename context menu



Do you want to rename these Documents?

Tags: projects, ~~2011~~, ~~2012~~

Path: /projects/niedersachsen/report guidlines.pdf
/projects/report guidlines.pdf

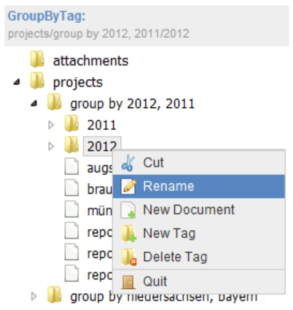
report guidlines until 2012.pdf

(b) rename dialog

Figure 3.21: Multifaceted tag-based rename document operation.

Multifaceted tag-based rename tag operation

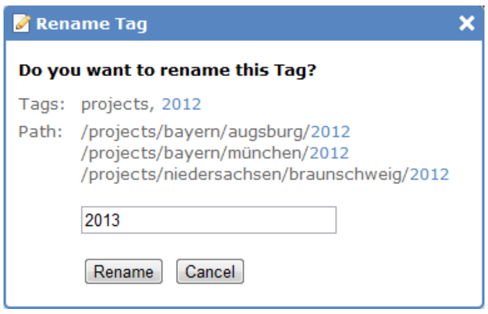
Figure 3.22 shows a problem similar to the multifaceted based document rename operation (see Chapter 3.3.2.2). Rename the tag `2012` with this synthetic context representation means to rename the folder `2012` on several paths in the hierarchical source file system (Figure 3.14)). In this case this is a desirable behavior.



GroupByTag:
projects/group by 2012, 2011/2012

```
attachments
projects
  group by 2012, 2011
    2011
    2012
    augs
    brau
    mün
    repo
    repo
    repo
  group by niedersachsen, bayern
```

(a) rename context menu



Do you want to rename this Tag?

Tags: projects, 2012

Path: /projects/bayern/augsburg/2012
/projects/bayern/münchen/2012
/projects/niedersachsen/braunschweig/2012

2013

(b) rename dialog

Figure 3.22: Multifaceted tag-based rename tag operation.

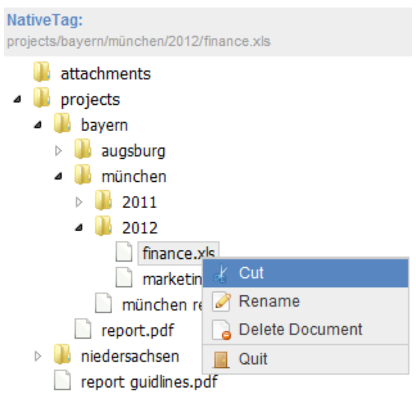
3.3.3 Move Operation

The concept of the move operations differs a bit from all others. In the first step, a source tag or document is put onto the clipboard by clicking cut in the context menu. When the clipboard contains a source and the user clicks on some tag the option paste is visible in the context menu.

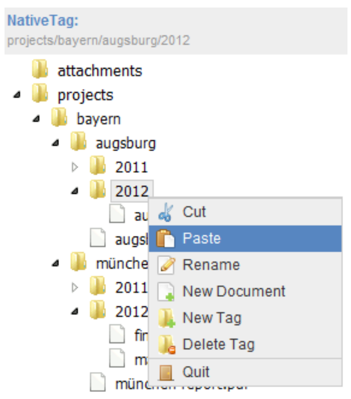
3.3.3.1 Native tag-based move operation

Native tag-based move document operation

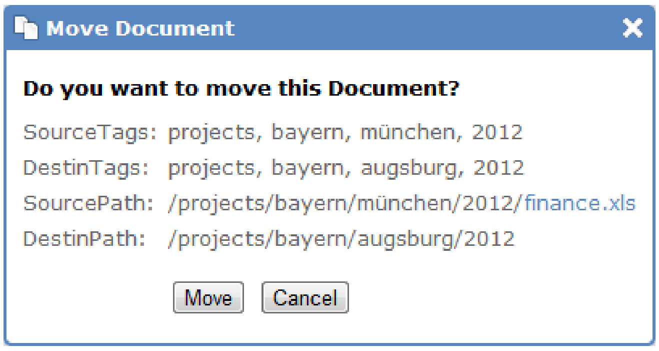
Figure 3.23 illustrates a simple native tag-based move document operation. First of all the source target is selected (Figure 3.23(a)) by clicking on the cut option in the context menu. The document with the name `finance.xls` in the context {`projects, bayern, münchen, 2012`} is chosen. In the second step a target destination is defined by clicking on the tag `2012` in the tag path `\projects\bayern\augsburg\2012\`. Corresponding to this move operation, the dialog shows the tag-based source and destination and also the assigned paths in the hierarchical source file system (Figure 3.14).



(a) cut context menu



(b) paste context menu



Move Document

Do you want to move this Document?

SourceTags: projects, bayern, münchen, 2012
DestinTags: projects, bayern, augsburg, 2012
SourcePath: /projects/bayern/münchen/2012/finance.xls
DestinPath: /projects/bayern/augsburg/2012

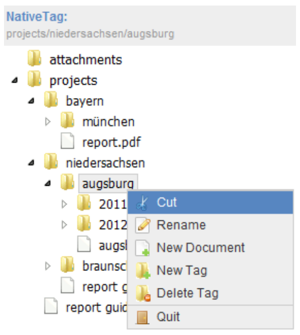
Move Cancel

(c) move dialog

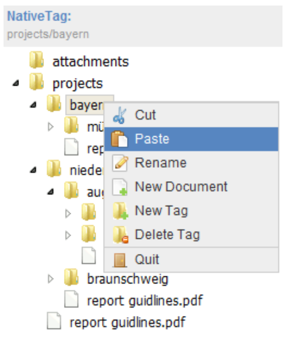
Figure 3.23: Native tag-based move document operation.

Native tag-based move tag operation

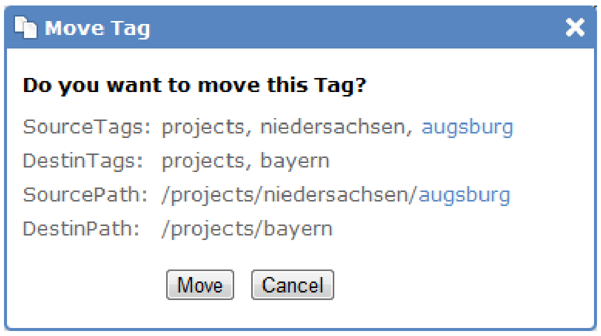
This sample illustrates a simple tag-based move tag operation. Augsburg is tagged with `projects` and `niedersachsen`, this needs to be corrected. The tag `augsburg` in the context {`projects, niedersachsen`} is put to the clipboard as source (Figure 3.24(a)). Selecting the tag `bayern` in the context {`projects`} and clicking paste defines the destination (Figure 3.24(b)). All relevant move tag information is shown in the dialog.



(a) cut context menu



(b) paste context menu



Do you want to move this Tag?

SourceTags: projects, niedersachsen, augsburg
DestinTags: projects, bayern
SourcePath: /projects/niedersachsen/augsburg
DestinPath: /projects/bayern

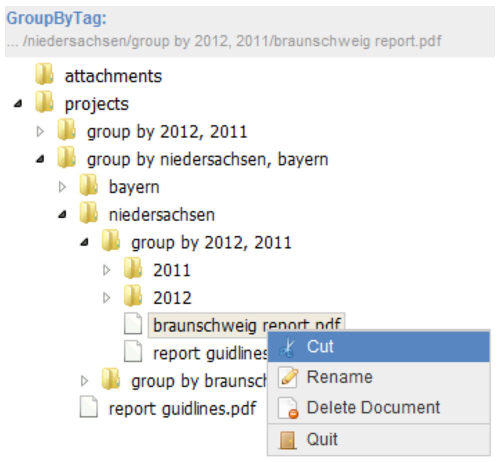
(c) move dialog

Figure 3.24: Native tag-based move tag operation.

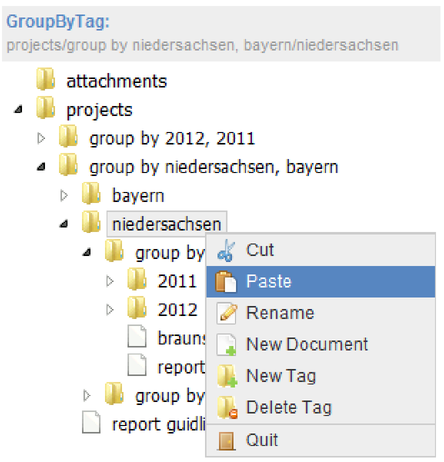
3.3.3.2 Multifaceted tag-based move operations

Multifaceted tag-based move document operation

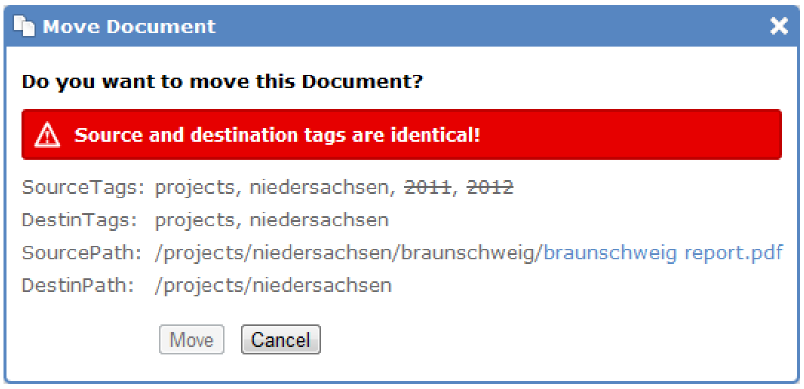
A multifaceted tag-based move document operation is illustrated in this sample (Figure 3.25). The selected source target is the document with the name `braunschweig report.pdf` in the tag path `\projects\group by niedersachsen, bayern\niedersachsen\group by 2011, 2012\`. As destination the tag-based path `\projects\group by niedersachsen, bayern\niedersachsen\` is chosen. The dialog shows an error message `Source and destination tags are identical!`. The multifaceted navigation concept is also named group by navigation explained in chapter 3.2.2. A `group by` folder does only change the representation and does not affect the context which is the reason for this error.



(a) cut context menu



(b) paste context menu


Move Document

Do you want to move this Document?

Source and destination tags are identical!

SourceTags: projects, niedersachsen, ~~2011~~, ~~2012~~
DestinTags: projects, niedersachsen
SourcePath: /projects/niedersachsen/braunschweig/braunschweig report.pdf
DestinPath: /projects/niedersachsen

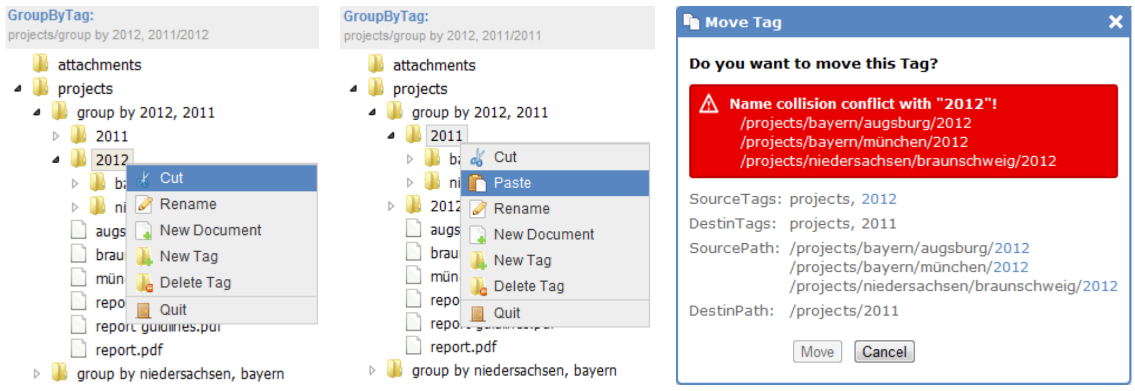
Move Cancel

(c) move dialog

Figure 3.25: Multifaceted tag-based move document operation.

Multifaceted tag-based move tag operation

The tag-based multifaceted move tag operation sample (Figure 3.26) illustrates a new problem. `2011` and `2012` represent a synthetic facet in the context {`projects`}. This operation tries to move the tag `2012` inside the tag `2011`. With native facets it is a simple move operation but in this case name conflicts exists. The error message in the dialog shows hierarchical file system based source path of the folders. The synthetic facet tag `2012` represents all these paths abstract as a tag. So far really helpful. Moving this tag `2012` tries to move all related paths to the destination, this is not possible because the folder name `2012` would no more represent a unique name in the hierarchical source file system. In the current prototypical implementation this operation is not executable. In a visionary scenario all different source file system folders could be merged. For folders which represent tags this is easy solvable merge all equal named folders. Merging folders come up with a new problem, filenames within these folders must be unique. One pragmatic solution could be to append a certain char to each not unique filename.



Do you want to move this Tag?

Name collision conflict with "2012"!
/projects/bayern/augsburg/2012
/projects/bayern/münchen/2012
/projects/niedersachsen/braunschweig/2012

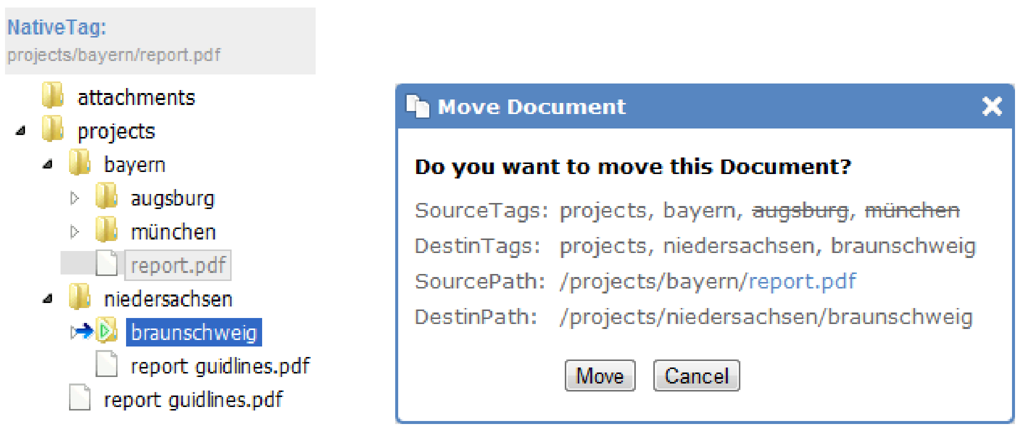
SourceTags: projects, 2012
DestinTags: projects, 2011
SourcePath: /projects/bayern/augsburg/2012
/projects/bayern/münchen/2012
/projects/niedersachsen/braunschweig/2012
DestinPath: /projects/2011

(a) cut context menu (b) paste context menu (c) move dialog

Figure 3.26: Multifaceted tag-based move tag operation.

3.3.3.3 Drag and drop move operation

All described move operations can also be initialized with a drag and drop operation (Figure 3.27). Moving the source target with pressed left mouse button over the target, after a short time the destination tag is marked blue, on left mouse button up the destination is selected (Figure 3.27(a)). When the mouse pointer is moved on not expanded tags, they will expand with a delay of one second. This delay is important for the usability. Otherwise all tags where you move over would be expanded and cause a confusing behavior. After the destination target is selected, the behavior is exactly the same as the default move operation initialization with the context menu. A dialog shows all important operation information (Figure 3.27(b)).



Do you want to move this Document?

SourceTags: projects, bayern, ~~augsburg~~, ~~münchen~~
DestinTags: projects, niedersachsen, braunschweig
SourcePath: /projects/bayern/report.pdf
DestinPath: /projects/niedersachsen/braunschweig

(a) cut context menu (b) paste context menu

Figure 3.27: Drag and drop move operation.

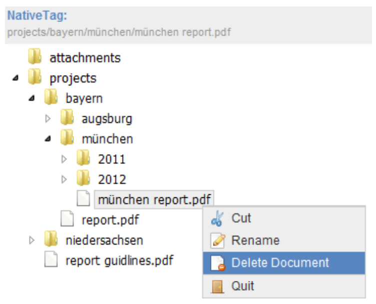
3.3.4 Delete Operation

Delete operations delete a document or a tag. In the case of a tag, all its sub tags and corresponding documents are deleted.

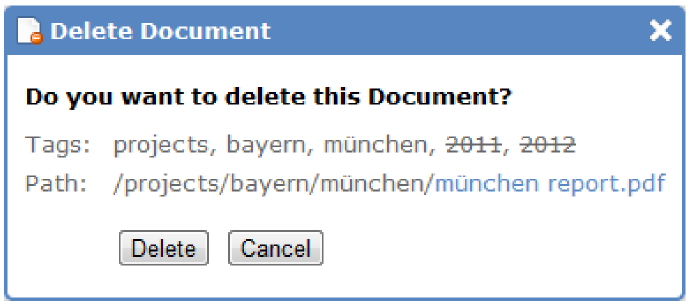
3.3.4.1 Native tag-based delete operation

Native tag-based delete document operation

Figure 3.28 illustrates a native tag-based delete document operation. This sample deletes the document `münchen report.pdf` in the context {`projects, bayern, augsburg`}. The tags `2011` and `2012` are not path tags. In this case it is trivial and the document in the hierarchical source file system with the path `\projects\bayern\münchen\münchen report.pdf` is deleted.



(a) delete context menu


Delete Document

Do you want to delete this Document?

Tags: projects, bayern, münchen, ~~2011~~, ~~2012~~

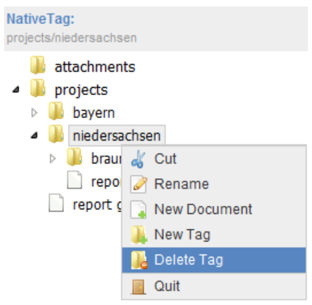
Path: /projects/bayern/münchen/münchen report.pdf

(b) rename dialog

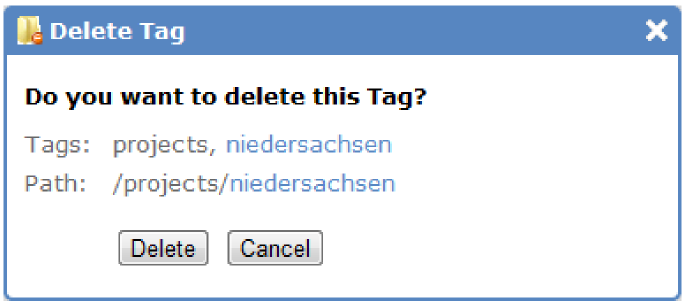
Figure 3.28: Native tag-based delete document operation.

Native tag-based delete tag operation

This samples illustrates a native tag-based delete tag operation. Context of this delete tag operation is {`projects`}. The tag `niedersachsen` and all its sub tags are deleted. The hierarchical file system mapping it unique, the operation is executable.



(a) delete context menu


Delete Tag

Do you want to delete this Tag?

Tags: projects, niedersachsen

Path: /projects/niedersachsen

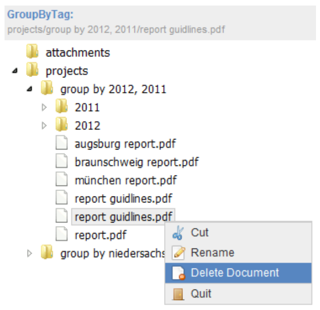
(b) rename dialog

Figure 3.29: Native tag-based delete tag operation.

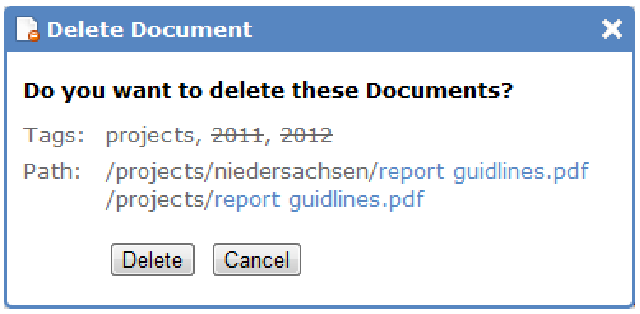
3.3.4.2 Multifaceted tag-based delete operations

Multifaceted tag-based delete document operation

Figure 3.30 illustrates a multifaceted tag-based delete document operation. The context {projects} is represented by a synthetic facet {`2011, 2012`}. Several documents are tagged with `projects` but not with `2011` or `2012`. All these documents are located in the `group by 2011, 2012` folder as direct children. This operation deletes all documents named `report guidlines.pdf` in the described context. The dialog shows the two affected files in the hierarchical file system.



(a) delete context menu



Do you want to delete these Documents?

Tags: projects, ~~2011~~, ~~2012~~

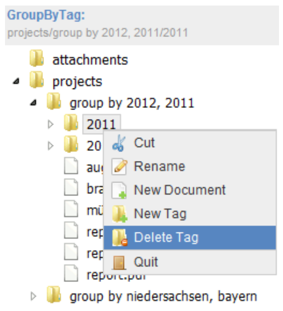
Path: /projects/niedersachsen/report guidlines.pdf
/projects/report guidlines.pdf

(b) delete dialog

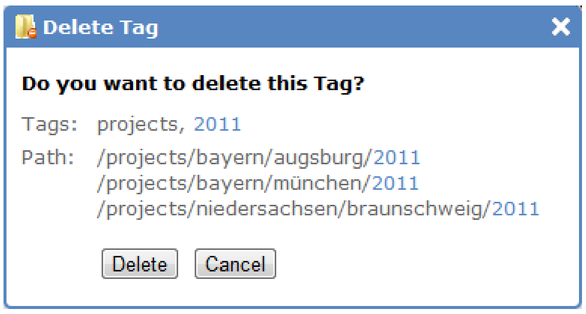
Figure 3.30: Multifaceted tag-based delete document operation.

Multifaceted tag-based delete tag operation

Finally there is one more delete operation, the multifaceted tag-based delete operation (Figure 3.31). This operation deletes the tag `2011` in the context {`projects`}, represented by a synthetic facet {`2011, 2012`}. The corresponding folder `2012` exists more than once in the hierarchical file system, illustrated on the dialog (Figure 3.31). All these directories are affected.



(a) delete context menu



Do you want to delete this Tag?

Tags: projects, 2011

Path: /projects/bayern/augsburg/2011
/projects/bayern/münchen/2011
/projects/niedersachsen/braunschweig/2011

(b) delete dialog

Figure 3.31: Multifaceted tag-based delete tag operation.

3.4 TACKO Files Web Interface

The TACKO Files web interface provides all navigation concepts. Additionally importing testdata, migrating tags and a visualized facet import is offered. Figure 3.32 illustrates the TACKO Files plugin integrated in tricia. On top there is the blue tricia search bar. A navigation menu on the left provides navigating within the plugin. The content of every menu item is represented on the right in the centered main component.

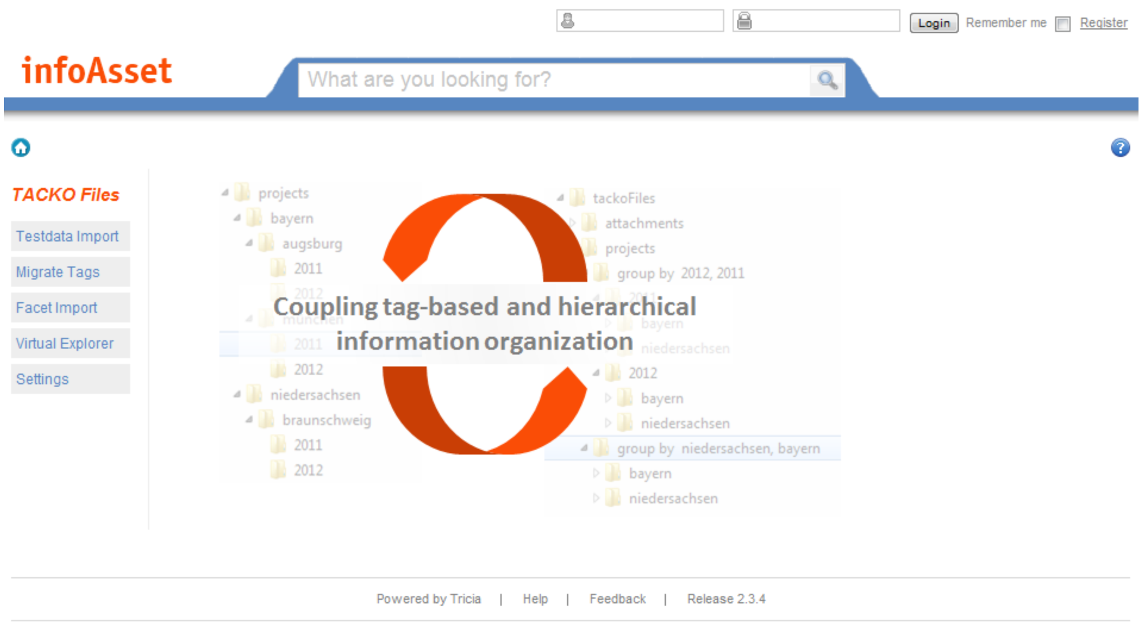
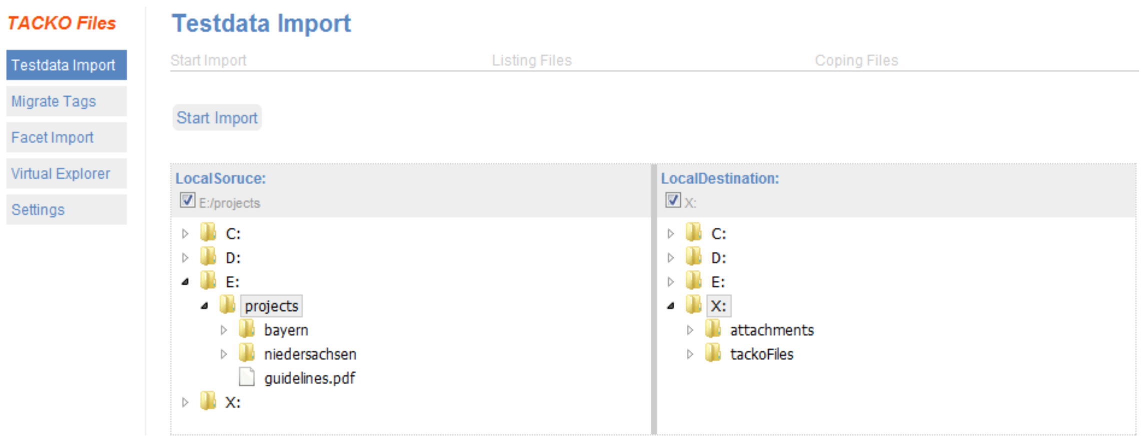
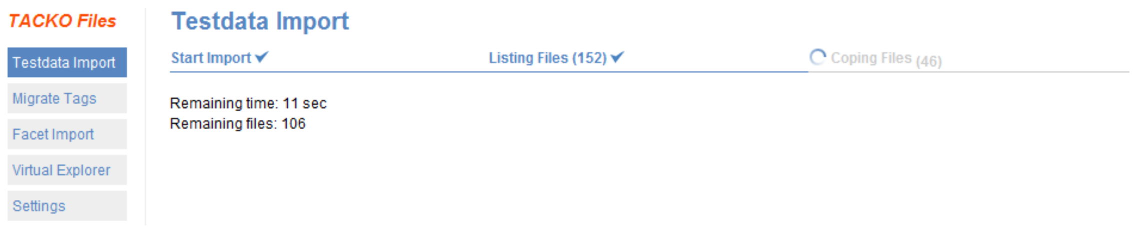

Login
Remember me
Register
infoAsset
What are you looking for?
TACKO Files
Testdata Import
Migrate Tags
Facet Import
Virtual Explorer
Settings
Coupling tag-based and hierarchical information organization
Powered by Tricia | Help | Feedback | Release 2.3.4

Figure 3.32: TACKO Files integrated in tricia.

The testdata import is the first menu item (Figure 3.33). This interface provides importing test data. Local folder and file structures can be imported easily as folders and dummy files. That means all selected folders within the structure are copied one to one. All files within the structure are only copied as empty files with the original name. Copying the files only as dummy files improves the performance massively and does not affect the algorithms. This allows fast initialisation of real data structures for testing. Hint, this user interface is only useful for local testing. All data which is accessible via web interface is the local data of the server operating system. The screen illustrates the initial testdata import state. Centered there are two local file system explorers, the left one represents the source and the right one the destination. In the first step the local source folder is selected. A folder on the mountable network device should be always the destination. After source and destination folders are defined by browsing, the paths must be confirmed. On the top of the explorers, a gray bar shows the current selected local file system path. With the check box before this path the selection can be confirmed. This confirm is necessary, otherwise it is possible to override already existing local files. If a path is changed, the check box is unchecked automatically to prevent undesired behavior. The button `Start Import` triggers the import process. Below the headline a state bar represent the current state of the import. During the import the bar (Figure 3.33(b)) visualizes the unfinished steps in gray, all steps in progress are represented with a calculation symbol. The finished steps are chopped off and they are colored in blue. Additionally all processed files are counted and placed in brackets behind the state name. The left files are represented with the remaining files counter and the corresponding left time is estimated.



(a) testdata import initializing



Remaining time: 11 sec
Remaining files: 106

(b) testdata import

Figure 3.33: TACKO Files testdata import.

One of the TACKO Files core components is the facet import algorithm. Figure 3.34 illustrates the facet import user interface after a facet import. This interface does not affect the internal persisted facets, it is only to illustrate the facet finding algorithm for certain hierarchical folder structures. Aim is to illustrate small samples understandable. Importing facets is based on the internal tricia directories, it is not possible to import facets from the local file system directly. They must be imported before with the testdata import. In the first step, one tricia directory is selected as root import directory. With a click on the `Start Import` button the facet import starts. Comparable to the testdata import, a state bar visualizes the current import state. During the import the results are represented in various notations. The selected part of the hierarchical source file system is illustrated on the left. Files are not included in this representation, they does not effect the import. A graph visualizes the calculated subsumption of each tag. Additionally the corresponding facets are represented with the related context. The graph and facet placement differ from the represented screen.

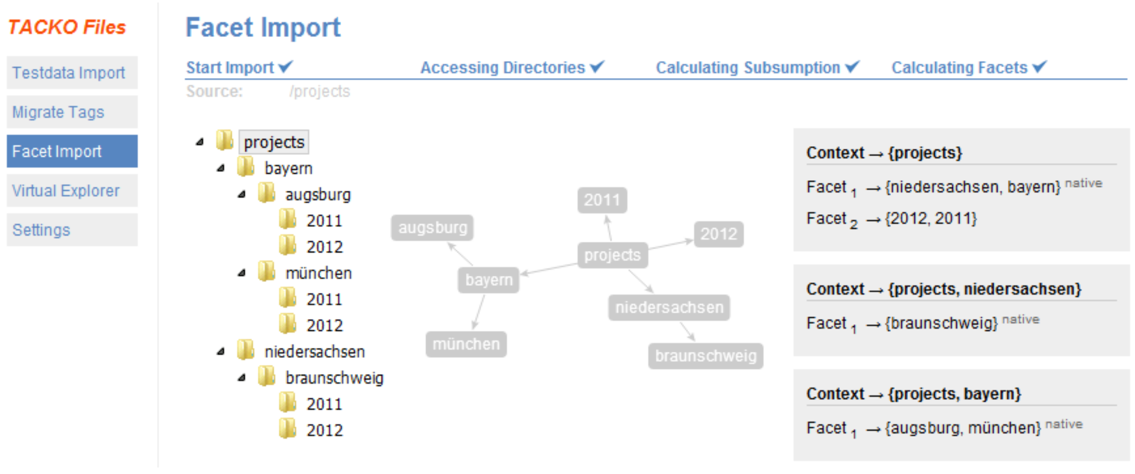


Figure 3.34: TACKO Files facet import.

All different kinds of navigation are provided by the virtual explorer (Figure 3.35). The interface contains two main components, on the left the tricia hierarchical file system and on the right the corresponding tag-based navigation. As described in the previous chapters, there are several tag-based navigation concepts, depending on the current system setting tag-based navigation concept is selected. This sample shows the multifaceted navigation with count option. Every explorer has a gray bar on top, the name of the visualized navigation concept is written in blue. The current selected path is represented below the name. If the path is too long, folder names are cut and replaced with three dots. In this case it is not possible to represent one single folder name, the path is cut within a folder name.

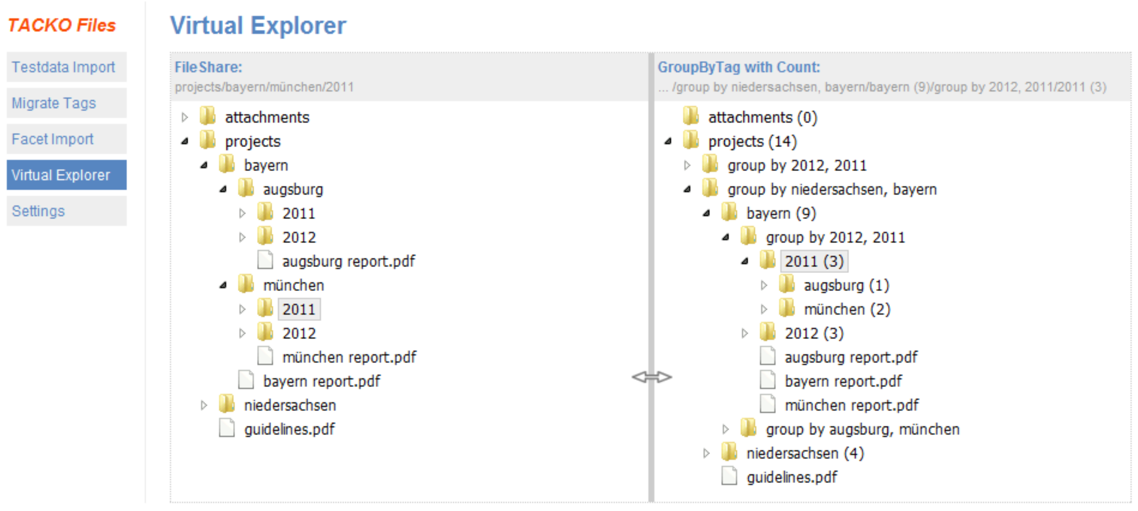


Figure 3.35: TACKO Files virtual explorer.

These two navigation components are resizable with a left click on the split bar and moving the mouse to the left or right. By default both components have an equal size. Especially the multifaceted navigation concept needs more space so it is easily possible to use more space for this and less for the source component. Additionally it is also possible to move the split bar completely to the left or right than only one navigation concept is visible. Within this tag-based view of the virtual explorer, all CRUD operations are executable with a context menu. The virtual explorer allows to compare each tag-based view easily with the hierarchical source file system.

Finally, the menu item settings allow to change between the tag-based navigation concepts. Figure 3.36 illustrates the settings view. All four tag-base navigation concepts are illustrated with a sample. The concept with the blue border is the current selected one. Clicking on any other concept selects this.

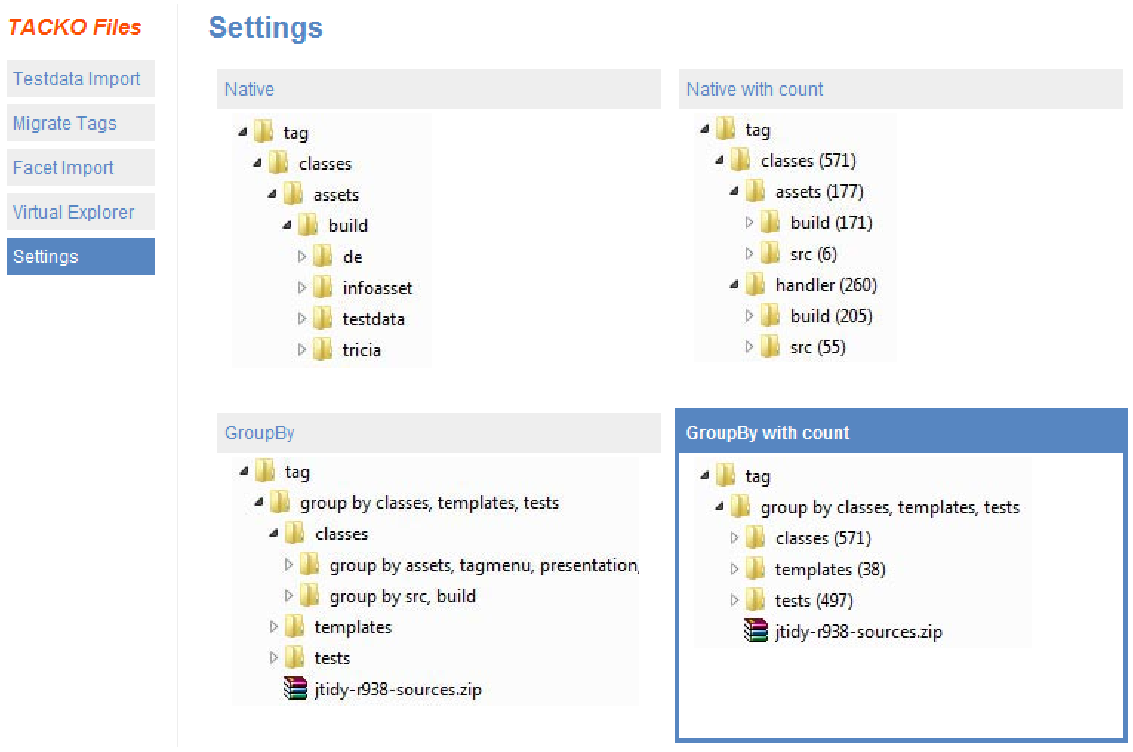


Figure 3.36: TACKO Files navigation concept settings.

3.5 Visionary user interface scenario

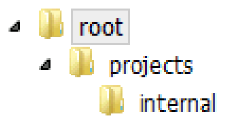
In a visionary future scenario all views and manipulating operations are fully integrated on the mountable network device. Although the technical implementation is possible, there are still some open issues (see Chapter 3.3.3.2). The context menu items and dialogs from a common hierarchical file system don't match completely for a tag-based system. Users are confused by wrong names and unexpected results. Currently all crud operation dialogs provide a lot of file system mapping information (see Chapter 3.3). To improve the usability for a normal user this information should be removed.

3.6 Facet testcase definition

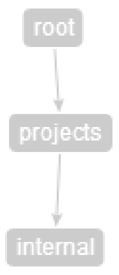
All navigation concepts described in Chapter 3.2 are based on the TACKO model. The TACKO data model stores facets corresponding to a certain context. The context and facets are imported with the TACKO Files import and stored in the data model. All navigation concepts using these imported facets. Therefore it is really important to define these facets clearly for certain a context. This chapter describes samples of hierarchical source file systems and defines the facets for each context. Testcases define the facet finding algorithm (see Chapter 4.1.3) behavior based on samples. Each example contains a hierarchical source file system and the corresponding context and facets. In addition, a graph represents the subsumption (see Chapter 4.1.4) of each tag. The first sample explains the basics more detailed.

3.6.1 Trivial two folder facet testcase

This first facet test case uses a trivial hierarchical file system as source (Figure 3.37(a)). On the right side, the corresponding context and facets (Figure 3.37(c)) are illustrated. In between, the related subsumption graph is pictured (Figure 3.37(b)). The sample has only three directories, beginning with the `root` directory on top. This contains a subdirectory `projects`, which itself contains a subdirectory named `internal`. In relation to the hierarchical file system, the graph visualizes the subsumption of each tag as graph. The subsumption graph is as trivial as the file system and represents three hierarchically ordered tags. The `root` tag subsumes the `projects` and `internal` tag. `projects` is subsumed by `root` and subsumes itself the tag `internal`. Derived from this graph, all contexts and its corresponding facets are formed. Each node contained in the subsumption graph represents potential contexts. A context contains all tags along the graph. Facets are formed from direct child nodes of a context. Therefore the subsumption graph leaves do not result in an own context. In this case it is also trivial, the context `root` contains a facet with only one tag `projects`. The second context {`root, projects`} has a facet with the tag `internal`. All resulting facets are native, that means the corresponding context exists as folder path in the hierarchical source file system and the context related folder contains all facet tags as subdirectories.



(a) hierarchical file system



(b) subsumption graph

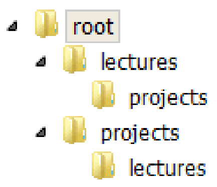
$$\begin{array}{rl} \textbf{Context} \rightarrow & \{\textbf{root}\} \\ \text{Facet}_1 \rightarrow & \{\text{projects}\}^{native} \\ & \\ \textbf{Context} \rightarrow & \{\textbf{root, projects}\} \\ \text{Facet}_1 \rightarrow & \{\text{internal}\}^{native} \end{array}$$

(c) context and facets

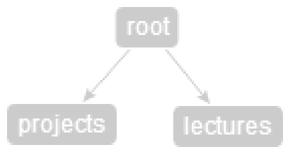
Figure 3.37: Trivial two folder facet testcase.

3.6.2 Permutation facet testcase

The first testcase is trivial, real hierarchical file systems include several special cases from the tag-based point of view. This sample covers a simple permutation in a hierarchical file system (Figure 3.38(a)). The `root` directory has two subdirectories, `lectures` and `projects`. The directory `lectures` itself contains a subdirectory named `projects` and the directory `\root\projects\` contains a subdirectory `lectures`. Last one causes the permutation in this sample. The hierarchical file system represents not the coverage of tags. Compared to the subsumption graph (Figure 3.38(b)), the file system has two more subdirectories. In tag-based systems a tag is unique in comparison to hierarchical file system where a path is unique and each directory itself has only unique children. The directory names set of a hierarchical file system represent all possible tag names. Resources with equal names are represented by one tag. The subsumption graph (Figure 3.38(b)) shows the subsumption of this sample. The graph represents no permutation. Tags which exists more than once in the source file system are only represented once in the subsumption graph. The tag is placed as in the graph as the most general directory name in the file system. In this case more directories for the tag lectures exist. These directories are `\root\lectures\` and `\root\projects\lectures\`. First one is more general and the tag is related to this path placed in the subsumption graph. Equivalent to this the tag `projects` is placed in the subsumption graph. One context exists for this graph, the `root` context, which contains one facet with two tags `projects` and `lectures`. The resulting facet is not native due to the permutation. A facet is native if all facet tags exists exactly once in the hierarchical source file system as directory. In general the native facet of a context represents the first facet. In this sample only one resulting facet exists and the facet ordering is not influenced.



(a) hierarchical file system



(b) subsumption graph

Context $\rightarrow$ {**root**}
Facet$_1$ $\rightarrow$ {projects, lectures}

(c) context and facets

Figure 3.38: Permutation facet testcase.

3.6.3 Identical names within a permutation facet testcase

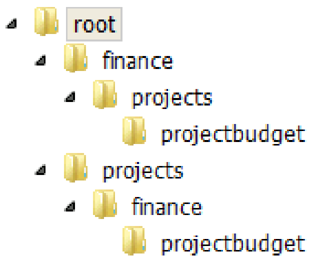
The identical names within a permutation sample extend the permutation (see Chapter 3.6.2) sample. Compared to this sample, the single structural difference is the new subdirectory `projectbudget` within the permutation. The directory `lectures` is renamed to `finance`, this has no structural effects and is only for a consistent and meaningful naming. The permuted directories are `finance` and `projects`. They are handled as in the first permutation sample (Figure 3.38).

Identical directory name paths with permutation:
`\root\finance\projects\`**`projectbudget`**`\`
`\root\projects\finance\`**`projectbudget`**`\`

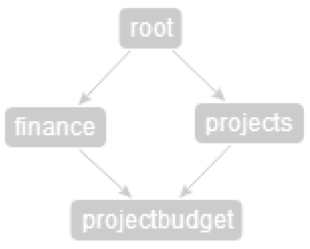
Identical directory name paths without permutation:[2]
`\root\finance\`**`projectbudget`**`\`
`\root\projects\`**`projectbudget`**`\`

As already known, permutated directory names are only represented with one tag in the subsumption graph. The Figure above illustrates the synonym directory path with and without permutation. Be careful this is not a generalization like the golf example (see Chapter 3.6.4). The subsumption graph represents the second case without permutation (Figure 3.39(b)). According to the illustrated path example, projectbudget is subsumed by two different tags, `finance` and `projects`. The graph contains two different contexts, {`root`} and {`root, finance, projects`}. The second one contains both tags `finance` and `projects` because `projectbudget` does not exist in a path with only one of these tags. Not in the path `\root\finance\` and neither in the path`\root\projects`. It exists only in one of the synonym name paths with permutation, that means both tags are required but the order does not matter. The first resulting context { `root`} contains a merged facet {`finance, projects`}. The second special context { `root, finance, projects`} contains one facet {`projectbudget`}. Due to the permutation all facets are synthetic and not native.

[2]This only for clarification and NOT a part of any algorithm, it illustrates what the subsumption graph represents in this special case.



(a) hierarchical file system



(b) subsumption graph

Context →	{**root**}
Facet$_1$ →	{projects, finance}
Context →	{**projects, root, finance**}
Facet$_1$ →	{projectbudget}

(c) context and facets

Figure 3.39: Synonyms within a permutation facet testcase.

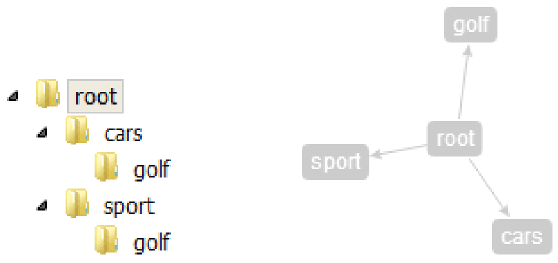
3.6.4 Golf generalization facet testcase

Generalization of folder names is covered in this chapter. The concrete example is a hierarchical file system which contains two folders named `golf` (Figure 3.40(a)). In this case `golf` is a homonym, it has two different meanings. One describes the sport golf and the other one the car. This sample is semantically not generalizable but it is not possible to differ between synonyms and homonyms. Analyzing file and folder names and compare it with some other tagged resources with more meta data could be one approach. This is an open issue and not covered in this work. In this example the simplest case of generalization is illustrated.

Identical directory name paths:
`\root\cars\`**`golf`**`\`
`\root\sport\`**`golf`**`\`

Identical directory name path generalized:
`\root\`**`golf`**`\`

The Figure above shows two synonym name directory paths. Generalization of tags means to find the most general path of a directory name. The paths differs only in one path section. After removing all names which are not contained in both paths, the subsumption of the synonym is found (Figure 3.40(b)). In general the intersection of all synonym name paths resulting in the most general path tags. The resulting subsumption graph has an edge form the `root` to `golf`. This edge represents the generalized subsumption. Only one context is contained in the graph, `root` and the corresponding facet {`car, sport, golf`}. All facets are native with one exception, the synthetic generalized facet {`golf`}.



(a) hierarchical file system (b) subsumption graph

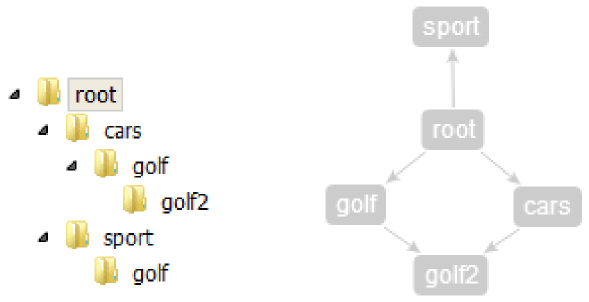
Context $\rightarrow$ {**root**}
Facet$_1$ $\rightarrow$ {cars, sport}^{native}
Facet$_2$ $\rightarrow$ {golf}

(c) context and facets

Figure 3.40: Golf generalization facet testcase.

3.6.5 Golf2 generalization facet testcase

This testcase extends the golf generalization sample (Figure 3.6.4). In comparison to the hierarchical source file system of the golf sample, one directory named `golf2` is added to the path \\`root`\\`cars`\\`golf`\\ (Figure 3.41(a)). The hierarchical file system illustrates clearly that the folder `golf2` is only accessible in this path. This navigation restriction is also considered by the resulting subsumption graph (Figure 3.41(b)). The tag `root` subsumes directly only `sport`, `cars` and `golf`. Only in the context {`root, cars, golf`} contains the tag `golf2`. These subsumption graph results in an additional context compared to the golf example. The additional context is {`root, cars, golf`} with a corresponding facet {`golf2`}. Equal to the golf sample only the facet generalized facet {`golf`} is not native.



(a) hierarchical file system (b) subsumption graph

Context $\rightarrow$ {**root**}
Facet$_1$ $\rightarrow$ {cars, sport}^{native}
Facet$_2$ $\rightarrow$ {golf}

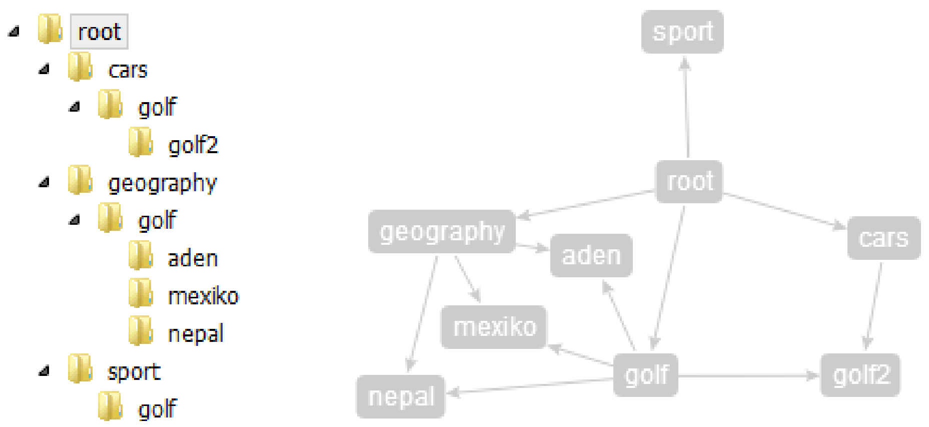
Context $\rightarrow$ {**root, cars, golf**}
Facet$_1$ $\rightarrow$ {golf2}^{native}

(c) context and facets

Figure 3.41: Golf2 generalization facet testcase.

3.6.6 Golf2 advanced generalization facet testcase

This golf2 advanced testcase extends the golf2 sample. It illustrates the already described concepts with more data and presents no more new conceptual ideas. Some more folders are added to the hierarchical source file system of the golf2 (Figure 3.41(c)) sample. In the context of geography golf gets one more synonym meaning. A folder `geography` is added to the `root` which contains a `golf` subdirectory. This subdirectory contains several childs, `aden`, `mexiko` and `nepal` (Figure 3.42(a)). For each `...\geography\golf\` subfolder the conceptual solution is equal with the one of the folder `golf2`. The subsumption graph (Figure 3.42(b)) illustrates that the new tags `aden`, `mexiko` and `nepal` are only accessible with the tags path `\root\golf\geography\...` or `\root\geography\golf\ ...`. It is not possible to reach them with only one of the tags `geography` or `golf`. Compared to the golf2 extended sample (Figure 3.41(c)), one more native facet {`aden, mexiko, nepal`} exists within the context {`root, golf, geography`} and the first facet of the context {`root`} is extended with the facet tag `geography`.



(a) hierarchical file system (b) subsumption graph

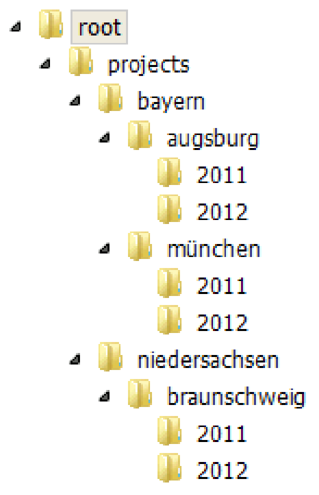
Context →	{**root**}
Facet$_1$ →	{cars, sport, geography}^{native}
Facet$_2$ →	{golf}
Context →	{**root, golf, geography**}
Facet$_1$ →	{aden, mexiko, nepal}^{native}
Context →	{**root, cars, golf**}
Facet$_1$ →	{golf2}^{native}

(c) context and facets

Figure 3.42: Golf2 advanced generalization facet testcase.

3.6.7 Projects facet testcase

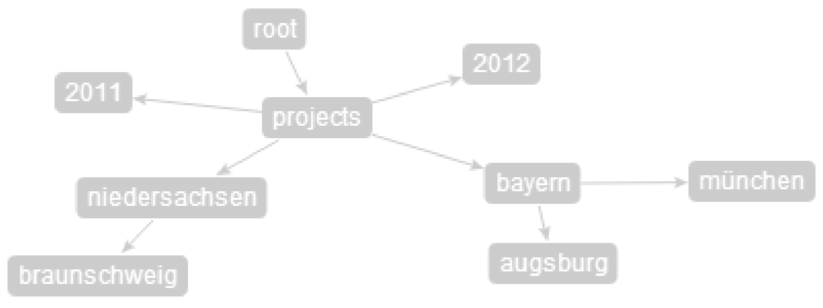
The project facet testcase represents a sample data structures with common problems. On the root level of hierarchical file systems are often category folders which contain in several subdirectories. Some on these subdirectories itself are structured with subfolder which represent a certain year. In most cases structuring the data on the root level with year folders within the categories would be also possible. In a hierarchical file systems only one structural concept can be applied. This hierarchical source file system (Figure 3.43(a)) has a folder `projects` structured with subdirectories named like the federal states of Germany. These federal states folders are organized by cities. Each city folder contains some year folders. This testcase use the tag generalization concept of the golf sample (see Chapter 3.6.4). The subsumption graph illustrates that in the context {`root, projects`} already contains the generalized year tags `2011` and `2012` as facet. This is a synthetic facet due to the generalization. In other words the tag `projects` is by default browsable by the federal states and with this new facet also by year.



(a) hierarchical file system

Context $\rightarrow$	{**root**}
Facet_1 $\rightarrow$	$\{\text{projects}\}^{native}$
Context $\rightarrow$	{**root, projects**}
Facet_1 $\rightarrow$	$\{\text{bayern, niedersachsen}\}^{native}$
Facet_2 $\rightarrow$	{2011, 2012}
Context $\rightarrow$	{**root, projects, bayern**}
Facet_1 $\rightarrow$	$\{\text{augsburg, münchen}\}^{native}$
Context $\rightarrow$	{**root, projects, niedersachsen**}
Facet_1 $\rightarrow$	$\{\text{braunschweig}\}^{native}$

(b) context and facets



(c) subsumption graph

Figure 3.43: Projects facet testcase.

4 Transforming Algorithms

In order to provide the navigation concepts, several algorithms are necessary. Import and export algorithms such as CRUD operation mapping algorithms are systematically developed. The first Subchapter 4.1 covers the different kinds of imports. This algorithm is encapsulated and has no dependencies to other Chapters. In general the diagram in Figure 4.1 illustrates the major dependencies between all algorithms with exception of the import algorithms. The diagram is structured into three layers, a data layer, an abstraction layer to encapsulate basic often used functions and the algorithms which combines all helper methods. On the bottom a hierarchical file system and the TACKO facet data model are placed. In between there are files which are tagged. These files are accessible via the hierarchical file system or with a tag search. All red colored functions represent fundamental accessing methods. In general these are illustrated in Chapter 4.2. Basic methods which are only used for one algorithm are described in the Chapter of the main algorithm. The gray algorithms can be divided into two groups. On the right the export algorithms are placed (see Chapter 4.3) Further, on the left all CRUD operation algorithms are presented (see Chapter 4.4).

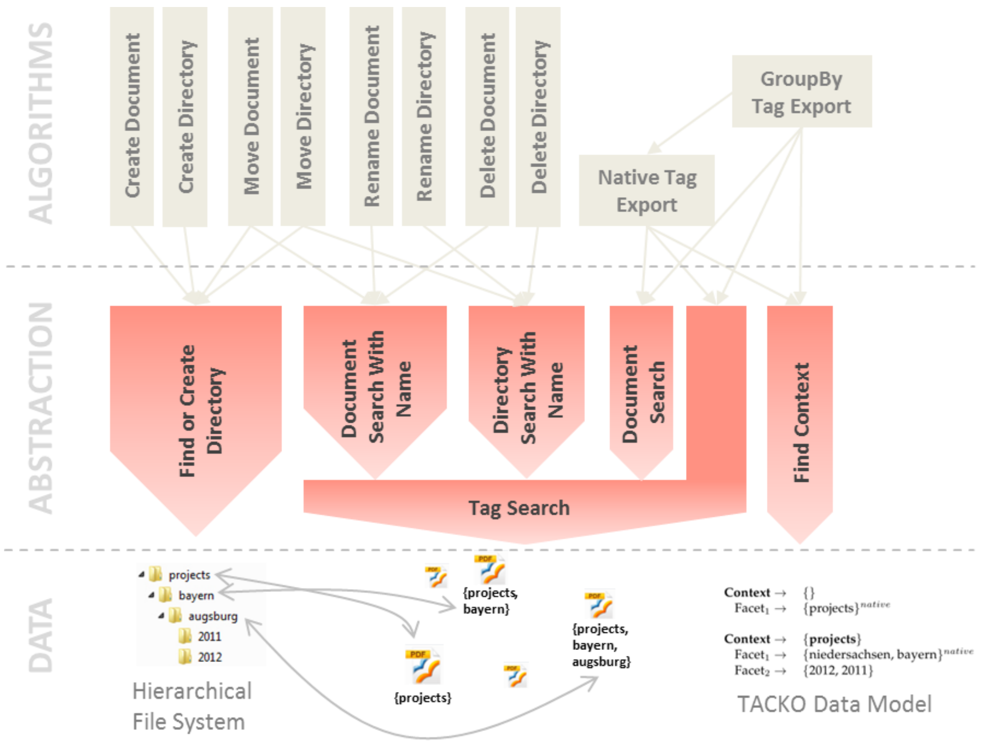


Figure 4.1: TACKO Files abstract algorithm overview.

4.1 Importing tags

Import tags are a synonym for transforming a hierarchical file system into a tag-based system (Figure 4.2). Before any import algorithm is applied on the file system, it must be checked for certain properties and adjusted if necessary (see Chapter 4.1.1). Two different types of imports are described. First of all the simple name based tag import is covered in Chapter 4.1.2. This import considers only the path names to import the tags. The resources are assigned with the tags. Further the advanced facet import algorithm is covered in Chapter 4.1.3.The resulting facets with the corresponding context are stored in the TACKO data model. Additionally a subsumption graph can be calculated together with the facet import.

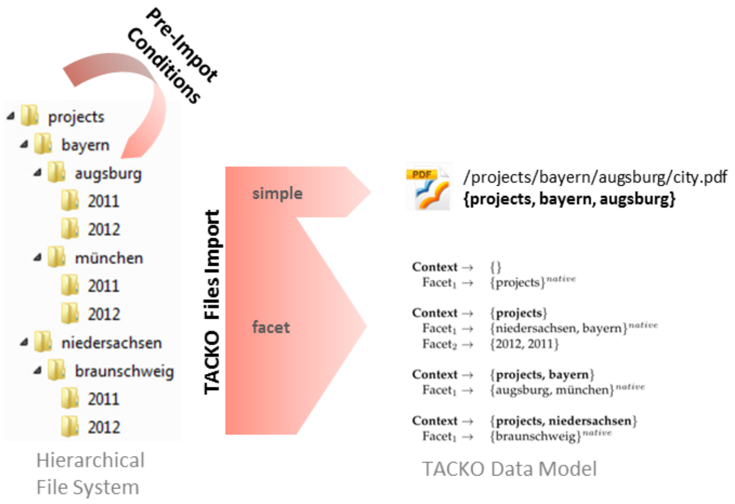

Pre-Impot Conditions
projects
bayern
augsburg
2011
2012
münchen
2011
2012
niedersachsen
braunschweig
2011
2012
Hierarchical File System
TACKO Files Import
simple
facet
/projects/bayern/augsburg/city.pdf
{projects, bayern, augsburg}
Context → {}
Facet$_1$ → {projects}^{native}
Context → {projects}
Facet$_1$ → {niedersachsen, bayern}^{native}
Facet$_2$ → {2012, 2011}
Context → {projects, bayern}
Facet$_1$ → {augsburg, münchen}^{native}
Context → {projects, niedersachsen}
Facet$_1$ → {braunschweig}^{native}
TACKO Data Model

Figure 4.2: TACKO Files impot overview.

4.1.1 Hierarchical file system pre-import-conditions

A hierarchical file system represents a hierarchical order on folder and files. In the best case it is hierarchally ordered and its folders represents a taxonomy. That's the theoretical organization of a hierarchical file system, but in reality there is only partly a taxonomy identifiable. Not all directory paths represent semantically a parent child relation. The Import algorithms are not able to cope with some special cases in a meaningful fashion. Therefore a file system must fulfil some pre-conditions. In order to obtain as few synonymous tags as possible, it is necessary to rename all folder names to lowercase folder names. This reduces the variation of synonyms enormously. The most important pre-condition is that a path doesn't contain any cycle. If this condition is not fulfilled, the naming is changed. It is important to rename first and remove the cycles in the second step.

Figure 4.3 illustrate an example which covers various problems. Subfigure 4.3(a) shows the original unchanged hierarchical file system. The root folder contains three subfolders starting with upper letters. This original example contains one cycle. Folders of cycle's are printed bold.

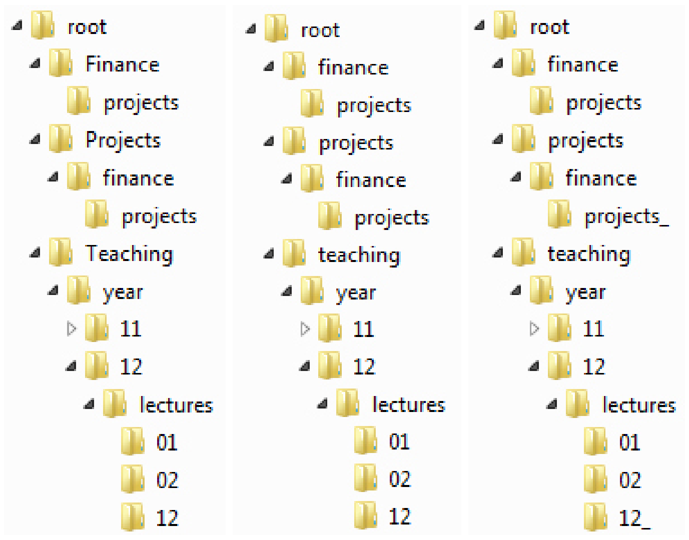
Cycle path 1:	/root/Teaching/year/**12**/lectures/**12**/

After converting folders to lower case (Figure 4.3(b)), the first two folders are containing a permutation. This is not a problem it only demonstrates that synonym tags can be reduced. But by renaming, the file system contains one more cycle.

Permutation:	/root/**Finance**/**projects**/
	/root/**Projects**/**finance**/
Cycle path 2:	/root/**projects**/finance/**projects**/

In the last step all cycles are removed by adding an underscore. If a path contains more than one cycle, one more underscore is added for each cycle. Figure 4.3(c) shows the final result of the pre-import.

Cycle path 1:	/root/Teaching/year/12/lectures/**12**_/
Cycle path 2:	/root/projects/finance/**projects**_/



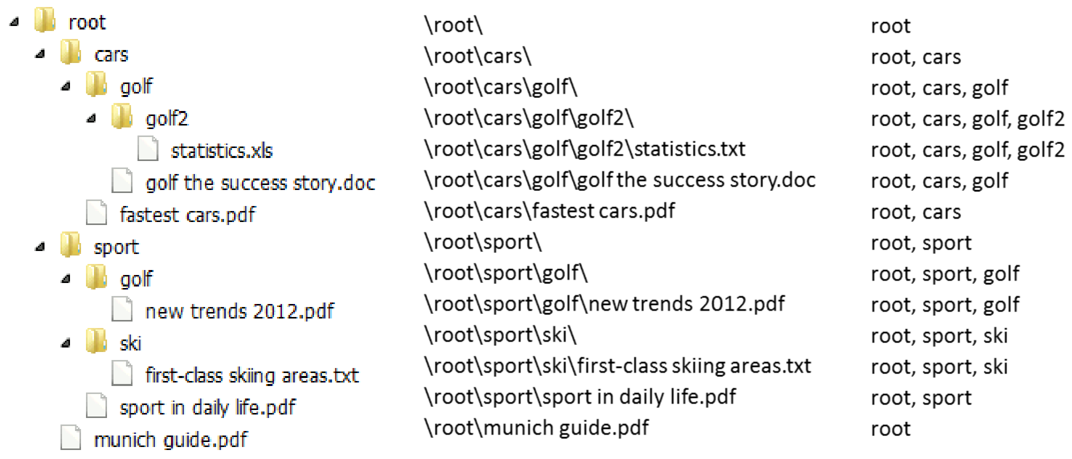
(a) unchanged (b) lower-case (c) no cycles

Figure 4.3: Tag-pre-import sample states.

Further there are some more nice to have pre-import conditions. Such as folder names, file names could also be converted to lower case file names. But windows already prevents having two file names or folder names as siblings which only differ in upper and lower case letters. Converting all file names to lowercase reduces the readability. Therefore file names are not converted. Unique document names in the whole file system would be really helpful for tag-based navigation and especially for crud operations (see Chapters 3.3, 4.4). If a good usability is desired, this is obviously not realizable.

4.1.2 Simple name based tag import

Importing tags based on folder names is the simplest fashion. Figure 4.4 exemplified an import, Figure 4.4(a) shows the hierarchical file system. This is a view of the web-based interface, it shows files and folders directly in one tree, it is extraordinary but a helpful representation. Each folder is tagged with the path containing parent directories including itself. Files are not tagged with the whole path, only with the path containing parent directories. In other words only with the folder names and not with the filename. Figure 4.4(c) shows the resulting tags for each path. The algorithm works on the fly, when a new directory or file is added, it is tagged with its path tags as described in this chapter.



```
root
  cars
    golf
      golf2
        statistics.xls
      golf the success story.doc
    fastest cars.pdf
  sport
    golf
      new trends 2012.pdf
    ski
      first-class skiing areas.txt
    sport in daily life.pdf
  munich guide.pdf
```

\root\	root
\root\cars\	root, cars
\root\cars\golf\	root, cars, golf
\root\cars\golf\golf2\	root, cars, golf, golf2
\root\cars\golf\golf2\statistics.txt	root, cars, golf, golf2
\root\cars\golf\golf the success story.doc	root, cars, golf
\root\cars\fastest cars.pdf	root, cars
\root\sport\	root, sport
\root\sport\golf\	root, sport, golf
\root\sport\golf\new trends 2012.pdf	root, sport, golf
\root\sport\ski\	root, sport, ski
\root\sport\ski\first-class skiing areas.txt	root, sport, ski
\root\sport\sport in daily life.pdf	root, sport
\root\munich guide.pdf	root

(a) hierarchical file system web-b. (b) paths (c) path tags

Figure 4.4: Simple tag import sample.

4.1.3 Facet finding algorithm

Aim of the Facet finding algorithm is to find a context and the related facets based on a hierarchical file system. In the previous Chapter 3.6 are facet test cases defined. These test cases describe a hierarchical file system and related to this the desired facets with corresponding context. To ensure that navigation behavior is correct, it is important to import the facets according to these well-defined test cases. The algorithm is decomposed into several sub algorithms. Figure 4.5 shows the sequence of applied algorithms and the corresponding in- and outputdata as resources. Every oval represents a resource. A dashed line with an arrow pointing to a resources visualizes output data and vice versa. First of all, a directory name mapping based on the hierarchical source file system is created (see Chapter 4.1.3.1). In the next step, the resulting directory name map is used as input for the subsumption calculation (see Chapter 4.1.3.2). The output is a subsumption map which contains the subsumption for each tag. Additionally, a subsumption graph can be calculated to visualize the subsumption of each tag (see Chapter 4.1.4). This graph does not affect any other part of the algorithm. Based on the subsumption map, the potential facets are calculated (see Chapter 4.1.3.3) and result in a potential facet map. For the further facet finding process some other basic data are necessary. Therefore, all sibling directory names of the hierarchical source file system are listed in the next step (see Chapter 4.1.3.4). The result is a sibling names set which contains all siblings. Related to this name set, all possible sibling pairs are calculated (see Chapter 4.1.3.5). All basic data is now calculated, and the facet merge algorithm can merge the potential facets in consideration of the sibling pairs (see Chapter 4.1.3.6). Finally the merged facets are ordered in a meaningful fashion (see Chapter 4.1.3.7).

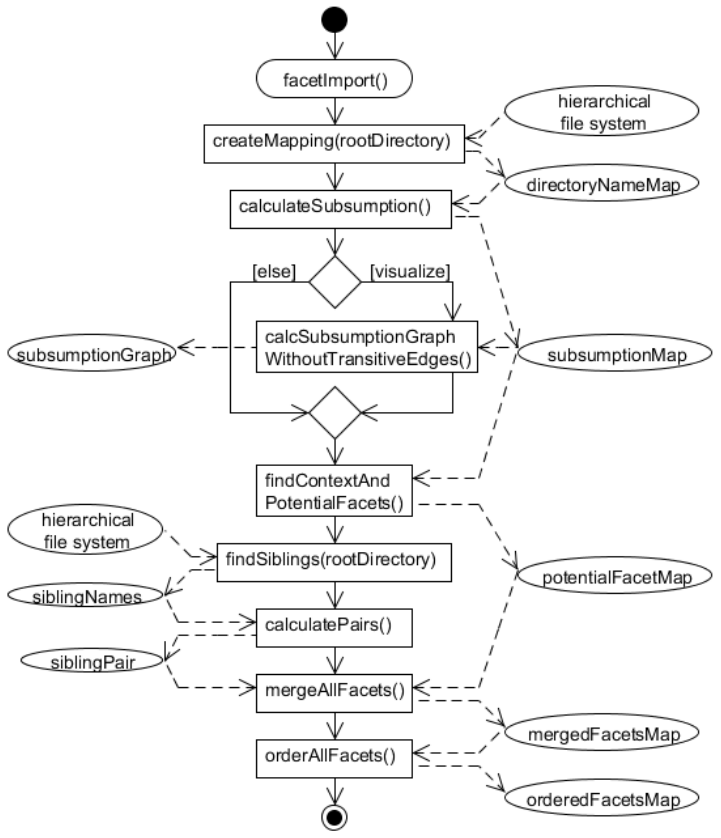
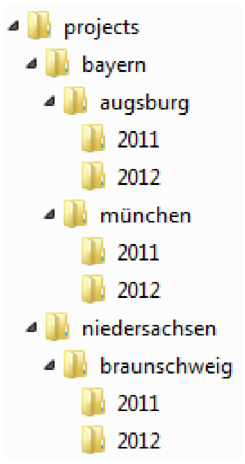


Figure 4.5: Facet import algorithm overview with in- and output resources.

4.1.3.1 Directory name mapping

The directory name mapping maps all directory names to all corresponding directory paths. The principle of path segmentation is equal to Chapter 4.1.2, but this algorithm works recursively and not on the fly[1]. Figure 4.6(a) shows the algorithm input data, a hierarchical file system. Aim of this algorithm is listing all directory names and map the corresponding paths as ordered list of path tags. Figure 4.6(b) illustrates this resulting output data.

[1]On the fly means incremental calculation, triggered from an on change event.



(a) hierarchical file system

$$directoryNameMap$$
$$< Map < directoryName, Set < List < pathTag >>>$$

Directory name	Path tags
2011	{[projects, bayern, augsburg], [projects, bayern, münchen], [projects, niedersachsen, braunschweig]}
2012	{[projects, bayern, augsburg], [projects, bayern, münchen], [projects, niedersachsen, braunschweig]}
augsburg	{[projects, bayern]}
bayern	{[projects]}
braunschweig	{[projects, niedersachsen]}
münchen	{[projects, bayern]}
niedersachsen	{[projects]}
projects	{[]}

(b) directory name to path tag map

Figure 4.6: Directory mapping.

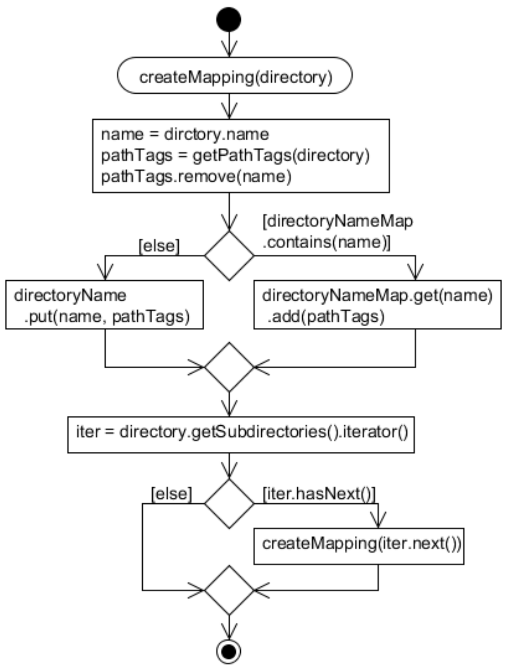

createMapping(directory)
name = dirctory.name
pathTags = getPathTags(directory)
pathTags.remove(name)
[else]
[directoryNameMap .contains(name)]
directoryName .put(name, pathTags)
directoryNameMap.get(name) .add(pathTags)
iter = directory.getSubdirectories().iterator()
[else]
[iter.hasNext()]
createMapping(iter.next())

Figure 4.7: Create directory name to path tags map algorithm.

The algorithm (Figure 4.7) starts with the root directory, in this example the `projects` directory. If the `directoryNameMap` (Figure 4.6(b)) does not contain the directory name as key, a new key with the corresponding path tags as value is added. Otherwise the path tags are added to a existing key in the map.

A new key `projects` is added to the map. The algorithm is applied recursively for all subdirectories. Further `bayern` is selected and added to the map. Next `augsburg` is selected and added etc. Figure 4.8 illustrates the results of the first few recursions, new items are printed bold. Adding a mapping of a not unique directory name is illustrated in Subfigure 4.8(g).

Directory name	Path tags
projects	{[]}

(a) step 1

Directory name	Path tags
bayern	{**[projects]**}
projects	{[]}

(b) step 2

Directory name	Path tags
augsburg	{**[projects, bayern]**}
bayern	{[projects]}
projects	{[]}

(c) step 3

Directory name	Path tags
2011	{**[projects, bayern, augsburg]**}
augsburg	{[projects, bayern]}
bayern	{[projects]}
projects	{[]}

(d) step 4

Directory name	Path tags
2011	{[projects, bayern, augsburg]}
2012	{**[projects, bayern, augsburg]**}
augsburg	{[projects, bayern]}
bayern	{[projects]}
projects	{[]}

(e) step 5

Directory name	Path tags
2011	{[projects, bayern, augsburg]}
2012	{[projects, bayern, augsburg]}
augsburg	{[projects, bayern]}
bayern	{[projects]}
münchen	{**[projects, bayern]**}
projects	{[]}

(f) step 6

Directory name	Path tags
2011	{[projects, bayern, augsburg], **[projects, bayern, münchen]**}
2012	{[projects, bayern, augsburg]}
augsburg	{[projects, bayern]}
bayern	{[projects]}
münchen	{[projects, bayern]}
projects	{[]}

(g) step 7

Figure 4.8: Directory name mapping algorithm, the first steps ... .

4.1.3.2 Subsumption

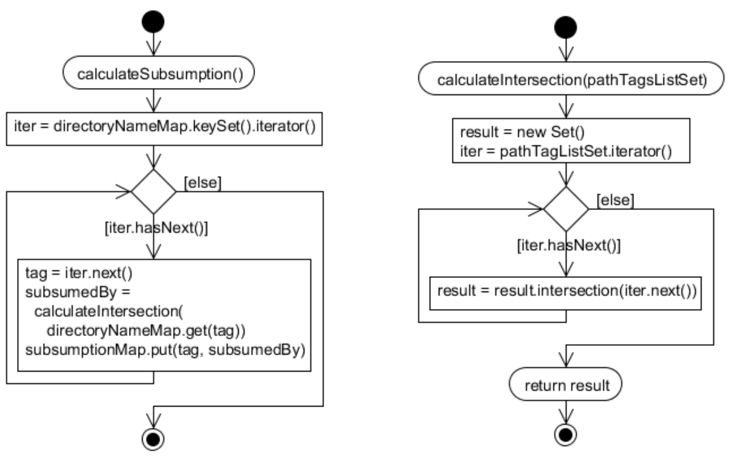
In this step the subsumption of all tags is calculated. Subsumption means, if `A` subsumes `B`, `B` is contained in one or more subdirectories of `A`, but never in a parent folder of `A`. The `directoryNameMap` is used as input data (Figure 4.6(b)). For each directory name in this map, the intersection of the corresponding paths are calculated. In the first step all path tag lists are converted into path tag sets. The intersection is pairwise calculated until only one result set for a directory name exists. This hierarchical file system example contains only two tags (`2011` and `2012`) which are affected by this calculation. Figure 4.9 shows this intersection principle.

$$\begin{aligned}
\texttt{2011}\text{ is subsumed by } &= \{\texttt{projects, bayern, augsburg}\} \cap \\
&\quad \{\texttt{projects, bayern, münchen}\} \cap \\
&\quad \{\texttt{projects, niedersachsen, braunschweig}\} \\
&= \{\texttt{projects}\}
\end{aligned}$$

$$\begin{aligned}
\texttt{2012}\text{ is subsumed by } &= \{\texttt{projects, bayern, augsburg}\} \cap \\
&\quad \{\texttt{projects, bayern, münchen}\} \cap \\
&\quad \{\texttt{projects, niedersachsen, braunschweig}\} \\
&= \{\texttt{projects}\}
\end{aligned}$$

Figure 4.9: Tag subsumption calculation principle.

Figure 4.10 illustrates the subsumption algorithm. The `calculateSubsumption` function iterates over all keys of the `directoryNameMap`. With the key which represents a directory name, all paths for this name are queried in the `directoryNameMap`. One or more lists of path tags are returned as a set. For calculating an intersection between more lists the helper function `calculateIntersection` is used. Within this function all lists of the set are merged to one resulting set. This set is returned and represents the subsumption of the directory name tag. After the iteration over all items is finished, the result is a subsumption map (Figure 4.11).


calculateSubsumption()
iter = directoryNameMap.keySet().iterator()
[else]
[iter.hasNext()]
tag = iter.next()
subsumedBy =
calculateIntersection(
directoryNameMap.get(tag))
subsumptionMap.put(tag, subsumedBy)
calculateIntersection(pathTagsListSet)
result = new Set()
iter = pathTagListSet.iterator()
[else]
[iter.hasNext()]
result = result.intersection(iter.next())
return result

(a) calc subsumption (b) intersection helper

Figure 4.10: Calculate subsumption algorithm.

$subsumptionMap$
$< Map < SubsumedBy, Set < Tag >>$

Subsumed by	Tags
2011	{projects}
2012	{projects}
augsburg	{projects, bayern}
bayern	{projects}
braunschweig	{projects, niedersachsen}
münchen	{projects, bayern}
niedersachsen	{projects}
projects	{}

Figure 4.11: Tag subsumption map.

The subsumption of tags represent also a graph with edges. Each subsumed tag has incoming edges from the corresponding tags, but here also transient edges may be included. If more than one tag subsumes a tag, there are transitive edges, because each folder has only one parent folder. E.g., `augsburg` has an incoming edge from `bayern` and `projects`. The edge from `projects` to `augsburg` is transitive. Tags which are subsumed by an empty set are root level tags. In this case `projects` is not subsumed by any other tags and represents a root tag.

4.1.3.3 Potential Facets

Potential facets are calculated based on the subsumption map (Figure 4.11). In this step a subsumption inverse map is formed. All values of the subsumption map represent the `context` and they are put in the new `potentialFacetMap` as key. All keys represent the `potential facets` and stored as values in the new map. When the key in the new map already exists, the potential facets are added to the already existing ones, see inverse algorithm in Figure 4.14. All inverse calculation steps are illustrated in Figure 4.13, new added items are printed bold. Figure 4.12 represents the resulting context facet map. The first row represents the empty context, that means the corresponding facet represents the root case.

$potentialFacetMap$
$Map < Set < ContextTag >, Set < PotentialFacet >>$

Context tags	Potential facets
{}	{projects}
{projects}	{2011, 2012, bayern, niedersachsen}
{projects, bayern}	{augsburg, münchen}
{projects, niedersachsen}	{braunschweig}

Figure 4.12: Context and potential facets map.

Context tags	**Potential facets**
{**projects**}	{**2011**}

(a) Step 1

Context tags	**Potential facets**
{projects}	{2011, **2012**}

(b) Step 2

Context tags	**Potential facets**
{projects}	{2011, 2012}
{**projects, bayern**}	{**augsburg**}

(c) Step 3

Context tags	**Potential facets**
{projects}	{2011, 2012, **bayern**}
{projects, bayern}	{augsburg}

(d) Step 4

Context tags	**Potential facets**
{projects}	{2011, 2012, bayern}
{projects, bayern}	{augsburg}
{**projects, niedersachsen**}	{**braunschweig**}

(e) Step 5

Context tags	**Potential facets**
{projects}	{2011, 2012, bayern}
{projects, bayern}	{augsburg, **münchen**}
{projects, niedersachsen}	{braunschweig}

(f) Step 6

Context tags	**Potential facets**
{projects}	{2011, 2012, bayern, **niedersachsen**}
{projects, bayern}	{augsburg, münchen}
{projects, niedersachsen}	{braunschweig}

(g) Step 7

Context tags	**Potential facets**
{}	{**projects**}
{projects}	{2011, 2012, bayern, niedersachsen}
{projects, bayern}	{augsburg, münchen}
{projects, niedersachsen}	{braunschweig}

(h) Step 8

Figure 4.13: Find context and potential facet calculation step by step.

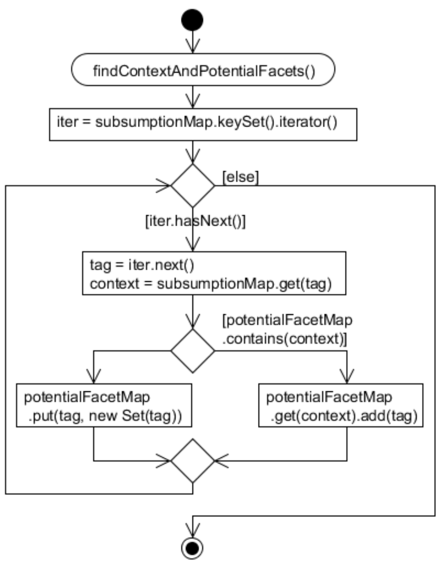
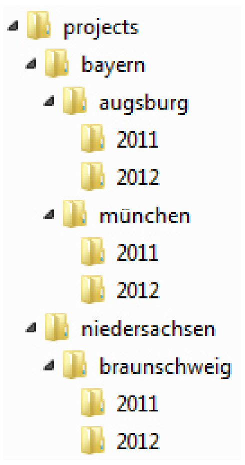


Figure 4.14: Find context and potential facet algorithm.

4.1.3.4 Sibling directory names

In order to merge the potential facets to meaningful facets, all pairs of sibling directory names are required. Based on the hierarchical file system (Figure 4.15(a)), this algorithm step lists all sibling directory names as result (Figure 4.15(b)). The algorithm (Figure 4.16) iterates recursively over all directories beginning with the root directory and stores the siblings names of each directory in a set. After accessing all subdirectories, the algorithm terminates and all siblings are calculated.



```
projects
  bayern
    augsburg
      2011
      2012
    münchen
      2011
      2012
  niedersachsen
    braunschweig
      2011
      2012
```

(a) hierarchical file system

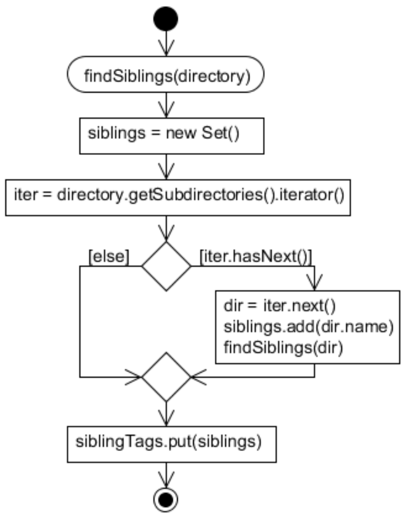
$siblingNames$

$Set < Set < SiblingDirectoryName >>$

Sibling directory names
{niedersachsen, bayern}
{projects}
{2012, 2011}
{braunschweig}
{augsburg, münchen}

(b) sibling directory names

Figure 4.15: Sibling directories.



```
findSiblings(directory)
siblings = new Set()
iter = directory.getSubdirectories().iterator()
[else]
[iter.hasNext()]
dir = iter.next()
siblings.add(dir.name)
findSiblings(dir)
siblingTags.put(siblings)
```

Figure 4.16: Find find siblings algorithm.

4.1.3.5 Sibling directory name pairs

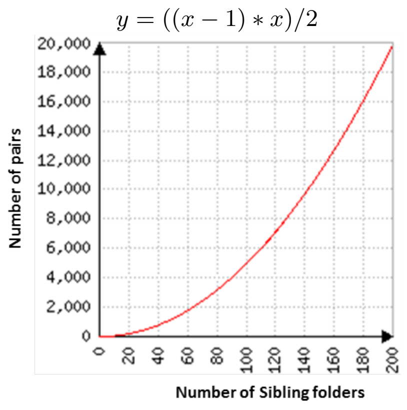
The sibling directory name pairs are calculated based on the sibling directory names (see Chapter 4.1.3.4). Input is the sibling directory name set (Figure 4.15(b)). The algorithm calculates all possible pairs of combinations for each set. The resulting pairs are illustrated in Figure 4.17(a). If a set contains one item e.g. `projects`, this will be discarded. In order to preserve the clarity of the whole example, this hierarchical file system contains maximum two siblings in one directory. That's the reason why this sample data does not end up with many pairs. Real data normally contains a huge amount of siblings, especially messy folder structures. The number of pairs are not increasing linear with the corresponding number of siblings directories (Figure 4.17(b)). Pairs are sets, as result $pair(A, B)$ is equal to $pair(B, A)$. The following formula expresses the relation between paris and subdirectories.

$$y_P : number\ of\ pairs; \qquad x_S : number\ of\ siblings; \qquad y_P = \frac{(x_s-1)*x_s}{2} \sim \frac{x^2}{2}$$

$siblingPairs$
$Set < SetPair < SibilngPair >>$

Sibling directory name pairs
{ niedersachsen, bayern}
{ 2012, 2011}
{ augsburg, münchen}

(a) sibling directory name pairs



(b) pairs in relation of siblings

Figure 4.17: Sibling directory names.

4.1.3.6 Facet merging

The facet merging is one of the final algorithm steps, based on the potential facets (Figure 4.12) and the sibling directory name pairs (Figure 4.17(a)). Aim is to find facets which represent meaningful a category e.g. the tags `2011` and `2012` represents the category year. First of all the potential facets are transformed. Each potential facet set item gets an own set which represents an initial facet (Figure 4.18).

$$splitFacetMap$$
$$< Map < Set < Context >, Set < Set < FacetTag >>$$

Context →	{}
Facet$_1$ →	{projects}
Context →	{**projects**}
Facet$_1$ →	{2011}
Facet$_2$ →	{2012}
Facet$_3$ →	{bayern}
Facet$_4$ →	{niedersachsen}
Context →	{**projects, bayern**}
Facet$_1$ →	{augsburg}
Facet$_2$ →	{münchen}
Context →	{**projects, niedersachsen**}
Facet$_1$ →	{braunschweig}

Figure 4.18: Context and potential facets split.

In the second step, the split facets are merged based on the sibling directory name pairs. Figure 4.18 illustrates all merge steps of this sample. The states before and after the merge are represented, merged facets are printed bold. In the first merge the third and fourth facet is merged to one facet with the tags `bayern` and `niedersachsen`.

Context →	{**projects**}
Facet$_1$ →	{2011}
Facet$_2$ →	{2012}
Facet$_3$ →	{bayern}
Facet$_4$ →	{niedersachsen}

(a) before first merge

Context →	{**projects**}
Facet$_1$ →	{2011}
Facet$_2$ →	{2012}
Facet$_3$ →	{**bayern, niedersachsen**}

(b) after first merge

Context →	{**projects**}
Facet$_1$ →	{2011}
Facet$_2$ →	{2012}
Facet$_3$ →	{niedersachsen, bayern}

(c) before second merge

Context →	{**projects**}
Facet$_1$ →	{**2012, 2011**}
Facet$_2$ →	{niedersachsen, bayern}

(d) after second merge

Context →	{**projects, bayern**}
Facet$_1$ →	{augsburg}
Facet$_2$ →	{münchen}

(e) before third merge

Context →	{**projects, bayern**}
Facet$_1$ →	{**münchen, augsburg**}

(f) after third merge

Figure 4.19: Facet merge steps.

The final merge result is represented in Figure 4.20. Compared to the split facet map, this merged facet map contains fewer facets. Three facets are merged and represented in the merged facet map as one facet.

$$mergedFacetMap$$
$$< Map < Set < Context >, Set < Set < FacetTag >>$$

Context →	{}
Facet$_1$ →	{projects}
Context →	{**projects**}
Facet$_1$ →	{2011, 2012}
Facet$_2$ →	{niedersachsen, bayern}
Context →	{**projects, bayern**}
Facet$_1$ →	{münchen, augsburg}
Context →	{**projects, niedersachsen**}
Facet$_1$ →	{braunschweig}

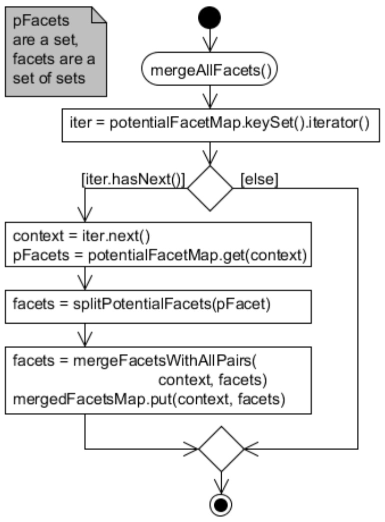
Figure 4.20: Context and merged facets.

All these visual illustrated merge steps are done by several merging algorithms (Figure 4.21). The function `mergeAllFacets` (Figure 4.21(a)) starts the merging process. Within one iteration over the `potentialFacetMap` all facets will be merged. In the first step within the iteration the function `splitPotentialFacets` (Figure 4.21(b)) splits the potential facets into a single set of facets[2]. Each item of the old set becomes an own set, all these sets together represent the facets for the iteration context. The splitted facets are returned and stored in a local variable.

In the next step, these facets are merged with the `mergeFacetWithAllPair` algorithm (Figure 4.21(c)). The splitted `facets` are handed over as parameter. The algorithm starts a iteration over all `siblingPairs`. The `facets` are merged with every sibling pair item. This merge is done with the `mergeFacetsWithPair` algorithm. Two parameters are necessary, the `pair` and the current `facets`. When the merge is done, the may merged `facets` are returned and the resulting facets are merged with all other sibling pairs.

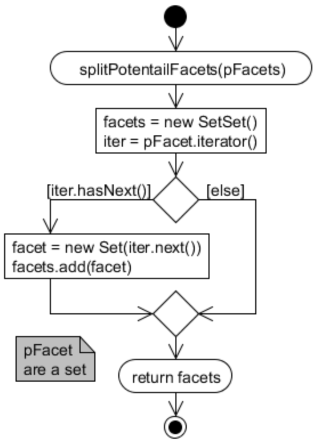
All facet merge logic is contained in the `mergeFacetsWithPair` algorithm (Figure 4.21(d)) which merges facets based on a pair. Parameters are the `facets` and the `pair`. A pair represents two sibling directory names. If the left element and the right element of a pair are contained in different facets, they are merged. Several variables need to be initialized. The variables `left` and `right` are the elements of the parameter `pair`. Corresponding to this variables, the variables `containsRight` and `containsLeft` express if some facet contains a element of the pair. They are initialized with the value false. Further there are two more variable declarations, `rightFacet` and `leftFacet`. When a facet contains an element of the pair this facet is stored in one of these variables.

[2] $Set < FacetTag > \rightarrow Set < Set < FacetTag >>$

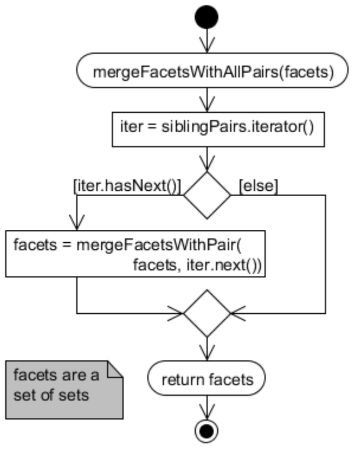


pFacets are a set, facets are a set of sets
mergeAllFacets()
iter = potentialFacetMap.keySet().iterator()
[iter.hasNext()]
[else]
context = iter.next()
pFacets = potentialFacetMap.get(context)
facets = splitPotentialFacets(pFacet)
facets = mergeFacetsWithAllPairs(
context, facets)
mergedFacetsMap.put(context, facets)

(a) facet merge main

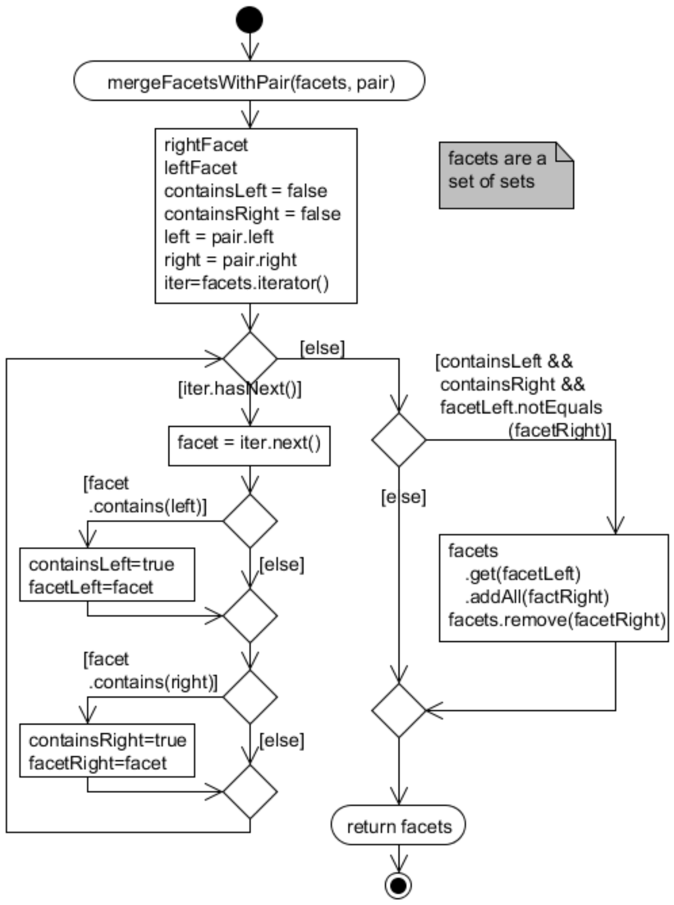


(b) facet split



mergeFacetsWithAllPairs(facets)
iter = siblingPairs.iterator()
[iter.hasNext()]
[else]
facets = mergeFacetsWithPair(
facets, iter.next())
facets are a set of sets
return facets

(c) merge facets with all pairs



(d) merge facets with pair

Figure 4.21: Facet merge algorithms.

In order to find mergable facets, the algorithm iterates over the sets of facets. Within each iteration, it is checked if the current facet contains the `left` element, then the corresponding variable `containsLeft` is set to the value `true` and the `leftFacet` is set to the facet of the current iteration step. Checking the `left` element is similar. After iterating all facets, it is checked if the set contains mergable facets. Mergable facets exist if `containsLeft` and `containsRight` is true. Additionally the facet `facetLeft` must not equals with the facet `facetRight`. In other words if both elements of a pair are contained in one facet, this facet is already merged. In case of mergable facets, all tags of the right facet are added to the left facet set. Further the right facet set is removed. The algorithm terminates and returns the may modified set of facets.

In general this algorithm needs a lot of comparisons to merge facets. Each facet must be checked for each pair element. Therefore the merge time increases rapidly with an increasing amount of sibling directories which represent the pairs. The algorithm break condition could be placed within the iteration but this will not really affect the run-time. The amount of pair comparisons which result in a merge of two facets are very low.

4.1.3.7 Facet ordering

Finally, facets are ordered are ordered in a meaningful way, in the first step the merged context facets set (Figure 4.20) are splitted in native and synthetic facets. A native facet is existing originally in the hierarchical file system. Synthetic facets are generated ones e.g. the facet {`2012, 2011`} does not exist in the context {`projects`}. The lexicographical order is applied as second priority. Figure 4.22 illustrates the ordered facets with the corresponding context. This is the final result of the facet import algorithm.

$$orderedFacetMap$$
$$< Map < Set < Context >, List < List < FacetTag >>$$

Context $\rightarrow$ {}
Facet$_1$ $\rightarrow$ {projects}^{native}

Context $\rightarrow$ {**projects**}
Facet$_1$ $\rightarrow$ {niedersachsen, bayern}^{native}
Facet$_2$ $\rightarrow$ {2012, 2011}

Context $\rightarrow$ {**projects, bayern**}
Facet$_1$ $\rightarrow$ {augsburg, münchen}^{native}

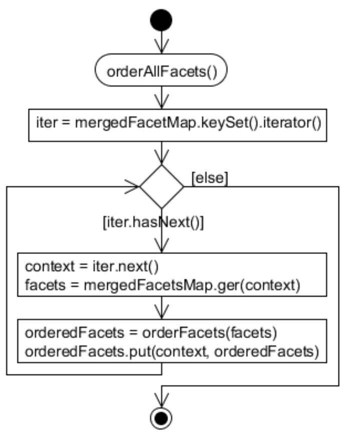
Context $\rightarrow$ {**projects, niedersachsen**}
Facet$_1$ $\rightarrow$ {braunschweig}^{native}

Figure 4.22: Ordered facets with corresponding context.

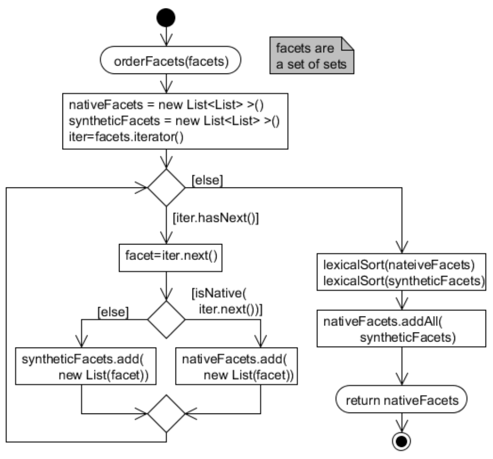
Ordering the facets is divided into three subalgorithms. Figure 4.23(a) describes the `orderAllFacets` function. This iterates over the `mergedFacetMap` once. Within each iteration the `orderFacet` algorithm (Figure 4.23(b)) is applied with the facets as parameter and returns the ordered facets for a certain context. The result is stored in the `orderedFacetMap`.

The `orderFacets` algorithm starts with initializing an empty native facets list and a synthetic facets list. For each facet of the parameter is checked if the facet `isNative` or not. In case of a native facet this facet is stored in the list of the native facets, otherwise in the list of the synthetic ones. After the iteration, the lexicographical order is applied for the native facets and the synthetic facets. That means all tags within a facet are listed in an alphabetical order and the facets are ordered in an alphabetic order. Finally the synthetic facet list is appended to the native facet list. The native facet list with all facets is returned.

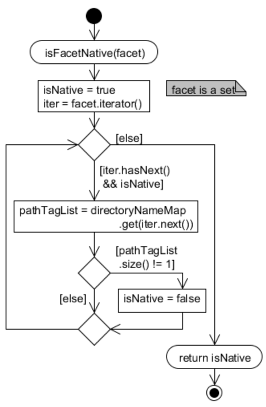
Figure 4.23(c) illustrates the `isNative` algorithm with the input parameter `facet`. For each tag of a `facet`, it is checked if the `directoryNameMap` (Figure 4.6(b)) contains this tag as key[3]. If for all facet tags exist exactly one path list, the facet is native.


orderAllFacets()
iter = mergedFacetMap.keySet().iterator()
[else]
[iter.hasNext()]
context = iter.next()
facets = mergedFacetsMap.ger(context)
orderedFacets = orderFacets(facets)
orderedFacets.put(context, orderedFacets)

(a) order all facets


orderFacets(facets)
facets are a set of sets
nativeFacets = new List<List> >()
syntheticFacets = new List<List> >()
iter=facets.iterator()
[else]
[iter.hasNext()]
facet=iter.next()
[isNative(iter.next())]
[else]
syntheticFacets.add(new List(facet))
nativeFacets.add(new List(facet))
lexicalSort(nateiveFacets)
lexicalSort(syntheticFacets)
nativeFacets.addAll(syntheticFacets)
return nativeFacets

(b) order facets



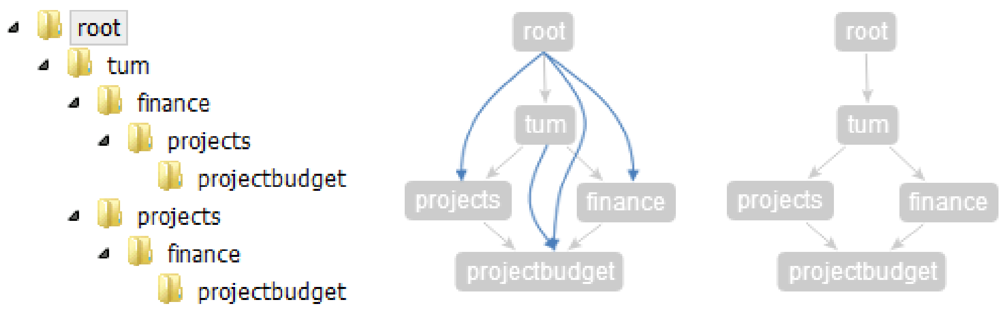
(c) is facet native

Figure 4.23: Facet order algorithms.

4.1.4 Subsumption graph algorithm

The subsumption graph expresses which tag is subsumed by another one. In general the tag `car` would subsume the tags `audi`, `daimler` and `bmw`. This graph expresses this subsumption for all tags. The tag on the root node of the graph subsumes all others. It is not necessary that only one root node exists. E.g. a resource `A` is tagged with the tag `a` and `b`. The resource `B` is also tagged with `a` and `b`. Tag `a` does not subsume tag `b` and tag `b` does not subsume `a`, this means both tags are represented as own root nodes in the graph.

[3]key = directory name



(a) hierarchical source file system (b) subsumption (c) subsumption graph

Figure 4.24: Visual subsumption graph sample.

Figure 4.24(a) illustrates the hierarchical source file system corresponding to the visualized subsumption in Figure 4.24(b). This is the output from the subsumption algorithm, see Chapter 4.1.3.2. This graph representation contains transitive edges, drawn in blue. In a small sample like this it is easy to recognize that the blue edges are transitive. In samples with more data this transitive edges reduce the readability of the graph massively. It is no more possible to find the subsumption of a tag with a quick look on the graph. One problem with more data in combination with the transitive edges is that the tags are placed totally different. The node placement algorithm does not consider these transitive edges problems, the output seems to bee messy. Therefore a graph without transitive edges (Figure 4.24(c)) is desired.

To the best of my knowledge no algorithm exists for removing transitive edges from a directed graph which scales pretty well. Figure 4.26 describes a recursive algorithm to find all not transitive edges based on the subsumption. The subsumption map in Figure 4.25(a) is used as input and in Figure 4.25(b) the resulting subsumption graph without transitive edges is represented. Each row in the subsumption graph map are the incoming edges for a tag. In other words the tag is directly subsumed by the incoming edges tags.

$$subsumptionMap$$
$$< Map < SubsumedBy, Set < Tag >>$$

Subsumed by	**Tags**
tum	{root}
projects	{root, tum}
finance	{root, tum}
projectbudget	{root, tum, projects, finance}

(a) input

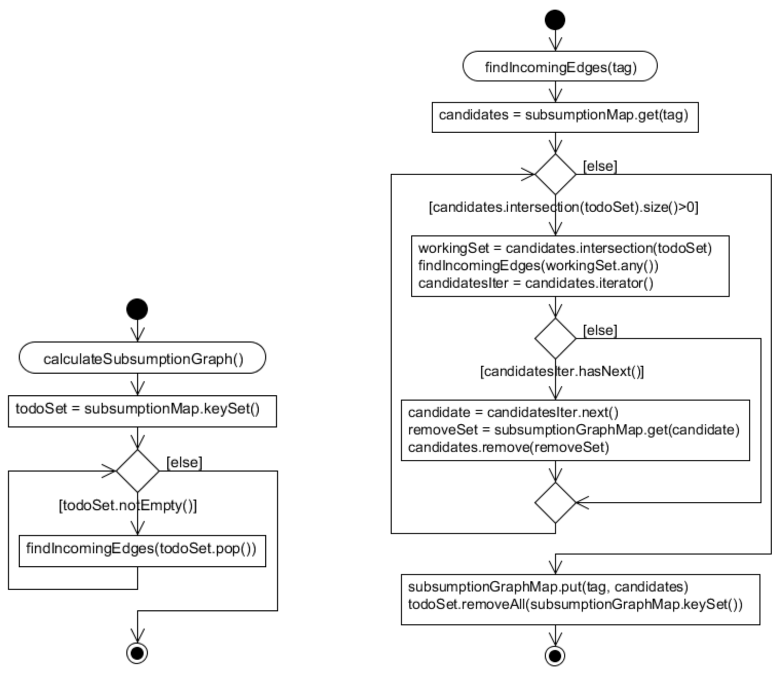
$$subsumptionGraphMap$$
$$< Map < Tag, Set < IncomingEdgeTag >>$$

Tag	**Incoming edges**
tum	{root}
projects	{tum}
finance	{tum}
projectbudget	{projects, finance}

(b) output

Figure 4.25: Subsumption graph algorithm in- and output data.

The algorithm is divided into two parts, the `calculateSubsumptionGraph` and `find IncomingEdges` algorithm. In the first step a `todoSet` is initialized with the keys of the subsumption map (Figure 4.26(a)). Any element is selected and the `findIncoming Edges` algorithm starts with these elements as parameter. Candidates for incoming edges are all tags which subsume a tag, the variable of `candidates` initialized with these. Next the intersection between all candidates and the `todoSet` is calculated. When the intersection is an empty result, the algorithm has found one or more not transitive new incoming edges. These edges are added to the `subsumptionGraphMap` and all tags for which the incoming edges are already calculated are removed form the `todoSet`. The subalgorithm terminates and returns to the main algorithm. If the `todoSet` contains any item, it begins again. In the other case when the intersection result between the `todoSet` and the `candidates` is not empty, a `workingSet` is defined. It is the intersection between the candidates and the todo set. This must be calculated new within each iteration because the `todoSet` has side effects. Any item of this working set is taken and the `findIncomingEdges` algorithm is recursively applied with the item. After this recursion is finished, an iteration over all `candidates` begins. All potential edges for this candidate are looked up in the `subsumptionMap`. All results are removed form the candidates. This prevents finding transitive edges. When the iteration over all `candidates` ends, the `workingSet` is new calculated and if it is not empty one more iteration within a recursion starts until the working set calculation is empty. In other words searching for incoming edges until all incoming edges for all parent nodes are calculated. Figure 4.27 illustrates this algorithm with sample data.

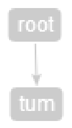

calculateSubsumptionGraph()
todoSet = subsumptionMap.keySet()
[else]
[todoSet.notEmpty()]
findIncomingEdges(todoSet.pop())
findIncomingEdges(tag)
candidates = subsumptionMap.get(tag)
[else]
[candidates.intersection(todoSet).size()>0]
workingSet = candidates.intersection(todoSet)
findIncomingEdges(workingSet.any())
candidatesIter = candidates.iterator()
[else]
[candidatesIter.hasNext()]
candidate = candidatesIter.next()
removeSet = subsumptionGraphMap.get(candidate)
candidates.remove(removeSet)
subsumptionGraphMap.put(tag, candidates)
todoSet.removeAll(subsumptionGraphMap.keySet())

(a) calculate subsumption graph (b) find incoming edges for a tag

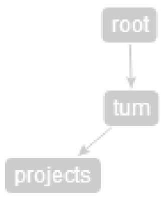
Figure 4.26: Subsumption graph algorithm.

```
todoSet:  {tum, projects, finance, projectbudget}
findIncomingEdges(projects){
      candidates:   {root, tum}
      todoSet:      {tum, finance, projectbudget}
      workingSet:   {tum}
      findIncomingEdges(tum){
            candidates:   {root}
            todoSet:      {finance, projectbudget}
            workingSet:   {}
            removeSet:    {}
            candidates:   {root}
            new edges:    tum ←{root}
      } graph 1
      removeSet:    {root}
      candidates:   {tum}
      new edges:    projects ←{tum}
}graph 2


todoSet:  {finance, projectbudget}
findIncomingEdges(projectbudget){
      candidates:   {root, tum, projects, finance}
      todoSet:      {finance}
      workingSet:   {finance}
      findIncomingEdges(finance){
            candidates:   {root, tum}
            todoSet:      {}
            workingSet:   {}
            removeSet:    {root}
            candidates:   {tum}
            new edges:    finance ←{tum}
      }graph 3
      removeSet:    {root, tum}
      candidates:   {projects, finance}
      new edges:    projectbudget ←{project, finance}

}graph 4
```

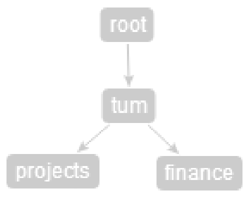
(a) Algorithm steps.



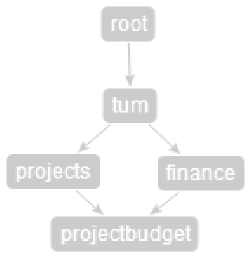
(b) graph 1



(c) graph 2



(d) graph 3



(e) graph 4

Figure 4.27: Subsumption graph algorithm sample.

4.2 Fundamental tag-based access algorithms

All fundamental tag-based access algorithms are listed and explained in this chapter. They provide some basic functionality to access directories and documents in an effective fashion. Aim is to encapsulate this basic functionality to simplify the real algorithms like the export algorithm or the CRUD operation algorithms. Furthermore they are easy to reuse. In Chapter 4.3 most of these defined algorithms are used to export tags. The create, rename, move and delete operations also use these methods (see Chapter 4.4).

4.2.1 Find context configuration

`findContextConfig(``pathTags, notPathTags`**`)`**

The context configuration provides the most basic search functionality. Based on the path tags and not path tags, the best matching context configuration is searched and returned. This context configuration contains the current context and all corresponding facets. Additionally for this context configuration are some helper functions defined. E.g. `hasFacet`, `getFacet` and `getFirstFacet`. The search algorithm of this context configuration is no more part of this work and not further described.

4.2.2 Tag-based search

`tagSearch(``mustTags, mustNotTags, type`**`)`**

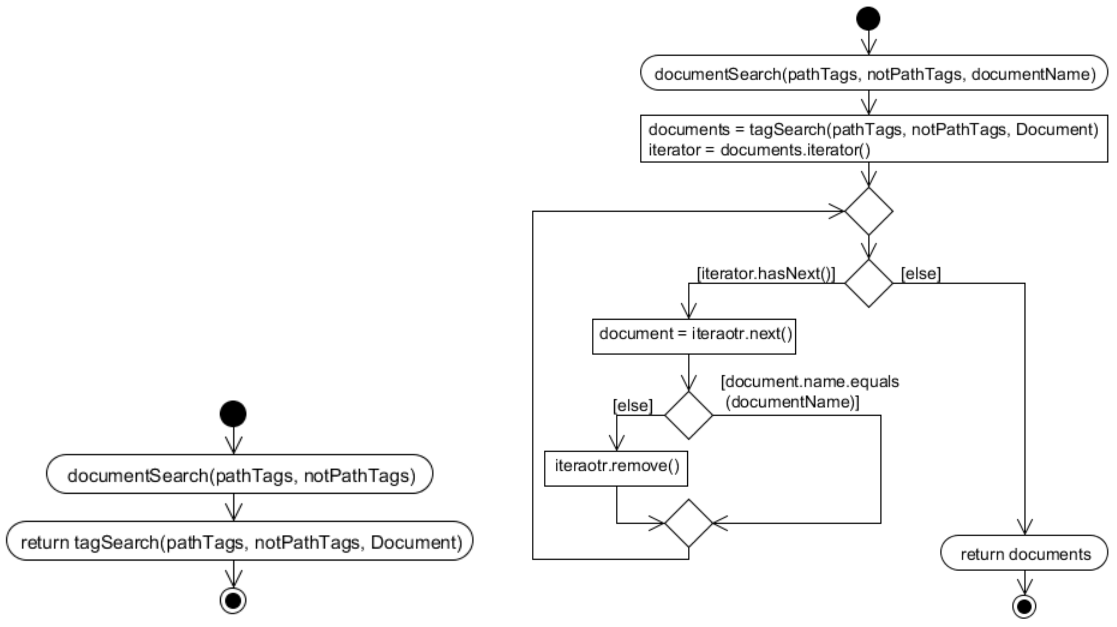
This search algorithm provides a fundamental tag-based search functionality. Depending on the `type`, the search contains directories or documents with the `mustTags` and `mustNotTags`. Finally the result set is returned. The conceptual idea behind this function is trivial, it is only listed as one of the basic functions to reference it in other algorithms. The internal logical and algorithms of this tag bases search are not part of this work.

4.2.3 Tag-based document search

`documentSearch(``pathTags`[4]`, notPathTags`[5]**`)`**
`documentSearch(``pathTags, notPathTags, documentName`**`)`**

The document search algorithm provides a tag-based search for documents and uses internal the basic `tagSearch` algorithm (see Chapter 4.2.2). Two different signatures are offered. The first one searches for all documents which are tagged with the `pathTags` (Figure 4.28(a)) and not tagged with the `notPathTags`. Second one (Figure 4.28(b)) filters the documents additionally with one iteration over the `tagSearch` result. The document name must be equal with the parameter `documentName` otherwise the document is removed for the result set. Documents are not tagged with their own document names, only with the `pathTags`, that's the reason for this name filter. The resulting documents are returned. These methods are used to export tags and to find documents for the CRUD operations.

[4]pathTags are ordered and mustTags are a set of tags.
[5]notPathTags are tags which are excluded that means a searched document does not assign these tags.


documentSearch(pathTags, notPathTags)
return tagSearch(pathTags, notPathTags, Document)
documentSearch(pathTags, notPathTags, documentName)
documents = tagSearch(pathTags, notPathTags, Document)
iterator = documents.iterator()
[iterator.hasNext()]
[else]
document = iteraotr.next()
[document.name.equals (documentName)]
[else]
iteraotr.remove()
return documents

(a) search for all documents (b) search for documents with certain name

Figure 4.28: Tag-based document search.

4.2.4 Tag-based directory search

Tag-based directory search provides likewise the document search (see Chapter 4.2.3) also two signatures. First one (see Chapter 4.2.4.1) returns all tag-based matching search results, the second one (see Chapter 4.2.4.2) applies a name filter.

4.2.4.1 Simple tag-based directory search

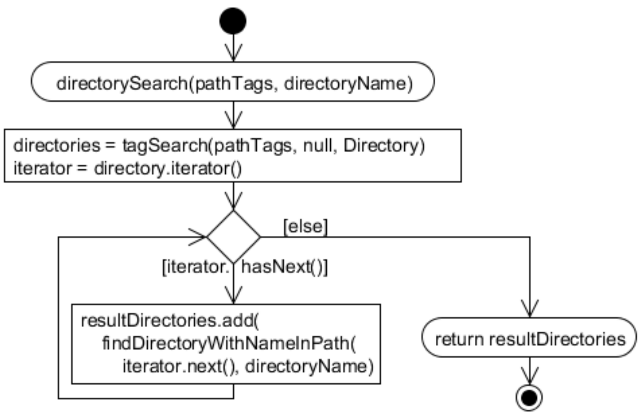
`directorySearch(``pathTags`**`)`**

The simple tag-based directory search algorithm search for all directories which are tagged with the `pathTags`. This is basically only a forward to the `tagSearch`, only two of the three parameters is really used. The forward look like **`tagSearch`**`(pathTags, null, Directory)`. This simple search does not apply any filter on the search result. It is directly returned. Additionally the `directorySearch` offers a second signature which allows to search for a certain directory within the tags (see Chapter 4.2.4.2).

4.2.4.2 Tag-based directory search with certain name

directorySearch(`pathTags, directoryName`**)**

The tag-based directory search with this signature (Figure 4.29) searches for all directories which are tagged with the `pathTags` and named like the parameter `directoryName`. For the basic tag-based search the `tagSearch` is used. Within one iteration over the result set, the algorithm `findDirectoryWithName` (see Chapter 4.2.4.3) is applied for each result directory and stores the may modified directory to the final result set. Probably one directory is more than once added to the final result set but it does not matter it is a set of directories. Finally the set of resulting directories is returned.



directorySearch(pathTags, directoryName)
directories = tagSearch(pathTags, null, Directory)
iterator = directory.iterator()
[else]
[iterator.hasNext()]
resultDirectories.add(
findDirectoryWithNameInPath(
iterator.next(), directoryName)
return resultDirectories

Figure 4.29: Tag-based directory search with name.

Figure 4.30 illustrates a common directory search. All directories which are tagged with `projects` and `lectures` and named `lectures` are searched. In the first step the tag-based search is executed and returns several results. Subdirectories of the searched directory also tagged with the searched tags and within the result set. Desirable are only folders which are named like the parameter `directoryName`. For each resulting directory, the algorithm `findDirectoryWithNameInPath` (Figure 4.31) is applied. In this sample the final results are `\projects\lectures\` and `\projects\org\lectures\`.

```
directorySearch({projects, lectures}, lectures)

          tagSearch({projects, lectures}, {}, Directory)
           return →   \projects\lectures\
                      \projects\lectures\2010\
                      \projects\lectures\2011\
                      \projects\lectures\2012\
                      \projects\org\lectures\

 return →   \projects\lectures\
            \projects\org\lectures\
```

Figure 4.30: Sample directory tag-search with certain name.

4.2.4.3 Find directory with name in path

findDirectoryWithNameInPath(directory, name**)**

Figure 4.31 illustrates this algorithm. The algorithm searches for the directory with the `name` in the path of the `directory`. If the name is not equal to searched `name` this algorithm is applied recursively with the parent directory. The algorithm terminates only when the directory with the right `name` is found, no other break condition is necessary. Each directory which is tagged with a name must contain this tag as directory. The resulting `directory` is returned. Figure 4.32 illustrates the algorithm with sample data. The directory with the name `bayern` is searched in the path `\projects\bayern\münchen\2011\`. Within the first call the directory name of the path does not match with the parameter `name`. The algorithm is applied recursively with the parent directory until the `name` match.

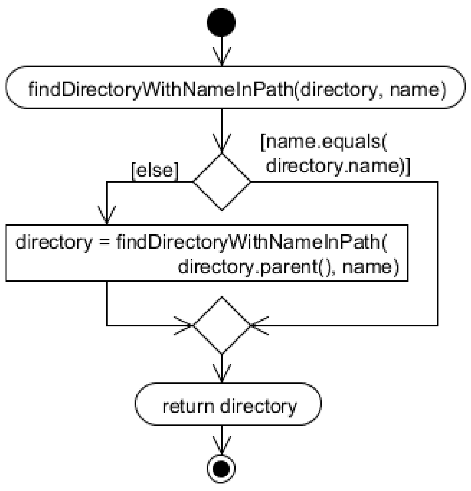

findDirectoryWithNameInPath(directory, name)
[name.equals(directory.name)]
[else]
directory = findDirectoryWithNameInPath(directory.parent(), name)
return directory

Figure 4.31: Find directory with name in path algorithm.

```
findDirectoryWithNameInPath(\projects\bayern\münchen\2011\, "bayern"){
    findDirectoryWithNameInPath(\projects\bayern\münchen\, "bayern"){
        findDirectoryWithNameInPath(\projects\bayern\, "bayern"){
            return →\projects\bayern\
        }
        return →\projects\bayern\
    }
    return →\projects\bayern\
}
```

Figure 4.32: Find directory with name in path algorithm sample.

4.2.5 Find or create directory

findOrCreateDirectory(`pathTags`**)**

Several methods need to find a physical path for a certain context. Not all navigation paths are also existing directly in the hierarchical source file system. Create and move operations (see Chapter 4.4.1, 4.4.3) use this algorithm. This `findOrCreateDirectory` algorithm is divided into two algorithms. Searching for the best matching already existing directory (see Chapter 4.2.5.1) and creating not existing directories (see Chapter 4.2.5.3). In the first step the recursive `findExistingDirectory(pathTags, directory)` is executed. Parameters are the corresponding `pathTags` and the file system root `directory`. It results in a `directory` with the corresponding `leftPathTags`. The second algorithm `createDirectoryForTags(leftTags, existingDirectory)` is executed based on the results of the first one. Finally a directory which contains all tags in the path is returned.

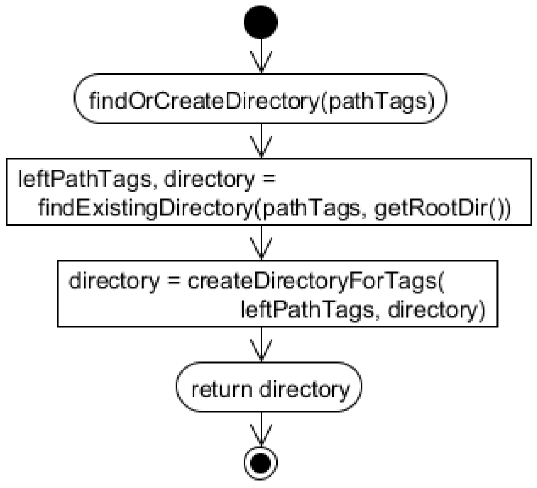

findOrCreateDirectory(pathTags)
leftPathTags, directory =
findExistingDirectory(pathTags, getRootDir())
directory = createDirectoryForTags(
leftPathTags, directory)
return directory

Figure 4.33: Find or create directory algorithm.

4.2.5.1 Find existing directory

findExistingDirectory(`leftPathTags, directory`**)**

Aim of this algorithm is to find based on the `directory`, for the best matching child directory. The parameter `leftPathTags` represent the tags which are not covered within the `directory` path. In gerneral a directory which contains more tags in the path matches better. That means on the other and the list of `leftPathTags` is as small as possibel. The represented algorithm performs a deep search first in a graph. All Subdirectories of the current directory are iterated. Within these iteration the `subPathTags` for the `subDirectory` are defined. Removing the current `directory` name form the `leftPath Tags` results in the `subPathTags`. When the list of `subPathTags` is shorter as the pathTags list, then a new potential matching directory is found. In this case the algorithm bestMatchingDirectory (see Chapter 4.2.5.2) is executed with the current leftPathTag and directory the second pair of parameters is recessively calculated with the findExistingDirectory(subPath Tags, subDirectory) algorithm. It returns the best matching pathTags and the corresponding best matching directory, they override the existing variable pathTags and directory. The iteration continues until no more subdirectory exists. Finally the directory which contains the most tags is returned.

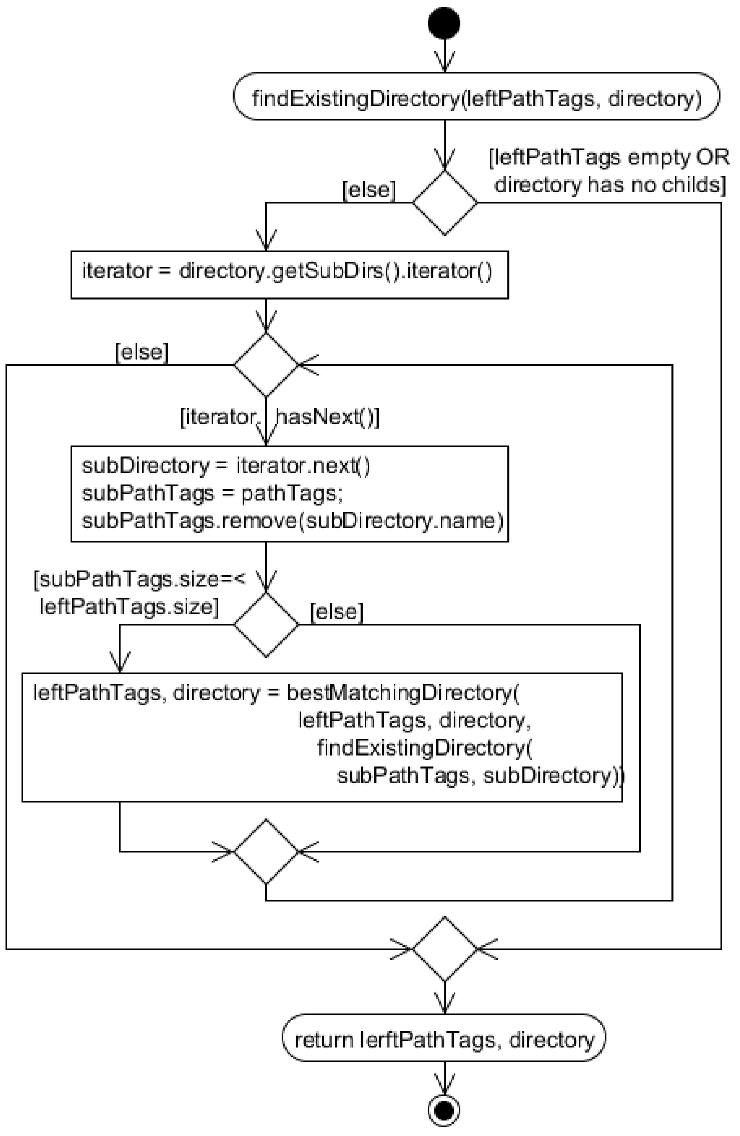

findExistingDirectory(leftPathTags, directory)
[leftPathTags empty OR directory has no childs]
[else]
iterator = directory.getSubDirs().iterator()
[else]
[iterator. hasNext()]
subDirectory = iterator.next()
subPathTags = pathTags;
subPathTags.remove(subDirectory.name)
[subPathTags.size=< leftPathTags.size]
[else]
leftPathTags, directory = bestMatchingDirectory(leftPathTags, directory, findExistingDirectory(subPathTags, subDirectory))
return lerftPathTags, directory

Figure 4.34: Find existing directory algorithm.

4.2.5.2 Best matching directory

```
bestMatchingDirectory(leftPathTagsA, directoryA,
                      leftPathTagsB, directoryB)
```

This algorithm (Figure 4.35) returns the best matching directory of `directoryA` and `directoryB`. Directories with the smaller list of `leftPathTags` match better. That means most tags are already covered in the directory path and no more contained in the `leftPath Tags`. In case of an equal length of both lists, the directory name is considered. The directory with the first name in the alphabetic order represents the best matching. This is defined and could be changed, to ensure a second execution with the same parameters will result in the same directory. The best matching `directory` with the corresponding `leftPathTags` is returned.

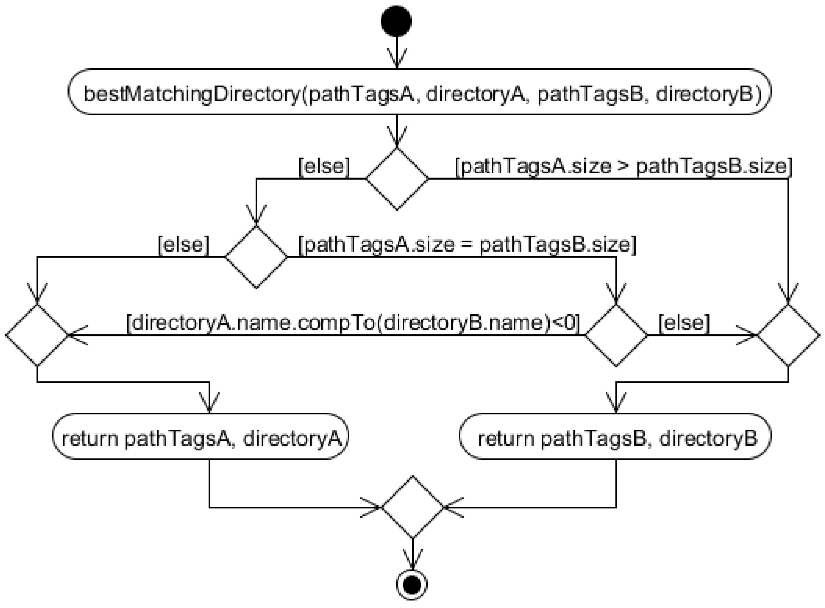

bestMatchingDirectory(pathTagsA, directoryA, pathTagsB, directoryB)
[else]
[pathTagsA.size > pathTagsB.size]
[else]
[pathTagsA.size = pathTagsB.size]
[directoryA.name.compTo(directoryB.name)<0]
[else]
return pathTagsA, directoryA
return pathTagsB, directoryB

Figure 4.35: Best matching directory.

4.2.5.3 Create directory for tags

`createDirectoryForTags(``tags, directory`**`)`**

Figure 4.36 illustrates the create directory for tags algorithm. This algorithm works recursively until the `directory` path contains all `tags`. When the list of `tags` is not empty a new directory is created, otherwise the algorithm terminates and the current directory is returned. First all tags are sorted in a descending alphabetic order. Sorting the `tags` is necessary to ensure a predictable behavior. It keeps the data structure clean. The last tag is removed from the list and a sub directory is created in the `directory` with the name of the removed tag. The algorithm is applied recursively with the left `tags` an the new created `directory`.

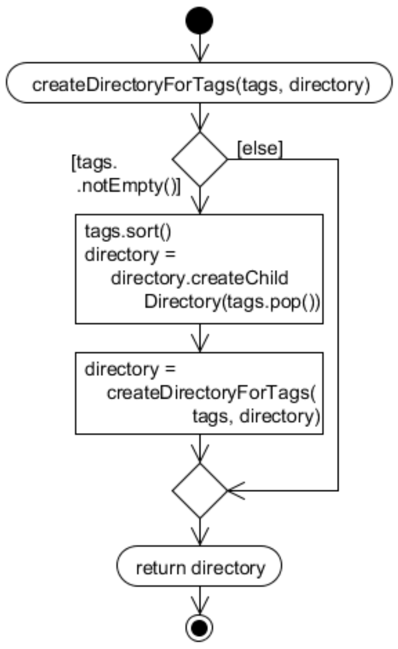


Figure 4.36: Create directory for tags algorithm.

4.3 Export tags

In general a tag export offers a hierarchal representation of tags. This export based on the previously imported context configurations. Two different concepts of exports are supported, native tag-based and multifaceted tag-based. These navigation concepts are explained in chapter 3.2.1 and 3.2.2. The following chapters illustrate the tag-based export on a conceptual level. Optional functionality like the document counting is not considered. The basic concept is the same but there are a lot more cases and search requests. Input of the export algorithm is a path, which represents a context. The following chapter 4.3.1 describes all really basic functions for the tag export.

4.3.1 Export tags utility methods

Get path tags

`getPathTags(``path`**`)`**

The input of this function is a tag `path` the structure is like a path from a hierarchical file system. Each `path` section represents directory which is separated with a backslash. This path is split by the backslash and all sections beginning with a `group by` are removed. The result is returned in the same order as the path contains the sections.

```
getPathTags(\projects\group by 2011, 2012\2011\group by bayern, niedersachsen\)
     return →  [projects, 2011]
```

Generate group by directory name

`generateGroupByName(``facet`**`)`**

A `facet` is the input of the generate group by name function. The result name is initialized with the `group by` prefix. Within one iteration over the facet each facet tag is appended to the result. When the iteration has more elements than additionally a comma and a space is appended to the result name, otherwise the result is returned.

```
generateGroupByName([2010, 2011, 2012])
     return →  "group by 2010, 2011, 2012"
```

Parse group by facet

`parseGroupByFacet(``path`**`)`**

This is the inverse function to the `generateGroupByName` function. Input is a `path`, the last section must contain a group by directory. The last path segment is selected and the `group by` is removed. In the next step the result is split by the comma and space. The resulting facet is returned.

```
parseGroupByFacet(\projects\group by 2010, 2011, 2012\)
     return →  [2010, 2011, 2012]
```

Is path group by directory

`isPathGroupBy(``path`**`)`**

Input of this function is a tag `path`. This function checks if the last `path` section is a `group by` directory, in this case it returns `true` otherwise `false`.

```
isPathGroupBy(\projects\group by 2010, 2011, 2012\)
     return →  true
```

4.3.2 Native tag-based export

nativeTagExport(`path`**)**

Native tag-based export is the default export option. All necessary export information is contained in the input parameter `path`. Based on this path the content of these tag-based directory is calculated (Figure 4.38). In the first step the `pathTags` are extracted from the path with the `getPathTags` function (see Chapter 4.3.1). The corresponding context configuration is searched based on the `pathTags`. Depending on the context the tag-based `directoryNames` are defined. If there exists at least one facet in the context, the first facet is set as directoryNames. Otherwise a `tagSearch` is executed with the `pathTags` as `mustTags`. The set of all in the search result containing tags is set as `directoryNames`. Tags which are already in the `path` must be removed, otherwise the hierarchical structure will never end. All `pathTags` are removed from the `directoryNames`. Now the tag-based directory names are defined for both cases. In the next step all documents which are tagged with the `pathTags` and not tagged with the defined `directoryNames` are searched with the `documentSearch` algorithm (see Chapter 4.2.3). Finally all `directoryNames` are added as sub directory to the path and all documents are added as well to the current path.

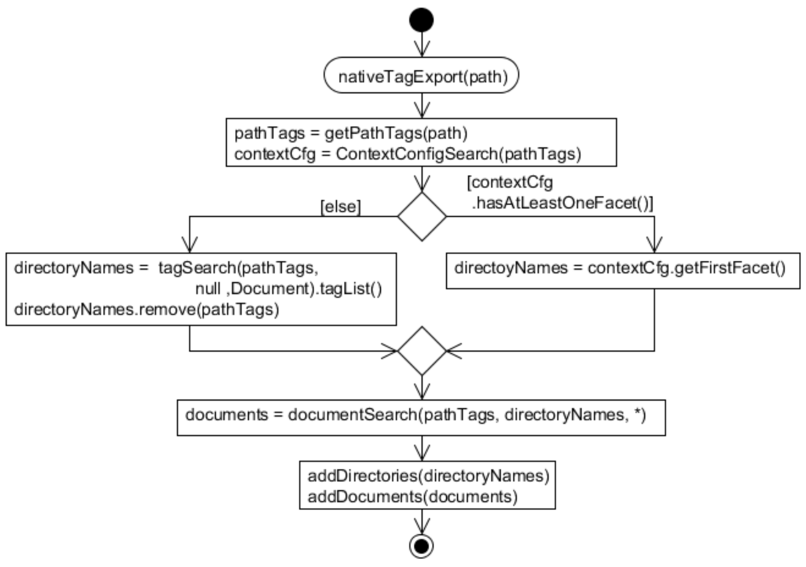

nativeTagExport(path)
pathTags = getPathTags(path)
contextCfg = ContextConfigSearch(pathTags)
[else]
[contextCfg .hasAtLeastOneFacet()]
directoryNames = tagSearch(pathTags, null ,Document).tagList()
directoryNames.remove(pathTags)
directoyNames = contextCfg.getFirstFacet()
documents = documentSearch(pathTags, directoryNames, *)
addDirectories(directoryNames)
addDocuments(documents)

Figure 4.37: Native tag export.

4.3.3 Multifaceted tag-based export

groupbyTagExport(`path`**)**

The multifaceted tag export (Figure 4.38) is an extension of the native one (see Chapter 4.3.2). In the first step the tags from the `path` are extracted with the `getPathTags(path)` function (see Chapter 4.3.1) and stored as `pathTags`. Three different main cases are treated. In the first case the current `path` is a `group by` path (see Chapter 4.3.1). In the second case it is not a `group by` path and the path corresponding context has more than one facet. The Third case is as well not a `group by` path and the related context has only one or no facet.

In the first case, the required facet is contained in the `path`. This facet is parsed with the function `parseGroupByFacet(path)` (see Chapter 4.3.1) and the resulting facet is set as `directoryNames`. Related to this facet all documents are searched with the `documentSearch` algorithm (see Chapter 4.2.3). Documents which are tagged with the `pathTags` and not tagged with the `directoryNames` are searched. All resulting documents are directly added to the corresponding path. Finally all `directoryNames` are added as directories to the current `path` and represent the facet as directories.

The second and third case use the `pathTags` to find a context configuration (see Chapter 4.2.1). If there exists only up to one facet in the context, the third case occurs and the `nativeTagExport(path)` (see Chapter 4.3.2) is executed. Otherwise it is the third case and more than one facet exists in the context. That means the next directory level will be a synthetic one with no documents. For each `facet` of the context one `group by` directory name is generated with the `generateGroupByName` function (see Chapter 4.3.1). All these directory names are added as directories to the current path.

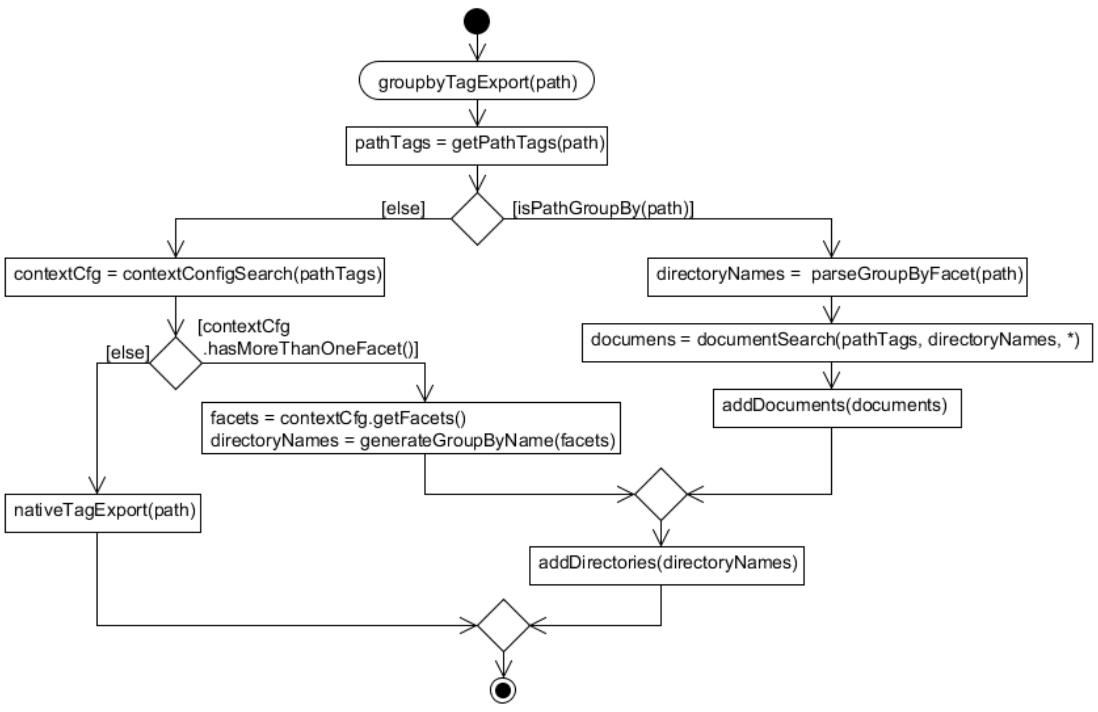

groupbyTagExport(path)
pathTags = getPathTags(path)
[else]
[isPathGroupBy(path)]
contextCfg = contextConfigSearch(pathTags)
directoryNames = parseGroupByFacet(path)
[contextCfg .hasMoreThanOneFacet()]
[else]
documens = documentSearch(pathTags, directoryNames, *)
addDocuments(documents)
facets = contextCfg.getFacets()
directoryNames = generateGroupByName(facets)
nativeTagExport(path)
addDirectories(directoryNames)

Figure 4.38: Multifaceted tag export.

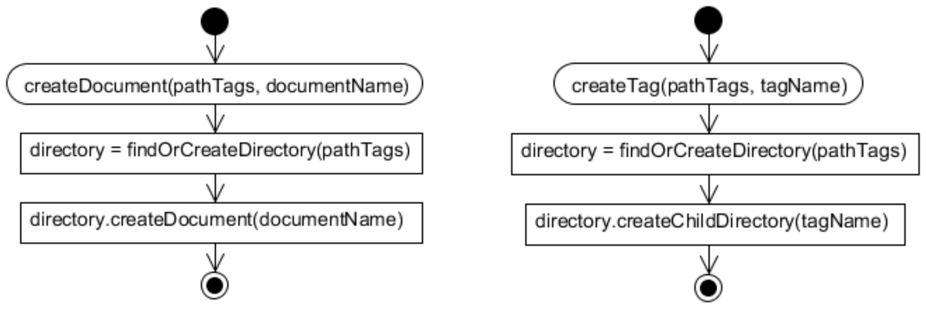
4.4 CRUD operation mapping algorithms

The user interface provides tag-based views and the corresponding operations. Internal all this tag-based data is stored within a hierarchical file system. Therefore each tag-based interaction needs to be assigned back to the hierarchical source file system interaction. This chapter defines all necessary algorithms to map the fundamental crud operations. All algorithms are described on a conceptual level. Error handling and abort conditions are not covered here. This chapter is structured related to the crud operations, create, rename, move and delete. Each subchapter covers a document and tag-based operation.

All operations which are executed on the native tag-based view are normally easier to map, it does not matter with or without count option. Executing operations on synthetic facets cause some issues. Synthetic facet tags normally exist more than once in the hierarchical source file system. Without any extra algorithm definition it is not clear which source should be selected. Chapter 4.2 describes some of these algorithms, they are used for all mapping operations.

In order to maintain consistency during each file system modification the tags must be updated as well. Documents can be tagged with more tags and not only with the path corresponding tags. This is a special case but it can happen. When the path of such a document changes only the tags which are related to, the changed directory should be updated. Tags which are not related with the operation shouldn't be touched.

4.4.1 Create Operation Mapping


createDocument(pathTags, documentName)
directory = findOrCreateDirectory(pathTags)
directory.createDocument(documentName)
createTag(pathTags, tagName)
directory = findOrCreateDirectory(pathTags)
directory.createChildDirectory(tagName)

(a) Create document (b) Create tag

Figure 4.39: Create operation algorithms.

4.4.1.1 Create document

createDocument(`pathTags, documentName`**)**

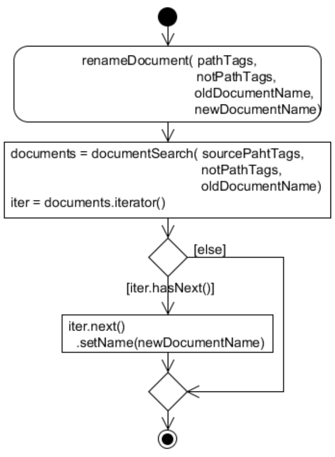
This algorithm (Figure 4.39(a)) map a tag-based document create operation to a common hierarchical file system create document operation. Aim of this algorithm is to find an already existing directory for the `pathTags`. Some times it is not possible to find an already existing directory, in that case new directories are created. For this algorithm it is transpreant whether an directory exists or not. The fundamental algorithm (see Chapter 4.2.5) `findOrCreateDirectory` covers this task. After the tag related directory is found or created, a child document with the name `documentName` is created.

4.4.1.2 Create tag

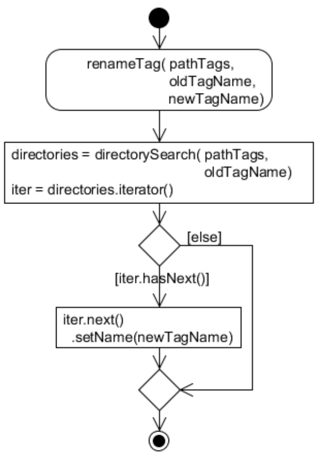
`createTag(``pathTags, tagName`**`)`**

Creating a tag based on a hierarchical file system means creating a folder with the `tagName`. In general the concept is similar to the create document operation (see Chapter 4.4.1.1). Finally a child directory with the `tagName` is created instead of a document. Figure 4.39(b) illustrates the algorithm.

4.4.2 Rename operation mapping


renameDocument(pathTags, notPathTags, oldDocumentName, newDocumentName)
documents = documentSearch(sourcePahtTags, notPathTags, oldDocumentName)
iter = documents.iterator()
[else]
[iter.hasNext()]
iter.next() .setName(newDocumentName)

(a) Rename document


renameTag(pathTags, oldTagName, newTagName)
directories = directorySearch(pathTags, oldTagName)
iter = directories.iterator()
[else]
[iter.hasNext()]
iter.next() .setName(newTagName)

(b) Rename tag

Figure 4.40: Rename operation algorithms.

4.4.2.1 Rename document

`renameDocument(``pathTags, notPathTags, oldDocName, newDocName`**`)`**

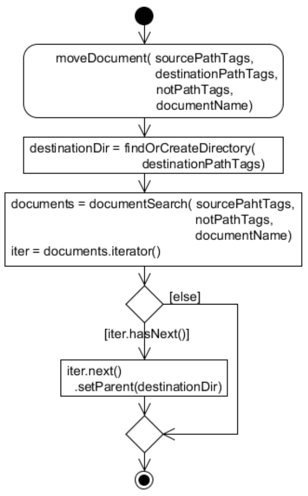
Execute a tag-based document rename operation means to search for all documents which are tagged with the `pathTags` and not tagged with the `notPathTags`. Documents are not tagged with its own documentname. Therefore the search must be filtered with the `oldDocName`. All this is done by the fundamental `documentSearch` (see Chapter 4.2.3) algorithm. In the final step one iteration over the document result set renames all documents which are named like the parameter `oldDocName` to the value of the parameter `newDocName`. (Figure 4.40(a)).

4.4.2.2 Rename Directory

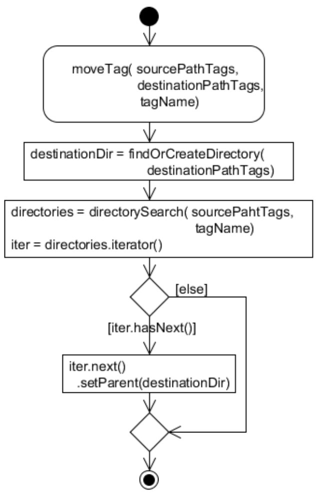
renameTag(`pathTags, oldTagName, newTagName`**)**

The rename directory algorithm (Figure 4.40(b)) searches for all directories which are tagged with the `pathTags`. This results in a set of directories which contain the `oldTag Name` in one path section. For each directory of the result set the first parent directory with the `oldTag Name` is searched. The fundamental `directorySearch` function provides this functionality (see Chapter 4.2.4). Within a iteration the name of all resulting directories is set to the `newTagName`.

4.4.3 Move operation mapping


moveDocument(sourcePathTags,
destinationPathTags,
notPathTags,
documentName)
destinationDir = findOrCreateDirectory(
destinationPathTags)
documents = documentSearch(sourcePahtTags,
notPathTags,
documentName)
iter = documents.iterator()
[else]
[iter.hasNext()]
iter.next()
.setParent(destinationDir)

(a) move document


moveTag(sourcePathTags,
destinationPathTags,
tagName)
destinationDir = findOrCreateDirectory(
destinationPathTags)
directories = directorySearch(sourcePahtTags,
tagName)
iter = directories.iterator()
[else]
[iter.hasNext()]
iter.next()
.setParent(destinationDir)

(b) move tag

Figure 4.41: Move operation algorithms.

4.4.3.1 Move document

moveDocument(`sourcePathTags, notPathTags, destinPathTags, name`**)**

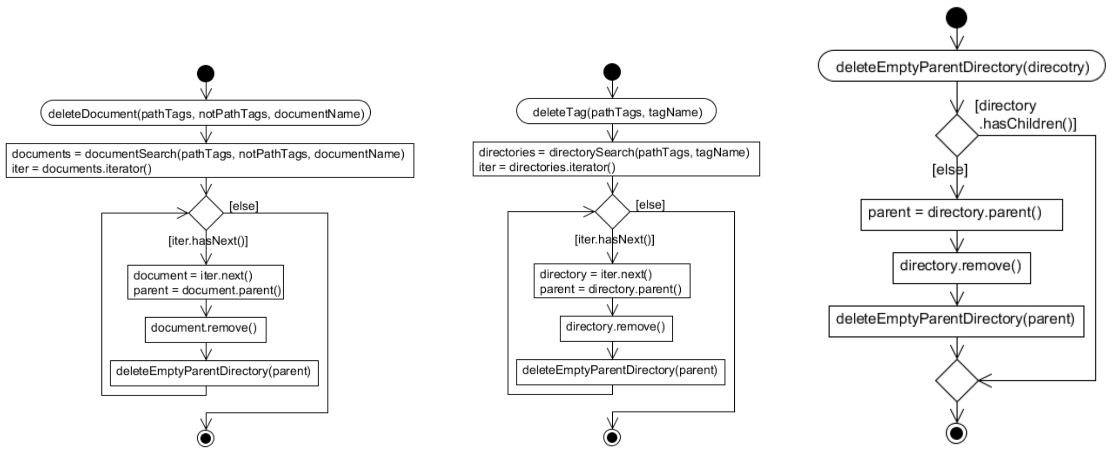
The Move document algorithm is illustrated in Figure 4.41(a). In the first step a destination directory corresponding to the `destinPathTags` is searched. Such a directory need not necessarily exist in the hierarchical source file system. In case of a not native facet tag this probably occurs. This search and create operation is encapsulated in the `findOrCreateDirectory` algorithm (see Chapter 4.2.5). In the second step all source documents are searched tag-based. The `documentSearch` algorithm (see Chapter 4.2.3) is executed with the parameters `sourcePathTags`, `notPathTags` and the parameter `name`. This results in a set of source documents. In the final step one iteration over the resulting documents, sets for each document the parent directory the searched destination directory.

4.4.3.2 Move directory

moveTag(`sourcePathTags, destinationPathTags, name`**)**

Figure 4.41(b) illustrates the move tag operation. In the first step, similar to move document operation (see Chapter 4.4.3.1), the destination directory is searched corresponding to the `destinationPathTags`. With the `directorySearch` algorithm (see Chapter 4.2.5) all source directories corresponding to the `sourcePathTags` and `name` are searched. One iteration over the resulting directories, sets for each directory the parent directory to the destination directory.

4.4.4 Delete Operation Mapping



deleteDocument(pathTags, notPathTags, documentName)
documents = documentSearch(pathTags, notPathTags, documentName)
iter = documents.iterator()
[else]
[iter.hasNext()]
document = iter.next()
parent = document.parent()
document.remove()
deleteEmptyParentDirectory(parent)
deleteTag(pathTags, tagName)
directories = directorySearch(pathTags, tagName)
iter = directories.iterator()
[else]
[iter.hasNext()]
directory = iter.next()
parent = directory.parent()
directory.remove()
deleteEmptyParentDirectory(parent)
deleteEmptyParentDirectory(direcotry)
[directory
.hasChildren()]
[else]
parent = directory.parent()
directory.remove()
deleteEmptyParentDirectory(parent)

(a) delete document (b) delete tag (c) delete empty parent directory

Figure 4.42: Delete operation algorithms.

4.4.4.1 Delete document

deleteDocument(`pathTags, notPathTags, documentName`**)**

Figure 4.42(a) represents the delete document algorithm. First of all, documents with the `pathTags` and `notPathTags` are searched and filtered with the `documentName`. This is done with with the `documentSearch` algorithm (see Chapter 4.2.3) . Afterwards the result set is iterated. Within each iteration the parent directory is stored in a local variable and the document is removed. To ensure that this operation does not result with an empty directory the `deleteEmptyParentDirectory` algorithm (see Chapter 4.4.4.3) is executed after each document removal.

4.4.4.2 Delete directory

`deleteTag(``pathTags, tagName`**`)`**

The delete tag algorithm (Figure 4.42(b)) searches for all directories which are tagged with the `pathTags` and named with the `tagName`. This search is executed with the `directorySearch` algorithm (see Chapter 4.2.4). In the final step one iteration over the result set removes all `tagName` corresponding directories. Before each removal the current directory is stored in a local variable and after directory removal, the `deleteEmptyParent Directory` algorithm (see Chapter 4.4.4.3) is executed with the locally stored directory. This prevents leaving empty directories.

4.4.4.3 Delete empty parent directory

`deleteEmptyParentDirectory(``directory`**`)`**

Deleting empty folders can improve the navigation concept. It is not the aim to find all empty folders in the hierarchical system that would not only have positive effects. E.g. within a synthetic facet tag, a create document operation is executed and with the next operation the document is removed. The new created directory would still exist without any content and affect the facet import algorithm negative. Another point is the hierarchical file structure would get messy after a certain time. When a subfolder is removed and the parent one does not contain any children, then the parent one is as well removed. Removing a document can cause the same problem. The delete empty parent directory algorithm (Figure 4.42(c)) is also applied. As parameter the current `directory` is handed over. When the `directory` is empty the parent one is stored in a local variable. In the next step the current `directory` is removed. The parent directory is handed over to itself recursively until the `directory` contains some children, a document or a directory.

5 Prototypical Implementation

This chapter illustrates the prototypical implementation of the TACKOFiles plugin[1]. First of all, the integration environment is briefly described (see Chapter 5.1). Further the architecture is explained (see Chapter 5.2). Finally the most important parts of the prototypical implementation are illustrated (see Chapter 5.3).

5.1 Integration Environment

The integration environment is described in this Chapter. In general the whole integration is based on the tricia platform with the TACKO model (see Chapter 5.1.1). Additionally all external used frameworks are covered in Chapter 5.1.2. The characteristics of all frameworks are illustrated briefly.

5.1.1 Tricia

Tricia is a "[...] commercial web-based enterprise collaboration platform [...]" [MNS12] developed and marked by the infoAsset AG[2].

> "Tricia provides an integrated enterprise collaboration and information management solution and allows to manage content items such as blog posts, wiki pages, and shared files. All these content items can be tagged with arbitrary text labels. Characteristic for the platform is its plugin architecture and its extensibility." [MNS12]

Aim of this work is to develop a TACKOFiles plugin which allows browsing files tag-based. The new plugin extends the existing TACKO model. All imported data form the hierarchical file system is stored in the persistence layer of the TACKO model. One advantage is all data can be easily accessed with the advanced TACKO multifaceted navigation.

5.1.2 External Frameworks

A web 2.0 application is usually built with jQuery[3], a common JavaScript Library.

> "jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development." [jF12]

All libraries described in subchapter 5.1.2.1 to 5.1.2.4 are jQuery plugins or they are using jQuery.

[1]TACKOFiles is an acronym for coupling file and tag-based context dependent knowledge organization.
[2]http://www.infoasset.de/ (accessed 15th of June 2012).
[3]http://jquery.com/, (accessed 11th of June 2012)

5.1.2.1 jQuery Splitter Plugin

The jQuery Splitter Plugin[4] split a component like a Java AWT SplitPanel. In this case it is an HTML `div` element. Different options are supported e.g. horizontal or vertical splitting. Panels can be divided in more than two parts, by default a panel is split in two parts. Parts are separated with a splitbar, while using default settings this splitbar can be moved and resize the size of parts. In this Implementation a horizontal resizable default splitter is used. In the left and in the right part a dynatree (see Chapter 5.1.2.2) is embedded.

5.1.2.2 jQuery Dynatree Plugin

The jQuery Dynatree Plugin[5] is a web-based visualisation for hierarchical filesystems. Files and folders are represented as trees similar to the windows explorer. Each folder or file node has certain parameters e.g. `title`, `key`, `isFolder`, `isLazy`, `isExpanded`. The `title` represents the file- or foldername. To identify resources clearly, each node has additionally a parameter `key`. Users navigate by clicking on files or folders. On a click event a JSON request is sent to the Server. Each request contains the path of keys from the current active node. On response all childnodes contained in the json object are added to the active node. There are also different loading options offered, e.g. load from JavaScript Objects or load nodes from file. Other more advanced operations are provided with a context menu (see Chapter 5.1.2.3).

5.1.2.3 jQuery Context Menu Plugin

The jQuery Context Menu Plugin[6] binds a context menu to HTML elements. In this case it is a `span` which represents the node of the DynatreePlugin (see Chapter 5.1.2.2). Right click on this `span` element opens the context menu at the click event position. A context menu contains as many action items as defined, they are easy to extend. Each context menu action is represented by one item in a HTML list. Every Action is bound to certain self-defined JavaScript function. Left click on an action item triggers the bound function and executes the selected operation, the context menu closes automatically.

5.1.2.4 Arbor Graph Library

> ”Arbor is a graph visualization library built with web workers and jQuery. Rather than trying to be an all-encompassing framework, arbor provides an efficient, force-directed layout algorithm plus abstractions for graph organization [...]” [DC11]

The Abor Graph Library[7] draws both directed and undirected graphs. Nodes and Edges are added as json objects with the method `graft(data))` or manually with `addNode (name, data)` and `getEdges(source, target)`. Figure 5.1 represents some json

[4]http://methvin.com/splitter/, (accessed 11th of June 2012)
[5]http://code.google.com/p/dynatree/, (accessed 11th of June 2012)
[6]http://abeautifulsite.net/2008/09/jquery-context-menu-plugin/, (accessed 11th of June 2012)
[7]http://arborjs.org/, (accessed 11th of June 2012)

example data. The edges in the json data are already nested in the right order. Adding nodes and edges one by one takes much more time. Using the first method causes a huge performance advantage. Reasonable is the nested json data structure, which contains the edges in the right order. After adding a new node, the library moves the graph nodes to prevent unnecessary edges intersections. There are several parameters to influence this ordering mechanism.

```
{
    "nodes": {
        "munich": {"label": "munich"},
        "bavaria": {"label": "bavaria"},
        "2012": {"label": "2012"},
        "2011": {"label": "2011"}
    },
    "edges": {
        "bavaria": {"munich": {"directed": true}},
            "2012": {},
            "2011": {}
        },
        "munich": {
            "2012": {"directed": true},
            "2011": {"directed": true}
        }
    }
}
```

Figure 5.1: JSON graph example data.

5.1.3 Used Licences

Figure 5.2 lists all used software licences. All in the previous chapter described plugins and frameworks are listed with the version number and the corresponding licence. Exists more than one, all licences are listed.

Framework	**Version**	**License**
Tricia	3.2.2	commercial
jQuery	1.7.1	MIT, GPL
jQuery Splitter Plugin	1.51	MIT, GPL
jQuery Dynatree Plugin	1.2.0	MIT, GPL Version 2
jQuery Context Menu Plugin	1.01	MIT, GPL
jQuery Arbor Plugin	0.91	MIT

Figure 5.2: Used licences.

5.2 Architecture

The architecture of the TACKO Files plugin and the dependencies to other used plugins are illustrated in Chapter 5.2.1. Furthermore, in Chapter 5.2.2 the internal packages are presented and package dependencies are described.

5.2.1 Plugin overview

This chapter describes briefly the general plugin structure. Tricia supports all core functionality and provides the opportunity for extensions. Each extension is represented as a plugin. Figure 5.3 illustrates the main dependencies between the plugins. The `Tacko FilesPlugin` is the result of this work. It has two main dependencies, to the `Tricia Platform` and to the JLan2 plugin. In this context essential components of the platform are the `asset.file` and `asset.serach`. All directories and documents are encapsulated in the file component. The `search` package provides searching all types of content within the system tag-based or conventional. Accessing all directories and documents via a mounted network device supports the `JLan2 Plugin`. In chapter 5.2.2 the structural concept of the `TackoFiles Plugin` is explained more detailed.

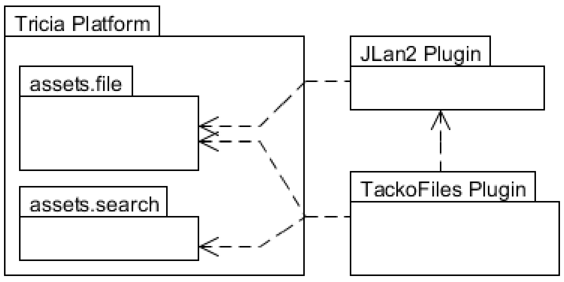


Figure 5.3: Plugin overview.

5.2.2 TackoFiles Plugin

The Tacko File plugin is the major implementation of this work. It is structured in three main components, request handling, algorithms and testing. Figure 5.4 illustrates the package diagram with all important dependencies.

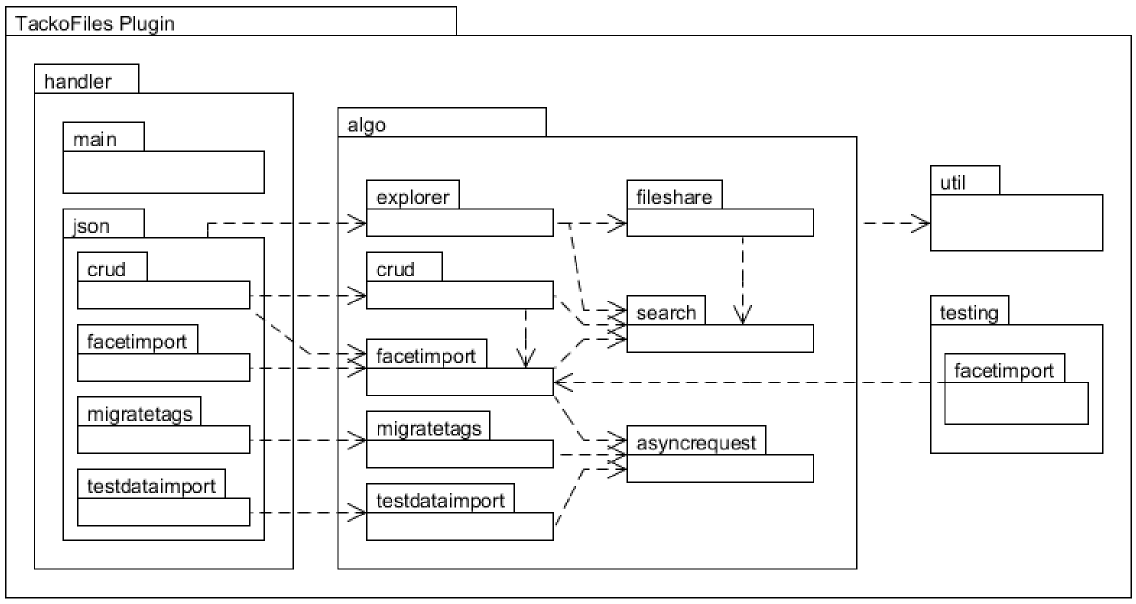


Figure 5.4: TackoFiles package diagram.

5.2.2.1 Handler Package

Each request of the tackoFiles Plugin is forwarded to this package. Directly in this package is only the `IndexHandler` located which handles the initial request. All further requests are ajax based. This package contains a `main` and a `json` subpackage. Main Handlers are requested by clicking on of the navigation menu links on the left page part. The current main element is replaced with the responsed html content. All json based requests are placed in the json subpackage. Within the `crud` package all tag-based create, rename, move and delete operations requests are handled. The Handler of the other three packages, `migratetags`, `facetimport` and `testdataimport` handles a statefull multithreaded process. During the process, the client polls the current process state with a process id and updates the client.

5.2.2.2 Algo Package

`Algo` is a shortcut for algorithm, this package contains different kinds of subpackages. The most general sub package is `search` and supports all kinds of tag-based search requests. It wraps the tricia default search package and offers a more powerful interface to access tag-based directories and files. The `fileshare` package supports the smb tag-based export. Each multithreades statefull process implements the abstract state handling from the `asyncrequest` package. This includes the `facetimport`, `migratetags` and the `testdataimport`. One of the core functionality is the `facetimport` algorithm. All related sub algorithms are also placed in this package like preparing the hierarchical file system for the import etc.. The `explorer` package provides the data for accessing each kind of hierarchical representations web-based. It includes also the tag-based navigation concept for the web-based output. Therefore this package uses some basic path manipulation methods form the `fileshare` package. Finally the `crud` package implements the tag-based create, rename, move and delete operation corresponding to the tag-based view generated form the `explorer` package.

5.2.2.3 Testing Package

The `testing` package contains a sub package `facetimport`. All jUnit testcases are placed there with the corresponding testdata initialization helpers. Testdata itself is stored as `xml` files in an folder named `Testing`. Each test casa is represented by an own file. A xml parser to read the data and initialize it is also placed there.

5.2.2.4 Util Package

All small and often used helper methods are contained in the `util` package. E.g. special often used tag-based list operations. Additionally the configuration manager is placed in this package. This configuration manager reads the `tackoFiles.properties` file and provides accessing and manipulating these properties. These property contains the current selected tag-based navigation concept.

5.3 Implementation

This chapter illustrates the main implementation concepts. Algorithms are not explicitly, explained in this chapter (see Chapter 4) they are only applied and the general context is explained. The Chapter 5.3.1 covers the different kinds of imports. An encapsulated accessing mechanism for tag-based access on hierarchical file systems is described in Chapter 5.3.2. Tag-based export is divided into different chapters, based on the samba network protocol (see Chapter 5.3.3) and web-based (see Chapter 5.3.4). Necessary tricia extensions for this implementation are explained in Chapter 5.3.5. Finally the testing with the corresponding data is illustrated in Chapter 5.3.6.

5.3.1 Import Tags

The two different kinds of tag imports are explained in this chapter, the simple import (see Chapter 5.3.1.1) and the facet import (see Chapter 5.3.1.2).

5.3.1.1 Simple tag import

In general the idea of this simple import is trivial, the directories of the path represents the tags. The simple tag import works event based. When a `path` is changed in a document or directory, the on change listener of the `Path` class is triggered. In order to maintain the consistency between tags and the path it would be enough to set the tags related to the new path. Documents can be tagged additional with more tags which are not contained in the path. Simply updating the tags related to the path would discard additionally tags. Therefore a document is searched in the cache. The tricia platform use internal caches for all kind of entities like directories and documents and provides to access these. If it is new created, no document will be found, otherwise the path related tags of the old document are calculated with the static `getPathTagNames` method. All returned tags are unassigned from the current document. In this state the document has only the additional tags, in the next step the path tags of the current document are calculated and assigned to the document. Path and tags are in a consistent state. Changing the directory tags is conceptual identical with these of changing the document tags. There is only a little difference, the last path section of a document contains the document name, in case of a directory it is the last directory name. The method `pathTagNames(...)` does not consider the last path section. The behavior is correct for all documents but not for the directories, the last directory in the path would not be tagged. Therefore an additionally slash is appended to the path which represents one more path section. The last section is not considered and the directory is tagged with all real path sections.

5.3.1.2 Facet import

Based on the existing directories facets are imported. Figure 5.5 shows a conceptual class diagram of the facet import. These classes represent only the logical part of the import. To visualize the facet import these classes are wrapped, more in Chapter 5.3.4.1. The `DirectoryAccessManager` encapsulates recursively directory accessing and provides this data in maps and sets. Accessing all directories recursively from the database is time-consuming, therefore this is done by a new thread. Related to this class, the `PathToLowerCase` and `DirectoryCycleDetector` make sure the pre-import conditions are fulfilled (see Chapter 4.1.1). All dotted lines in the diagram express a dependencies. The `Subsumption` class is the link between the different algorithms. Directly associated to this is the `SubsumptionGraph` and the `Facet MergeManager` which prepares the facet merging. A facet merge needs many comparisons until the facets are merged, this scales not linear with an increasing amount of data. Several `FacetMergeWorkers` threads merge the resulting facets simultaneously.

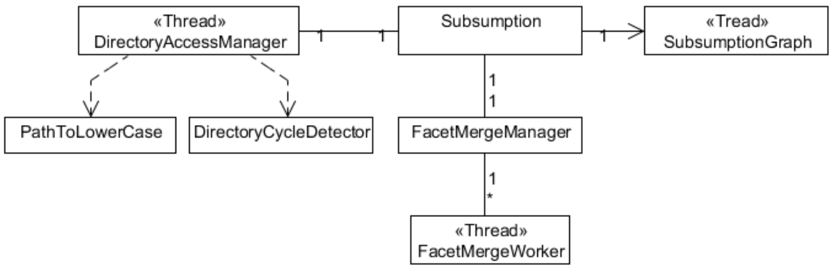


Figure 5.5: Facet import conceptual class diagram.

All theoretical algorithms for this facet import are already described in the Chapter 4.1.3. The sequence diagram in Figure 5.6 and 5.7 shows the algorithm method calls in dependence to the time and objects. Not all method calls are illustrated, this is a simplified representation, which is as simple as possible to clarify the concept. This import implementation is multithreaded, each thread is highlighted with an color. Light gray represents the algorithm's main thread and the dark gray ones are child threads.

The import starts with creating a new `Subsumption` object. The root path for the facet import is handed over, `\projects\`. Afterwards a new `DirectoryAccessManager` is instantiated with the root path as parameter. After initializing itself, a new instance of the class `PathToLowerCase` is created. Within the constructor all paths in the hierarchical source file system are renamed to lower case paths. In the next step path cycles are detected and removed. A new `DirectoryCycleDetector` is created. The detection begins with the system root and searches recursively in all subdirectories for cycles. When the directory name already exists in the path, a certain character is appended to the name. After the algorithm has terminated, the object is referenced nowhere and is collected by the garbage collector. All pre-import conditions are fulfilled and the conceptual import can start. In the first step the `DirectoryAccessManager` calculates recursively the directory name map. This recursion starts with the root parameter path `\projects\`. The resulting `directoryNameMap` is the basic data for all later calculations.The last action of the `DirectoryAccessManager` starts a new thread with the method call `start()`.The new thread, colored dark gray, starts the recursive sibling names calculation, beginning with the path `\projects\`. All siblings are stored in a set, $Set < Set < Sibling >$. Based on this nested sets all `siblingParis` are calculated and stored in a set of sibling pairs.

Meanwhile the main thread returns to the `Subsumption` object, illustrated in light gray. Now the subsumption of each tag is calculated with the method call `calculateSub-sumption`. For this calculation the `directoryNameMap` is needed as input. Therefore the method `getDirectoryNameMap` is called in the directory access manager.With this map, the subsumption of each tag is calculated. Depending on the algorithm mode the subsumption graph calculation starts. This graph is only calculated to visualize the subsumption without transitive edges. In this case a new `SubsumptionGraph` object is created to start the graph calculation. This calculation is time consuming and runs in a new thread after the graph is calculated, it is pushed to the client queue the resulting data is not needed internal.

During the subsumption graph calculation runs, the main thread calculates all potential facets based on the subsumption. The potential facets are basically the inverse of the subsumption. All potential facets and the corresponding context are stored in a map of sets named `potentialFacets`. In the next step, the instantiation of the `FacetMergeManger`, the final facet merging begins. All potential facets for one context are represented in one set, with the method `splitPotentialFacets` each potential facet tag gets its own set, `Map<Set<ContextTag>,Set<Set<FacetTag>>`.

Figure 5.7 illustrates the merging of these split facets. Several `FacetMergeWorker` threads are created. Each of these tries to get new workpackages from the `FacetMergeManager`. A workpackage contains a pair of a context and the split potential facets, `Pair<Set <ContextTag>, Set<Set<FacetTag>>>`. For the facet merging all name pairs are necessary and accessed with the method call `getNamePairs`. The method `mergeFacets WithPair` checks if the pair tags are contained in two different facets and merge these facets. This is done iterative, for all pairs.

In the next step the merged facets are ordered and persisted in a `ContextConfiguration` entity. The order priority is native facets first and than synthetic ones, the lexicographical order is applied as second priority. This order is important and influence the navigation-concept massively. The `FacetMergeWorker` gets a new workpackage if there are some facets left in the `splitFacetMap`. The main thread is waiting until all merge threads are terminated. Afterwards all facets are merged and the facet import is finished.

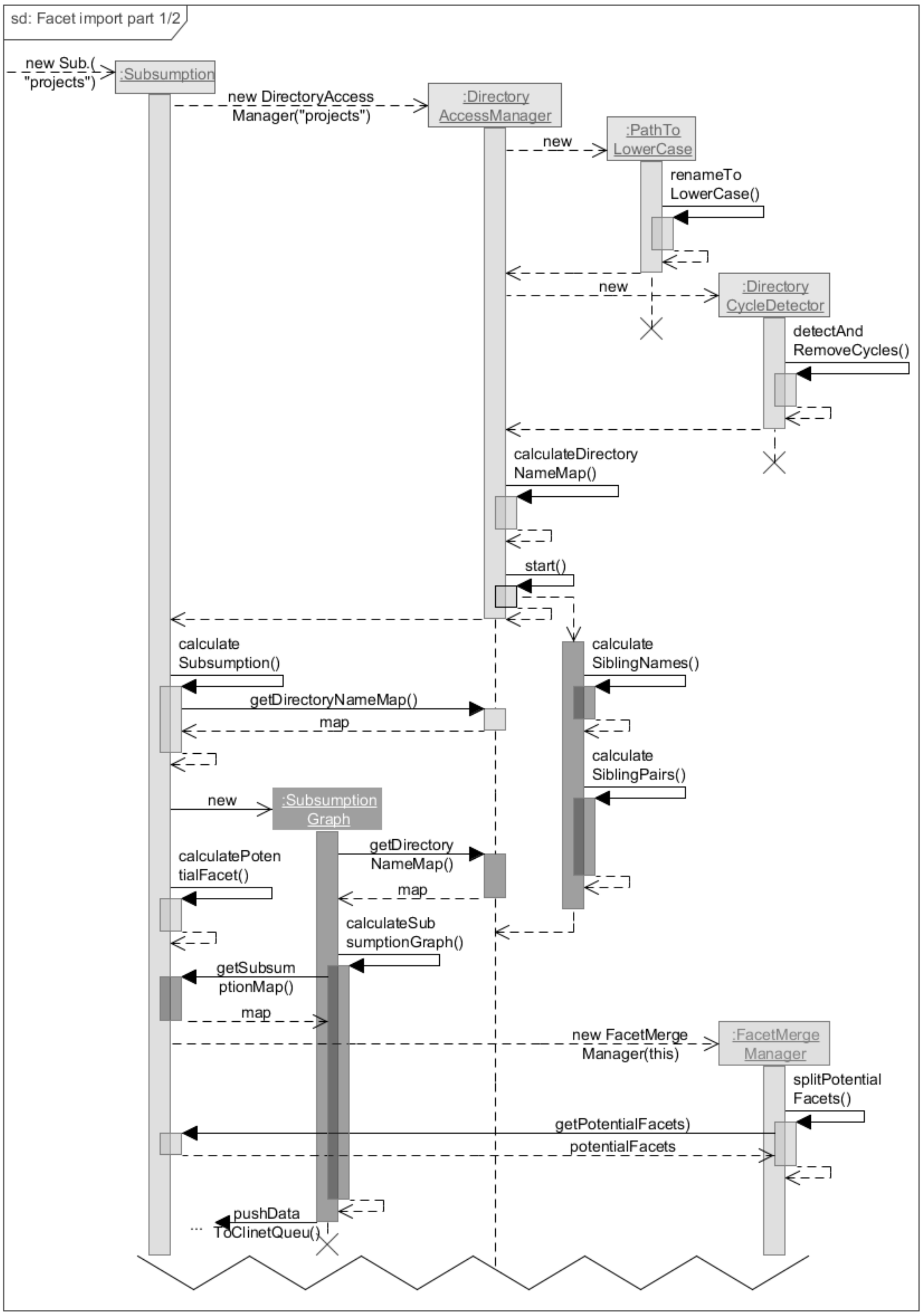


Figure 5.6: Facet import sequence diagram part 1.

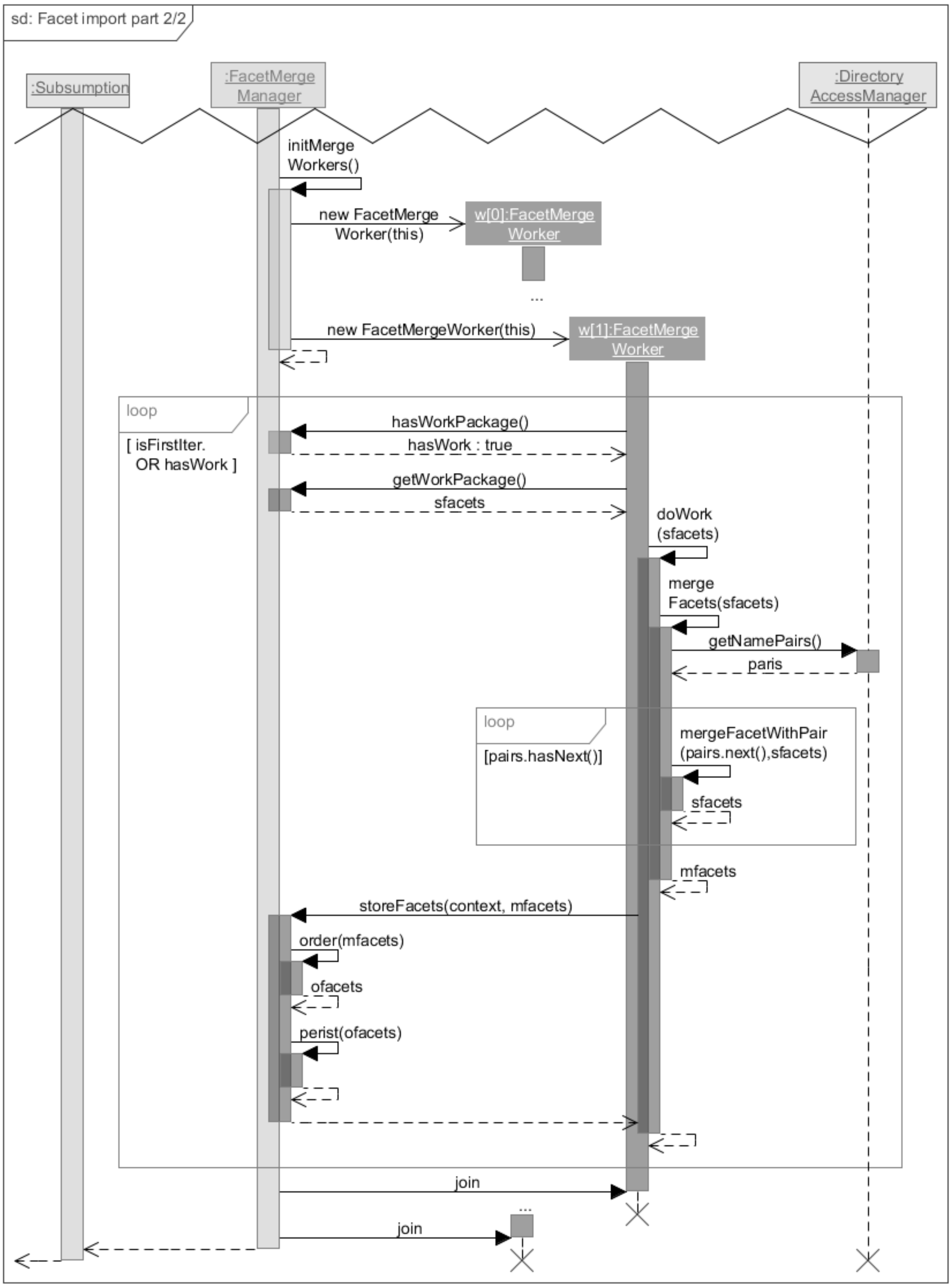


Figure 5.7: Facet import overview sequence diagram part 2.

Performance

This chapter describeS briefly the performance of the facet import algorithm. Figure 5.26 illustrates the threads within the java virtual machine during a facet import. The test environment contains approximately 2,300 directories and 4,750 documents. Documents should not affect the performance, they are not considered within the import algorithm. The colors in the figure represent the different thread states, see label on top of the figure. Each row contains one thread, beginning with the thread name. In the first row a sleeping thread named `TACKO Connection Pool Manager` is listed. This tread handles the database connections (see Chapter 5.3.5.1). The facet import main thread is listed in the third row and named `TACKO FacetImportState`. During the first time when only this thread is runnable, the pre-import conditions are established and the `directoryNameMap` is calculated. In the future, the pre-conditions check could be replaced by hierarchical source file system constraints. Then two third less time is needed before the process is parallelized. After the `subsumptionMap` is calculated by the main threadm, the `TACKO SubsumptionGraph` thread calculates all not transitive edges of the subsumption graph. This calculation is not necessary for the facet import only for the graphical visualization. In the performance import mode this graph is not calculated (see Chapter 5.3.4.1). Within the last import step all facets are merged with many comparisons, this is done simultaneously by several `TACKO FacetMergeWorker-[x]` threads. The figure illustrates only one testcase and is may not representative. In each average case the subsumption graph calculation needs the most time.

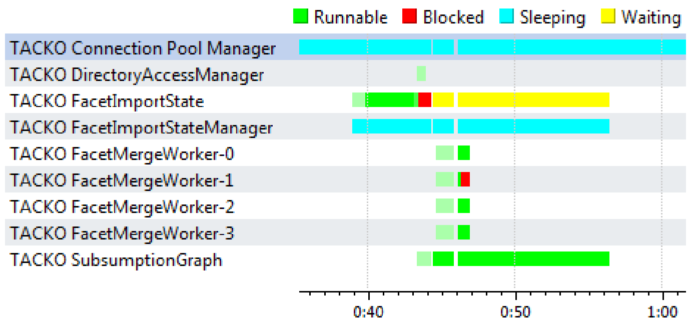


Figure 5.8: Facet import - YourKit Java Profiler screencapture.

5.3.2 Hierarchical tag-based search with the facet search wrapper

The `FacetSearchWrapper` (Figure 5.9) wraps all tag based searches and offers encapsulated some special hierarchical tag-based operations on the search results. This diagram is simplified and illustrates only the most important attributes and methods. The `FacetSearchWrapper` extends the abstract `SearchWrapper`, which provides all basic tag-based search functionality. Two different constructors are offered to create a `Facet SearchWrapper`, one with only the `mustTags` and the other one contains additionally the parameter `mustNotTags`. Following, all important method types are briefly described first the `SearchWrapper` ones and then the `FacetSearchWrapper` methods.

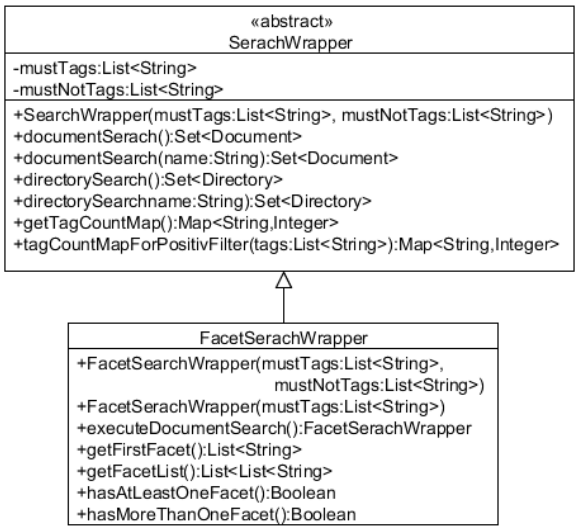

«abstract»
SerachWrapper
-mustTags:List<String>
-mustNotTags:List<String>
+SearchWrapper(mustTags:List<String>, mustNotTags:List<String>)
+documentSerach():Set<Document>
+documentSearch(name:String):Set<Document>
+directorySearch():Set<Directory>
+directorySearchname:String):Set<Directory>
+getTagCountMap():Map<String,Integer>
+tagCountMapForPositivFilter(tags:List<String>):Map<String,Integer>
FacetSerachWrapper
+FacetSearchWrapper(mustTags:List<String>, mustNotTags:List<String>)
+FacetSerachWrapper(mustTags:List<String>)
+executeDocumentSearch():FacetSerachWrapper
+getFirstFacet():List<String>
+getFacetList():List<List<String>
+hasAtLeastOneFacet():Boolean
+hasMoreThanOneFacet():Boolean

Figure 5.9: Facet search wrapper class diagram.

A trivial tag-based document search uses the method `documentSearch` without a parameter. It returns all documents which are tagged with the `mustTags` and not tagged with the `mustNotTags`. The second interface with a `documentName` parameter filters the resulting documents additionally. All documents which are not named like the `documentName` are removed from the result set. Corresponding to the document search, a `directorySearch` is provided, with and without name parameter. First one searches for all directories which are tagged with the `mustTags` and not tagged with the `mustNotTags` like the document search. The behavior of the directory search with name is like expected similar to the document search with name, all directories with the name are returned. Internal it means to manipulate each item of the search result. The conceptual problem and procedure is illustrated in Chapter 4.2.4. All directory and document searches use internal one generic method to simplify the searches.

Based on the search result, the method `getTagCountMap` returns the related tag map. The tag count map is used to represent folders when a context does not contain any facet. A tag which is already in the current path should no be placed as folder inside this path. Therfore before this map is returned, all items are removed which are contained in the mustTags. The even more specialized method tagCountMapForPositivFilter provides filtering the tag count map and returns a map which contains only the filter tags.

The FacetSearchWrapper provides all facet based methods on a search result. The method executeDocumentSearch searches for all documents which are matching with the mustTags and mustNotTags, returns the FacetSearchWrapper object itself like the builder pattern. Directly after this execution, some analyzing methods can be used. Based on the search with the method hasAtLeastOneFacet can be checked whether minimum one facet exists or not. Analogical it can be checked whether more than one facet exists or not. Further getFirstFacet and getFacetList provide accessing the search corresponding facets.

5.3.3 SMB integrated tag-based navigation

In this chapter the SMB integration and the tag-export is covered. First the integration environment is described (see Chapter 5.3.3.1). The tag-based export via Samba-protocol is described in Chapter 5.3.3.2. The export for both navigation-concepts, native tag-based and multifaceted tag-based are illustrated.

5.3.3.1 JLan2 Plugin integration

The de.infoasset.platform.assets.file package implements the basic hierarchical filesystem classes. Figure 5.10 illustrates this on the left part of the class diagram. The implementation of these classes represent a composite pattern. Aim is to access the hierarchical file system classes with the JLan2 plugin. Therefore the facade classes are implemented as interface for the file system classes, illustrated on the right side of the figure. They are placed in the de.infoasset.platform.assets.file.facade package. [Wis11, p. 37]

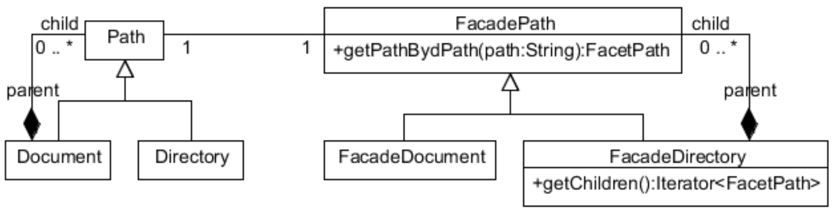


Figure 5.10: Hierarchical filesystem classes with the corresponding facades.

The tacko files plugin provides a new static folder tackoFiles on the root level of the virtual smb device. Therefore the getChildren method in the FacadeDirectory is modified. It is checked if the current directory is the root, in this case the tackoFiles folder is appended. Within this directory the tag-based navigation concept is provided. All actions containing these directories must be forwarded to the tacko files plugin. The modified getPathByPath method in the FacadePath forwards all these actions within the tag-base navigation folder to the tacko files plugin.

The JLan Plugin implements a virtual samba network device based on the Alfresco JLan Server Framework. All methods of the `DiskInterface` from the framework must be implemented to provide access on the virtual device. This interface is implemented by the `TriciaDiskDriverProxy`. Figure 5.11 illustrates a representative samba request for the path `\tackoFiles\projects\`. The request executes the method `startSearch` in the `TriciaDiskDriverProxy`, which forwards the search to the `TriciaDiskDriver`. A new `AssetSearchContext` is instantiated and calls the static method `getFacadePath ByPath` in the class `FacadePath`. When the path begins with `\tackoFiles\` and the `tackoFiles Plugin` is executed, method `getTagDirectory` is called. This method is implemented in the `TackoFilesPlugin`. Depending on the current `TackoFiles Configuration` a `NativeTagDirectory` or a `GroupByTagDirectory` is created. The distinction with or without tag-count option is done within each `TagDirectory`. Finally the `getChildren` method is called in the new created object. This method returns all children on this directory, subdirectories and documents. The children are appended to the hierarchical file structure on the virtual samba device.

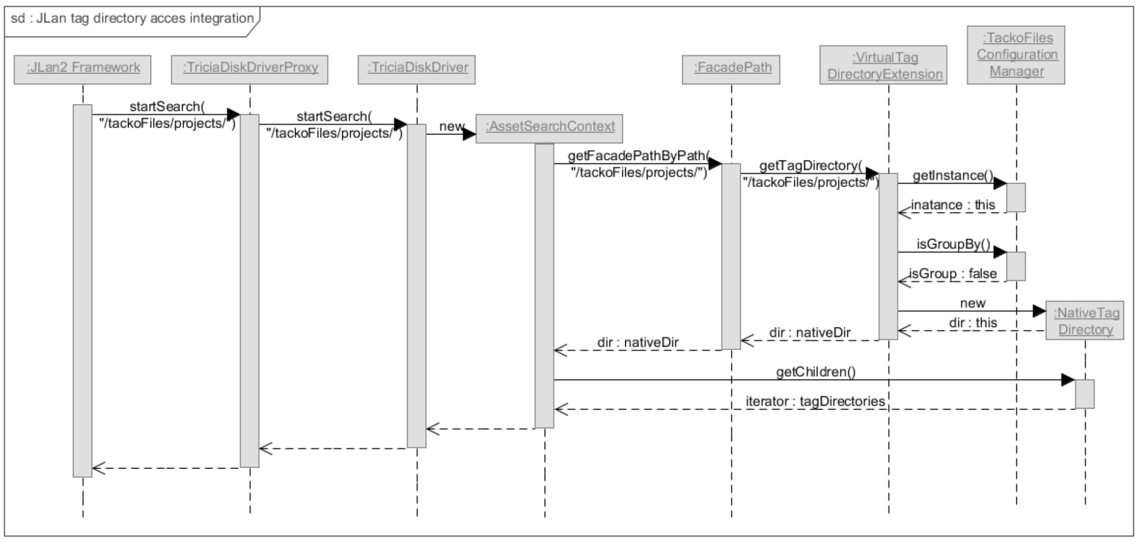

sd : JLan tag directory acces integration
:JLan2 Framework
:TriciaDiskDriverProxy
:TriciaDiskDriver
:AssetSearchContext
:FacadePath
:VirtualTag DirectoryExtension
:TackoFiles Configuration Manager
:NativeTag Directory
startSearch("/tackoFiles/projects/")
startSearch("/tackoFiles/projects/")
new
getFacadePathByPath("/tackoFiles/projects/")
getTagDirectory("/tackoFiles/projects/")
getInstance()
inatance : this
isGroupBy()
isGroup : false
new
dir : this
dir : nativeDir
dir : nativeDir
getChildren()
iterator : tagDirectories

Figure 5.11: JLan Plugin integration.

5.3.3.2 SMB-based tag export

Representing tags as a hierarchical file structure is a similar expression for export tags. The export provides the tag-based navigation concept integrated on a virtual samba network device (see Chapter 5.3.3.1). Figure 5.12 illustrates all important export classes and the dependencies. Already existing classes are marked gray. Gray classes with a `(+)` are modified for this export. In order to represent tags as directories, it is necessary to represent physically not existing directories. All physical path related methods are overridden in the abstract `VirtualDirectory` class which extends the `FacadeDirectory`. The `RootTagDirectory` extends the `VirtualDirectory` and represents the `tackoFiles` folder on the root level. Within this directory all subdirectories are from the type `Tag Directory`. There are existing two specific `TagDirectory`s, one represents the native tag-based navigation concept and the other one the multifaceted tag-based navigation concept. Creating the tag-based navigation directories depends on the current settings in the `TackoFilesConfiguration`. Navigation options like native or multifaceted tag-based navigation are supported. Additionally, it can be defined if each folder with tags contains a count postfix.

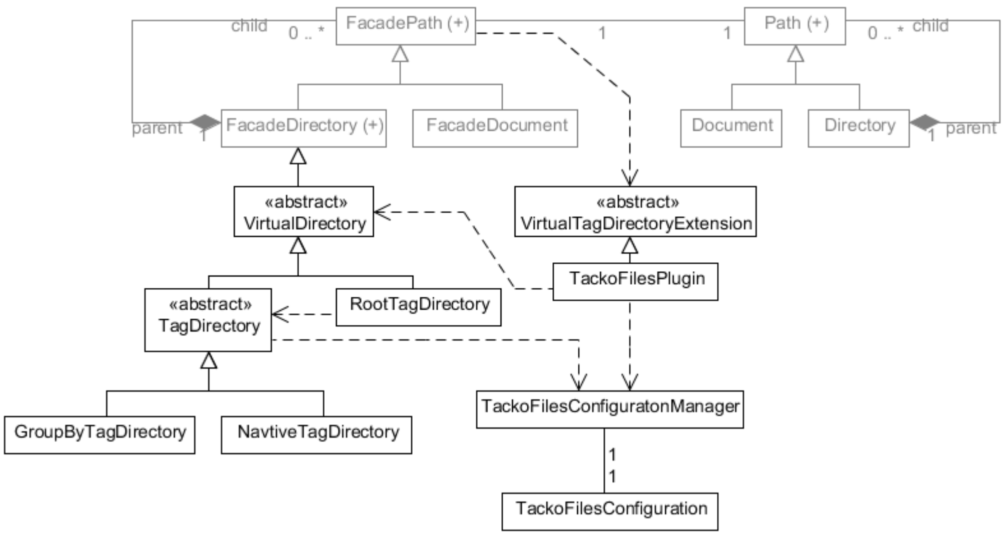


Figure 5.12: SMB-based tag export conceptual class diagram.

5.3.3.2.1 SMB-based native tag export

Based on the sequence diagram in Figure 5.11, this chapter illustrates more detailed the generation of a `NativeTagDirectory` (Figure 5.13). A new `NativeTagDirectory` is created with the parameter `\tackoFiles\projects\`. The input path is split into the path sections and the static root name `tackoFiles` is removed. This results in only one tag named `projects`. A `FacetSearchWrapper` is created to execute a documentSearch for this tag. If more than one facet is existing, the first one is added as directories to the hierarchical representation. In this case more than one facet exists and the facet is appended to the `FacadePath` with the method `addFacets`. Within this method the first facet is queried by the `FacetSearchWrapper`, the facet {`bayern, niedersachsen`} is returned. Both facet tags are added as `TagDirectory`. In the next step all documents are added. Therefore a new instance of the `FacetSearchWrapper` is created. Constructor parameters are the `pathTags` and `notPathTags`. In the sample the current path is represented with the `pathTag projects` and the tags of the facet are the `notPathTags`. The `documentSearch` returns one document named `guidelines.pdf`. This document is also appended as `FacadeDocument`. Finally a `FacadePath` iterator which contains all searched tags and documents is returned.

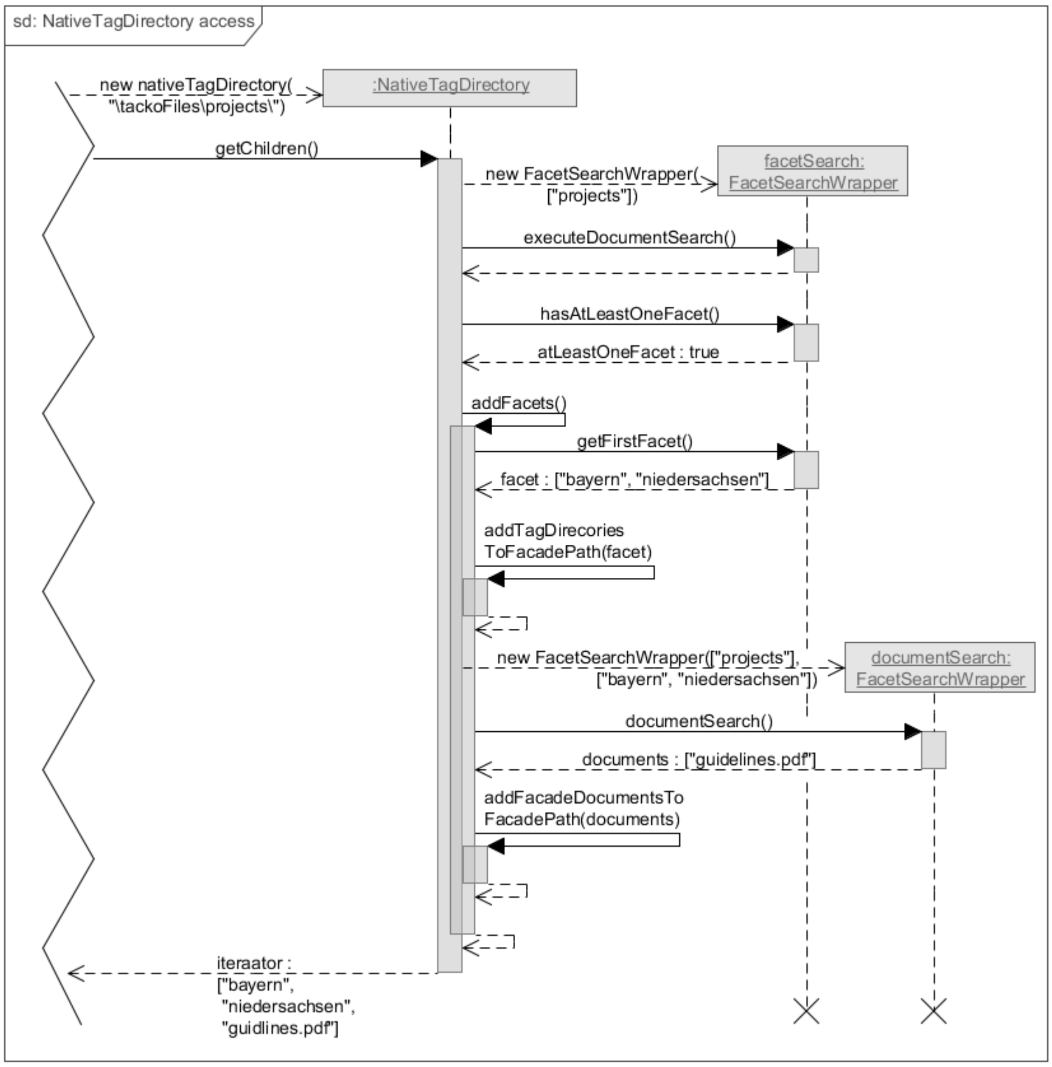

sd: NativeTagDirectory access
new nativeTagDirectory("\tackoFiles\projects\")
:NativeTagDirectory
getChildren()
new FacetSearchWrapper(["projects"])
facetSearch: FacetSearchWrapper
executeDocumentSearch()
hasAtLeastOneFacet()
atLeastOneFacet : true
addFacets()
getFirstFacet()
facet : ["bayern", "niedersachsen"]
addTagDirecories ToFacadePath(facet)
new FacetSearchWrapper(["projects"], ["bayern", "niedersachsen"])
documentSearch: FacetSearchWrapper
documentSearch()
documents : ["guidelines.pdf"]
addFacadeDocumentsTo FacadePath(documents)
iteraator : ["bayern", "niedersachsen", "guidlines.pdf"]

Figure 5.13: SMB-based native tag export sequence diagram.

5.3.3.2.2 SMB-based multifaceted tag export

This chapter explains the multifaceted tag-based export based on a sample. There are mainly three different cases. The last section of the current path contains a `group by` folder, then the names of the group by folder are added as directories to the current path. If there exists more than one facet all facets are added as group by folder, otherwise the native tag export is used. Adding facets as group by folder is represented in Figure 5.14 and used as sample in this chapter.

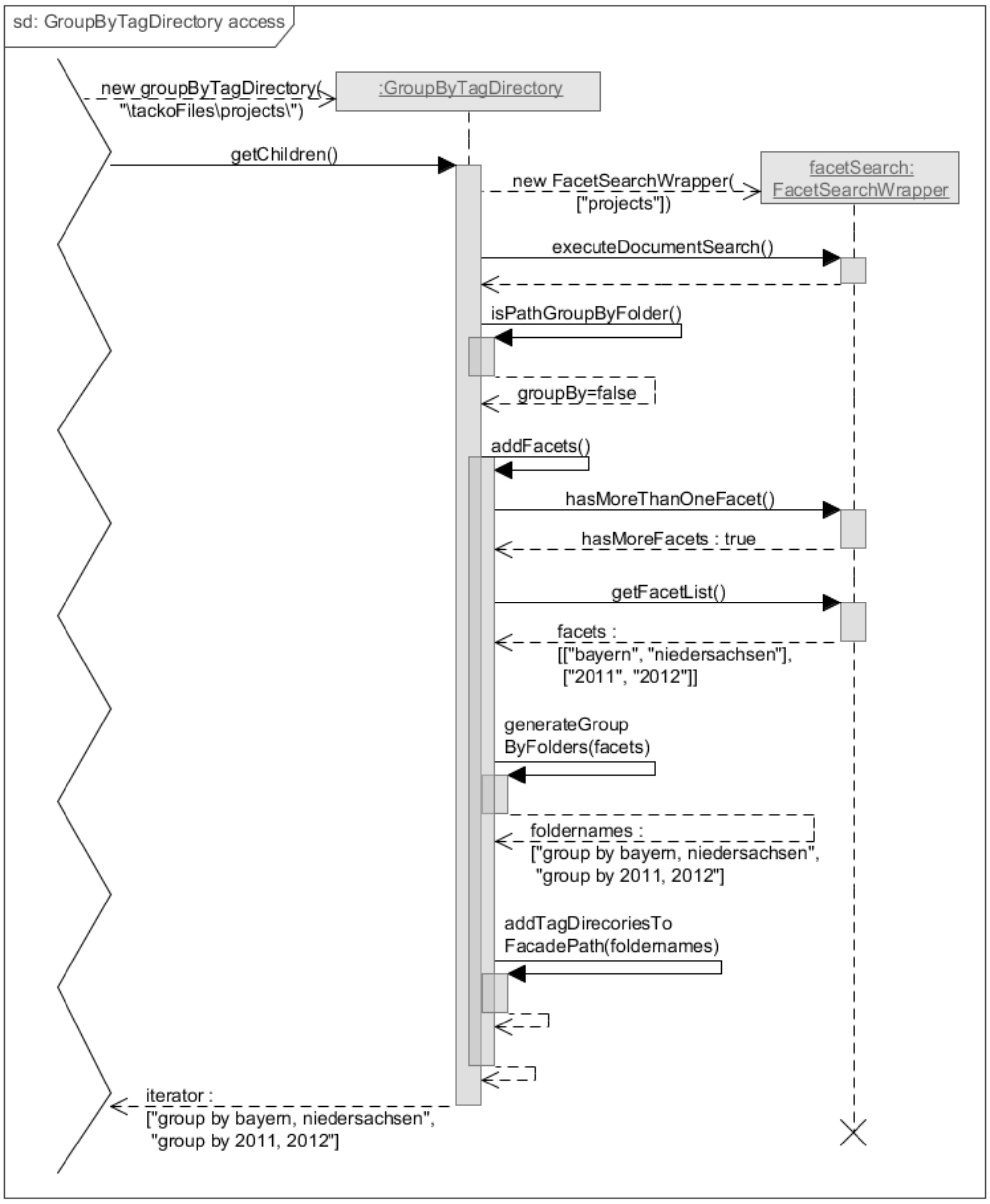

sd: GroupByTagDirectory access
new groupByTagDirectory("\tackoFiles\projects\")
:GroupByTagDirectory
getChildren()
new FacetSearchWrapper(["projects"])
facetSearch: FacetSearchWrapper
executeDocumentSearch()
isPathGroupByFolder()
groupBy=false
addFacets()
hasMoreThanOneFacet()
hasMoreFacets : true
getFacetList()
facets : [["bayern", "niedersachsen"], ["2011", "2012"]]
generateGroup ByFolders(facets)
foldernames : ["group by bayern, niedersachsen", "group by 2011, 2012"]
addTagDirecoriesTo FacadePath(foldernames)
iterator : ["group by bayern, niedersachsen", "group by 2011, 2012"]

Figure 5.14: SMB-based multifaceted tag export sequence diagram.

A new `GroupByTagDirectory` is created with the parameter `\tackoFiles\projects\`. Similar to the native export, the path is first converted into `pathTags`. With the resulting `pathTags` a new `FacetSearchWrapper` is initialized and the document search is executed. Group by folders are not added directly as subfolders into group by folders. This would result in an endless loop. Therefore the method `isPathGroupByFolder` checks if the current folder already represents a group by folder. Since this is not the case the facets are added with the method `addFacets`. In this search result more than one facet exists. The list of facets is accessed and for each facet a group by folder name is generated with the method `generateGroupByFolders`. In the last step all group by directories are added. Finally a iterator over all group by created directories is returned.

5.3.3.3 Problems

5.3.3.3.1 Limited Path Length

Especially the group by foldernames length increase normally fast[8] with an increasing amount of files and folders. The worst case scenario contains a lot of subfolders in each folder. All folders on the same level are potential candidates for one facet. One facet is represented by one group by folder, that means each facet tag is added to the group by name. More detailed algorithm information see Chapter 5.3.3.2.2.

The SMB protocol definition allows a maximum path length with around 256 characters, depending on the operation system and protocol version. Further the implementation in Tricia is more restrictive and limits the maximum path length to 210 characters. This is defined in the `Path` class with the attribute `MAXIMAL_FULL_PATH_LENGTH`. One approach to solve this problem is to reduce the path length. Shortcuts allow to compress a path by removing all unnecessary group by foldernames. The example in Figure 5.15, shows how effective a path compression with shortcuts is. All blue parts are selected paths and really necessary for navigating all other parts no more relevant.

uncompressed path:	`/group by daimler, bmw, audi/`**`bmw/group by 2009, 2010, 2011`**`/`
compressed path:	**`/bmw/group by 2009, 2010, 2011/`**
uncompressed path:	`/group by daimler, bmw, audi/`**`bmw`**`/group by 2010, 2011/`**`2011`**`/`
compressed path:	**`/bmw/2011/`**

Figure 5.15: Group by path compression sample.

The implementation of shortcuts is depending on the operating system. In windows shortcuts are simple files with the type `.lnk`, which are interpreted as links. Linux works with real links as shortcut representation. Manual creating shortcuts with a right click via context menu works. In conclusion it is theoretical possible to implement shortcuts but it is practically adjunct with a too high time and effort. More windows shortcut specific file formate information can be found in different specifications[9].

[8]path length increases > amount of files and folders

[9]http://ithreats.files.wordpress.com/2009/05/lnk_the_windows_shortcut_file_format.pdf and http://ithreats.files.wordpress.com/2009/05/ms-shllink1.pdf, (accessed 5th of June 2012)

5.3.3.3.2 Performance

After mounting the SMB network device on windows, all files and folders are recursively requested. This in combination with multiple search requests for generating one virtual folder causes the problem. With more files it causes a latency time which is not acceptable. When all files are cached it works well also after a remount.

5.3.4 Web-Interface

Several interactions are provided on the web-interface, the implementation of these are described in this chapter. In the first Subchapter 5.3.4.1 loading mechanisms are described. Web-based export tags are illustrated in Chapter 5.3.4.2. Furthermore there are several other web-based accessing possibilities explained. Finally the tag-based crud operations are illustrated in Chapter 5.3.4.3.

5.3.4.1 Asynchron load concept

This prototypical implementation contains several algorithm visualisations. These algorithms are computationally intensive and need some time until they terminate. If a normal request triggers one of these algorithms, the client would be blocked for a certain time. In case of usability it is a really bad behavior to block the userinterface for more than one second. Another problem is the browser time out. With a huge amount of data the response would take so long that the browser already throws a time out. Therefore an asynchron process handling is introduced, the concept is illustrated in Figure 5.16. The clients are placed on the left side and sent requests to the server on the right side. Within the server the different import processes are running and mapped with the manager to a client. In the first step a client sends an ajax request to the server which starts a new import. A new import state thread is created and started. Within the import manager a process identifier named pId is generated. The import state manager contains a map which links the pId to the corresponding import state thread. After initializing, the pId is returned to the client. The import process with its sub-process are still running on the sever and writing resulting data in a process related queue. While the calculation is not finished the client polls with the pId in certain intervals the import manager. The import manager returns all currently contained data in the queue to the poll request. This is done until the process is finished or some errors occur. Finished processes which are no more polled for a certain time expire and they are removed by the import state manager thread. This concept represents some kind of a broker pattern but clients are not registered. Websockets would be probably another solution for this concept.

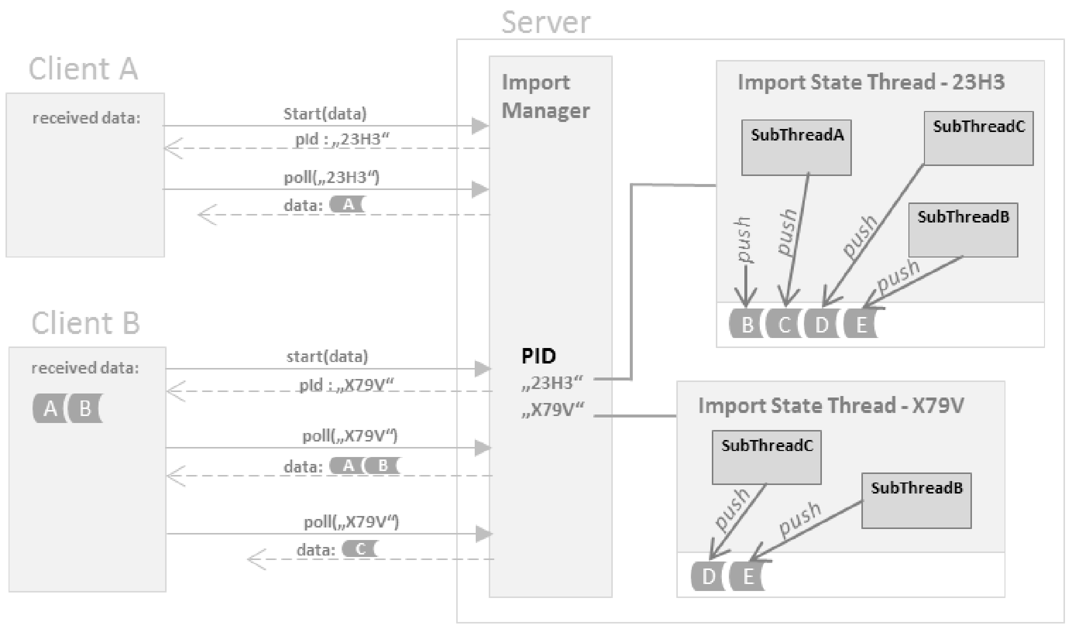


Figure 5.16: Asynchron multithreaded process handling concept.

The the generic implementation of this asynchron load concept is illustrated in Figure 5.17. Each `ImportState` runs in a new thread. With the static attribute `TIMEOUT_IN_SEC` the time until a time out occurs is defined in seconds. With the method `isExpired`, it can be checked if the import state is already expired. The `clientQueue` contains all data from the import process which is destined for the client. Another attribute represents the current state of the algorithm. The `serverState` is a generic attribute which depends on the actual implementation. This is used to visualize the state on the client side in the process bar. In the diagram there are only the relevant methods and attributes visible, some more are existing.

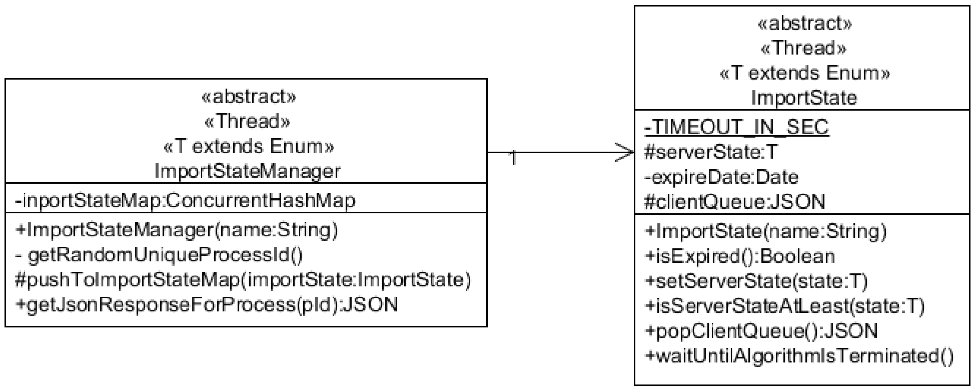

«abstract»
«Thread»
«T extends Enum»
ImportStateManager
-inportStateMap:ConcurrentHashMap
+ImportStateManager(name:String)
- getRandomUniqueProcessId()
#pushToImportStateMap(importState:ImportState)
+getJsonResponseForProcess(pId):JSON
1
«abstract»
«Thread»
«T extends Enum»
ImportState
-TIMEOUT_IN_SEC
#serverState:T
-expireDate:Date
#clientQueue:JSON
+ImportState(name:String)
+isExpired():Boolean
+setServerState(state:T)
+isServerStateAtLeast(state:T)
+popClientQueue():JSON
+waitUntilAlgorithmIsTerminated()

Figure 5.17: Generic asynchron import implementation.

Each `ImportantState` is linked to a `pId` within the `ImportStateManager`. The method `pushToImportStateMap` links a new instantiated `ImportState` and the inverse method `getJsonResponseForProcess` returns the queue data for a certain `pId` as json. Both abstract classes containing a name parameter in the constructor to define the thread name.

Different views implement the generic implantation like the testdata import, migrate tags and the facet import. The facet import is the most important one and most also complex one. A conceptual class diagram illustrates the dependencies (Figure 5.18). All classes are ordered corresponding to the call hierarchy. On the bottom the `Subsumption` class is placed which represents an interface for the facet import algorithm. All resulting data from the facet import is pushed to the `FacetImportState`. This class extends the generic `ImportState` class. The blue dashed boxes represent the synchron and the asynchron variant of the facet import. The synchron `PerformantFacetImport` is used to recalculate the facets after each crud operation (see Chapter 5.3.4.3). Therefore it is necessary to execute the import and wait until the algorithm is terminated. Furthermore the subsumption graph is not calculated in this mode to reduce the computing time. The import process itself is always started in a new thread. Therefore the method `waitUntilAlgorithmIsTerminated` is called and the `PerformanceImportThread` is set to the state `waiting` and is notified when the algorithm terminates. This sample is focused more on the asynchron import. All facet imports are mapped in the `FacetImport StateManager` which extends the generic `ImportStateManager` class. On top there is a start handler and a reload handler. The reload handler is also generalized.

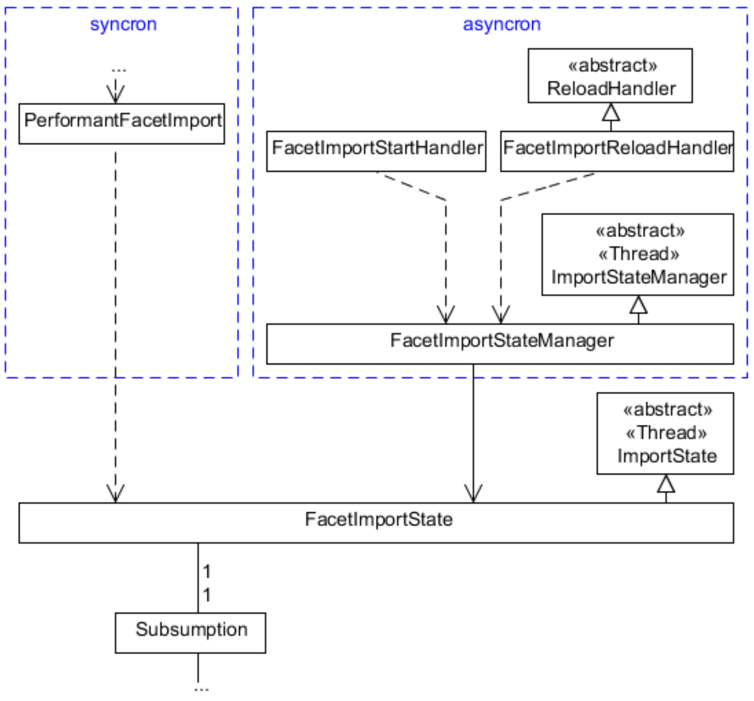


Figure 5.18: Conceptual facet import class diagram.

Figure 5.19 illustrates a visualized facet import. In the first step an ajax request with a parameter `directoryId` is sent to the relative url `\json\FacetImportStart\`. The related `FacetImportStartHandler` calls the `startImport` method with the request parameter in the `FacetImportStateManager`. Within this method a new `FacetImport State` object is created which contains the client queue and several other import related state information. This new object is linked to a generated unique process id and stored in a map. The unique process id is returned to the client. Direct with the creation of `FacetImportState` object a new thread is started and the facet import begins, colored in light gray. A new Subsumption instance and all other relevant objects are created (see Chapter 5.3.1.2). During the process all visualisation data is pushed to the client queue in the `FacetImportState` instance. The client sends poll requests in certain time intervals to the relative url `\json\FacetImportReload\`. As parameter the `pId` is contained in each poll request. The requested reload handler calls the `ImportStateManager` to get the queue data for the process with the `pId`. The expire date is increased and all data contained in the queue is removed and returned to the client. The client polls until the algorithm has terminated. Each `FacetImportState` expires after a certain time without poll. All expired imports are removed in defined time intervals from the `ImportStateManager` thread, colored in dark gray.

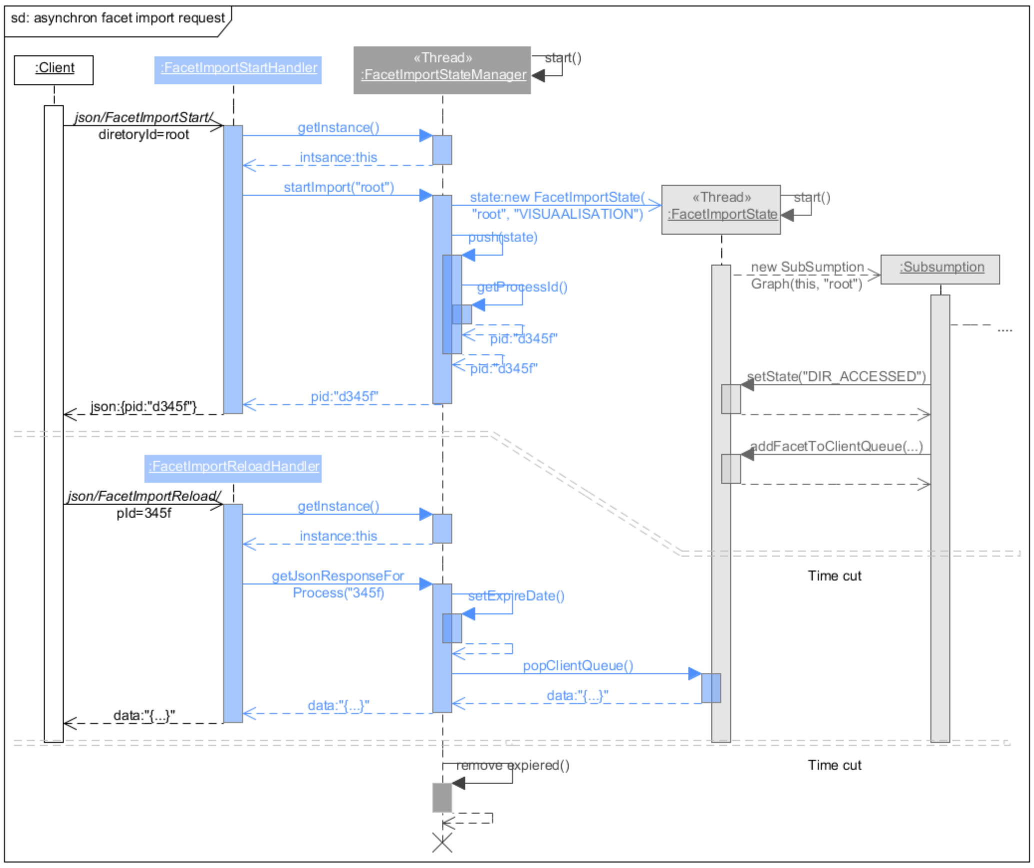


Figure 5.19: Asynchron facet import sequence diagram.

5.3.4.2 Accessing hierarchical structures web-based

All data for a web-based hierarchical folder representation is provided by the `Explorer` class. Three mainly different hierarchical structures are offered with inherited classes. First of all, data from the local file system is provided by the `LocalExplorer`. Usage of this data is only senseful with a locally installed web-server otherwise users could access the files of the server. With the `FilesShareExplorer` all tricia based directories and files are provided. For this work the most important tag-based data access is provided by the `TagExplorer`. The `NativeTagExplorer` and `GroupByTagExplorer` are inherited from the `TagExplorer` and contain the concrete implementation. Figure 5.20 illustrate the class inheritance and the associations.

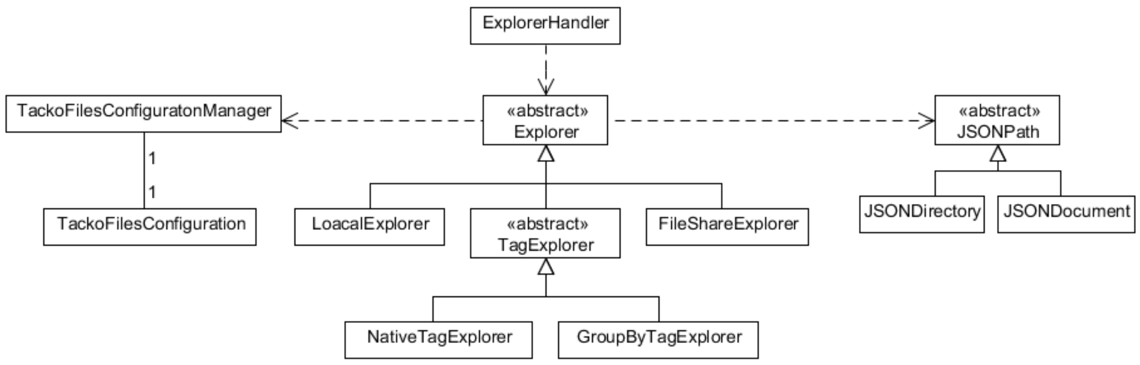
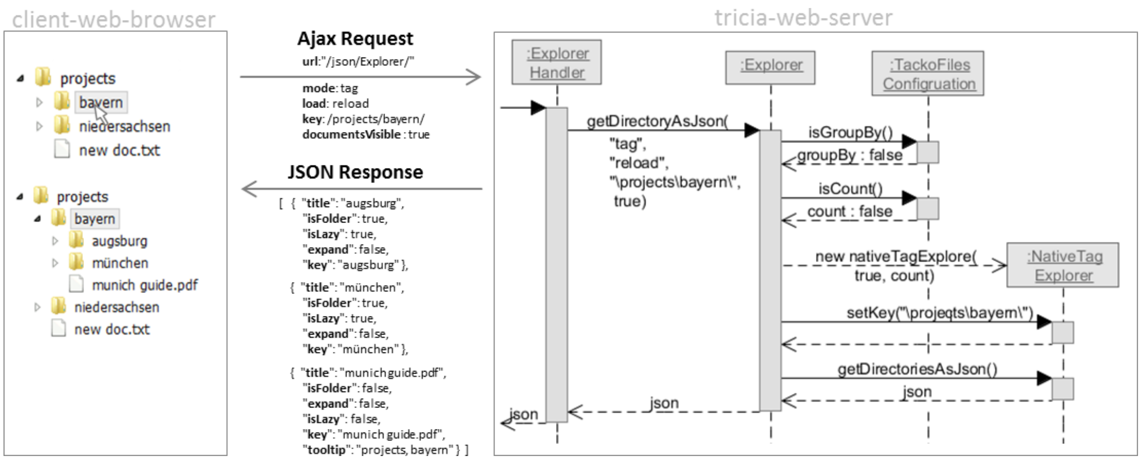


Figure 5.20: explorer overview class diagram.

The web-based tag export sample in Figure 5.21 illustrates the interaction between client and server. With a click on a lazy not expanded folder, this folder is requested with an ajax call. All requests to access hierarchical represented data structures are sent to url `\json\Explorer\`. Parameters allow to differ between the different kind of requests. The parameter `mode` selects the kind of data structure, tag-based, local or the tricia directories and documents. For each of these there are two different `load` types possible, an `initial` load or a `reload`. The `key` parameter describes the source. In the tag-based and in the local mode a path is set as key. For accessing the tricia directories and files the key represent the unique identifier. Within the `initial mode` no key is set, the source is set by default to the system root. The last parameter `documentsVisible` describes if documents are appended to the representation or not. On the server side the `ExplorerHandler` is executed and calls the static method `getDirectoriesAsJson` in the abstract `Explorer` class. Depending on the parameter `mode` this static method instantiate a specialized explorer object. In this case the parameter has the value `tag`. The `TagExplorer` can represent a native tag-based navigation or a multifaceted tag-based navigation, with or without count option. It depends on the current settings in the `TackoFilesConfiguration`. The sequence diagram illustrates the request for a native tag-based navigation without count option. A new instance of the `NativeTagExplorer` is created and the key is set to `\projects\bayern\`. The internal algorithm concept works similar to tag export via SMB (see Chapter 5.3.3.2). After the internal algorithm has calculated the tag-based directories and files they are returned as json. The classes `JSONDocument` and `JSONDirectory` (Figure 5.20) encapsulate the transforming into the target json format. Each represented item in a hierarchical structure has several attributes. The `title` describes the visible name of the item, the attribute `isFolder` defines an item as directory or a document. Loading mechanisms can be influenced with the attribute `isLazy` by, default it is `true` for each directory in this implementation, if some content within the directory exists. The attribute `expand` allows to append already opened directories to the hierarchical structure. Finally with the attribute `tooltip` some additional information can be added. The tag-based implementation shows for each document the corresponding tags. These described steps are repeated for each click on a lazy directory.



```
url:"/json/Explorer/"

mode: tag
load: reload
key: /projects/bayern/
documentsVisible : true
```

```
[ { "title": "augsburg",
    "isFolder": true,
    "isLazy": true,
    "expand": false,
    "key": "augsburg" },
  { "title": "münchen",
    "isFolder": true,
    "isLazy": true,
    "expand": false,
    "key": "münchen" },
  { "title": "munichguide.pdf",
    "isFolder": false,
    "expand": false,
    "isLazy": false,
    "key": "munich guide.pdf",
    "tooltip": "projects, bayern" } ]
```

Figure 5.21: Web-based tag export sample.

5.3.4.3 CRUD Operations

The implementation concepts of the create, rename, move and delete operations are explained in this chapter. The conceptual class diagram in Figure 5.22 illustrates the dependencies of each operation in the general context. The classes on the diagram can be separated in three different groups, `Handler`, `Operation` and general algorithms like the `TagBasedPathFinder` and the `FacetSearchWrapper`. Each `Operation` has a corresponding `OperationHandler` to execute this operation. The `OperationManager` stores initialized operations until they are executed or the operation expires after a certain time and is removed. Each operation has a dependency to the `JSONResponse` class which encapsulate the json response generation with defined string keys. The operations are placed depending on which algorithm they use. The create and move operations are both using the `TagBasedPathFinder` (see Chapter 5.3.4.3.2) to find a physical path in the hierarchical source file system. This path search returns a `TransientDirectory` which represents a path which is maybe not physically existent. All operations with exception of the create operation use the `FacetSearchWrapper`(see Chapter 5.3.2) to search for directories or documents. The abstract `Operation` defines an abstract `initialize()` method which is implemented in each specialized operation class. Depending on the logical algorithms the method is implemented in the directly abstract inherited operation class or in the lowest inheritance level. E.g. the `initialize` method could be implemented in the abstract `MoveOperation` class or in the `MoveDocument` and `MoveDirectory` class. There is a second abstract method in the `Operation` class defined named `execute( JSONObject json)`. The abstract `OperationHandler` applied the `PerformantFacet Import` after each execute operation.

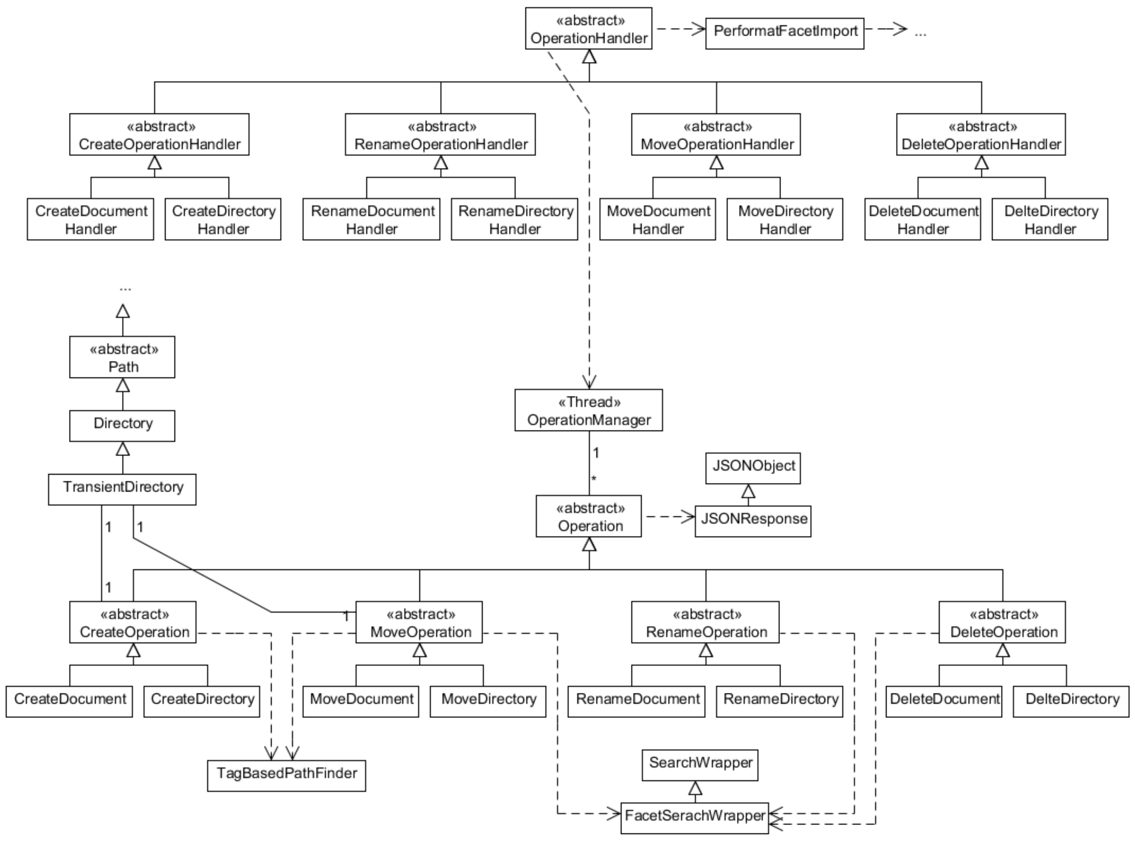


Figure 5.22: crud operation overview class diagram.

All algorithms are already illustrated in Chapter 3.3, the sample in Figure 5.23 explains the general operation architecture. The left side on the figure describes the client state and the right side show the conceptual process on the server. In between the requests and responses are illustrated. Based on the multifaceted tag-based navigation concept a new tag is created in the tag path `\projects\group by 2012, 2011\2011\`. With the click on the context menu the path from the selected folder is recursively calculated with a parent function. An ajax request is sent to the web server with the relative url `\json\CreateDirectory\`. The request contains two parameters, `mode` and `pathTags`. In this case the mode is set to the value `initial` to initialize this operation. The second parameter describes the current path of the operation. On the server side the `Create DirectoryHandler` is mapped to this requested url. Depending on the mode a new operation is created or an already initialized one is executed. In this case the parameter pathTags is read and transformed into the corresponding tags, projects and 2011. A new CreateDirectory operation with the tags as parameter is instantiated. In the next step the method of initializeAndStoreOperation, implemented in the abstract Operation class, is called with the new created CreateDirectory operation as parameter. The CreateDirectory operation is stored in a map with an operation identifier as key. Afterwards the initialize method is called on the operation object and returns a JSONResponse object. There is data for the dialog contained. Due to all existingNames in this physical path are contained in the json object to validate the input name on the client. Finally to the JSONResponse object the operation id is added. The initial response is sent to the client. On the web-based client a dialog pops up and shows all necessary data. If the name is valid, the create button is enabled and the tag can be created. With the on click event of the create button a second, ajax request is sent to execute the initialized operation. This request contains the operation id and a parameter executeData which is additionally necessary to execute the operation. In this case a parameter name with the value hessen is contained. The server reads the parameter, the mode is now execute. In the first step the stored operation is searched with the method getOperation, as parameter the request operation id is handed over. The initialized CreateOperation is returned. With the executeData from the request the method execute is called on the CreateOperation object. It returns a JSONResponse object which contains execution information. Modifications on the hierarchical source filesystem influence the facet building. Therefore after each execute operation a PerformantFacetImport instance is created, which re-imports all facets. The response is sent to the client and a java script method triggers a reload of the hierarchical representation.

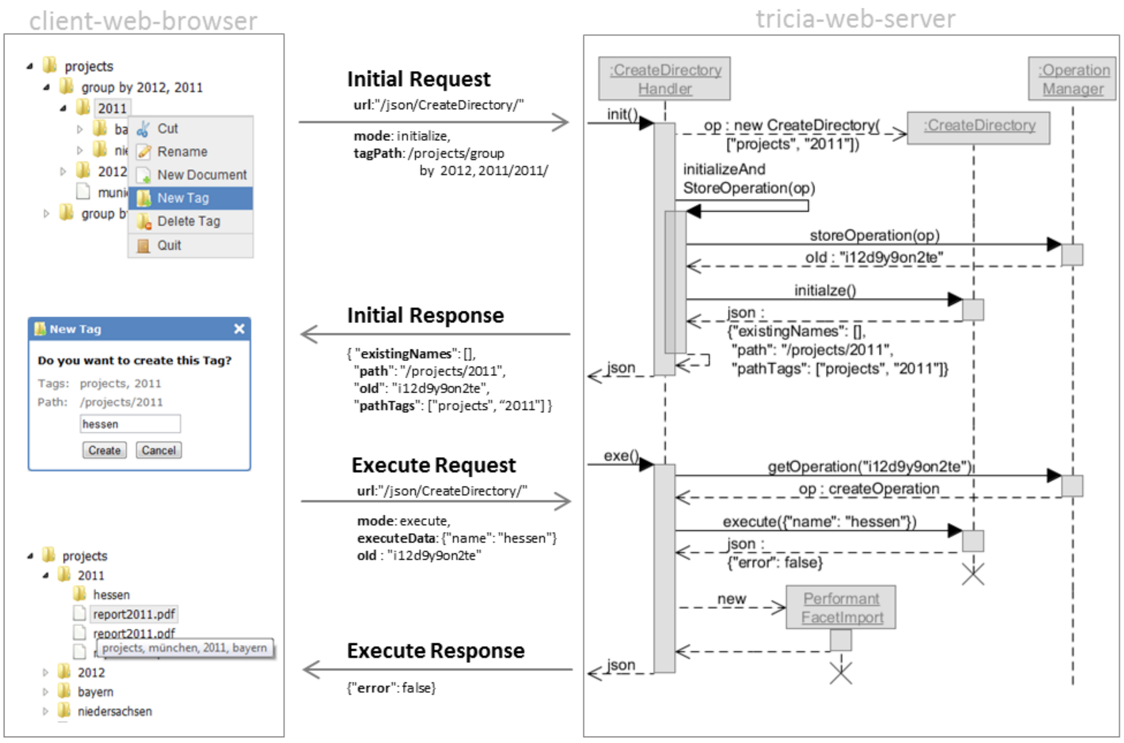


Figure 5.23: Create new tag operation on the multifaceted view.

5.3.4.3.1 Transient directory

The `TransientDirectory` extends the `Directory` and provides the additional functionality to create a transient directory and use the basic class methods without persisting this directory. Conceptual aim is to encapsulate the transient state and provide a behavior like a persisted directory. A `TransientDirectory` contains a `persistedDirectory` which represents the last already existing path of this directory. Further the attribute `transientDirectory` contains all children which are not persisted as list of strings. These two attribes must be set as parameter in the constructor. The method `isTransient` returns the state, true if it is not persisted and false if it is already persisted. Several methods are overridden to achieve the desired behavior, `getSubdirectories` and `get Documents` are returning an empty result during the transient state in the other case the calls are forwarded to the real directory. The `getFullPath` method uses the `getFull Path` method of the persisted directory and appends all transient directory paths. The create directory and document method persist the transient directory and forwards the create call to the persisted directory or document. This class is used to initialize some crud operations, uses some methods and wait until the user confirms the create or move operation.

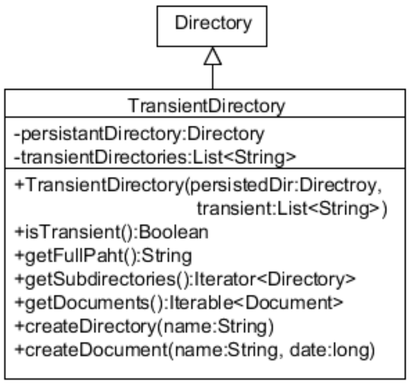

Directory
TransientDirectory
-persistantDirectory:Directory
-transientDirectories:List<String>
+TransientDirectory(persistedDir:Directroy, transient:List<String>)
+isTransient():Boolean
+getFullPaht():String
+getSubdirectories():Iterator<Directory>
+getDocuments():Iterable<Document>
+createDirectory(name:String)
+createDocument(name:String, date:long)

Figure 5.24: Transient directory class.

5.3.4.3.2 Tag-based path finder

The tag-based path finder searches for the best matching path the hierarchical file system. One public static method named `findOrCreateTransientDirectory` provides the search. Each tag-based create and move operation uses this method. Figure 5.25 illustrates the search for the path tags `["projects", "2011", "bayern"]`. The order of the path tag represents the navigation path. In this context the tag `2011` is a synthetic tag. All recursion calls are explicit drawn to simplify it and highlighted with blue lines which represent the scope. The theoretical algorithm of the `TagBasedPathFinder` class is explained in Chapter 4.2.5. The implementation needs to create a `TransientDirectory` (see Chapter 5.3.4.3.1) at the end to ensure it is only created if a create or move operation is really executed and not already during the initial time.

In the first step, the static method `findOrCreateTransientDirectory` is initialized with the file system rood directory. With the method call `findExistingDirectory` the best matching directory is searched recursively. Within this first recursion scope the best matching directory is initialized with the root and the corresponding not contained tags in the path, in this case with all path tags. The iteration over all subdirectories begins, all calls within the loop are explicit drawn. The root directory has one subdirectory name `projects`. When the subdirectory name is contained in the path tags, then the method `bestMatchingDirectory` is called. The current directory root and the new subdirectory is compared. The best match is returned in this case the `\projects\` directory and the corresponding tags `["2011", "bayern"]`. After every best matching check the method is recursively called to find a better matching. In the third loop there is no more better matching possible and the recursion terminates. In this case the found directory is `\projects\bayern\`. For the tag `2011` is no subdirectory found in this context. Finally a new `TransientDirectory` is created and returned.

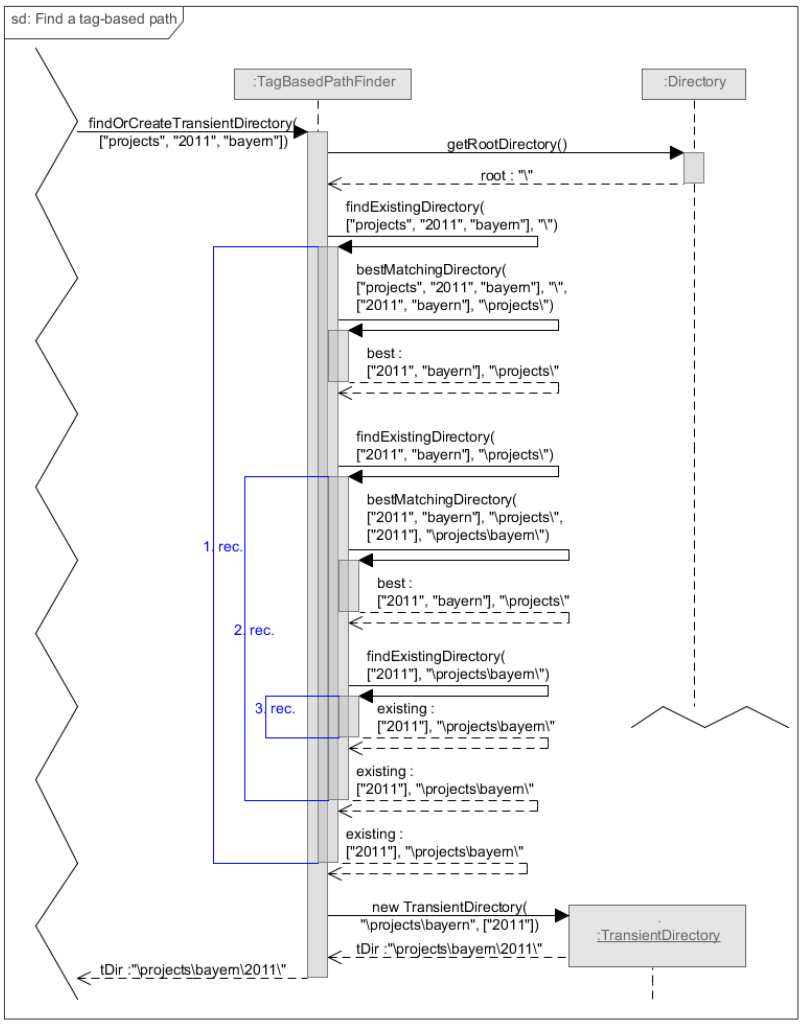

sd: Find a tag-based path
:TagBasedPathFinder
:Directory
findOrCreateTransientDirectory(
["projects", "2011", "bayern"])
getRootDirectory()
root : "\"
findExistingDirectory(
["projects", "2011", "bayern"], "\")
bestMatchingDirectory(
["projects", "2011", "bayern"], "\",
["2011", "bayern"], "\projects\")
best :
["2011", "bayern"], "\projects\"
findExistingDirectory(
["2011", "bayern"], "\projects\")
bestMatchingDirectory(
["2011", "bayern"], "\projects\",
["2011"], "\projects\bayern\")
best :
["2011", "bayern"], "\projects\"
findExistingDirectory(
["2011"], "\projects\bayern\")
existing :
["2011"], "\projects\bayern\"
existing :
["2011"], "\projects\bayern\"
existing :
["2011"], "\projects\bayern\"
1 rec.
2 rec.
3 rec.
new TransientDirectory(
"\projects\bayern", ["2011"])
:TransientDirectory
tDir :"\projects\bayern\2011\"
tDir :"\projects\bayern\2011\"

Figure 5.25: Tag based path finder sequence diagram.

5.3.5 Tricia extensions

All general tricia extensions are illustrated in this chapter. Modifications and extensions which are only concern certain methods, are explained together with the methods.

5.3.5.1 Database connection handling

Multithreaded database access is not supported for new created threads. After the thread is terminated the connection is not closed. Approximately 60 seconds after the termination the connection time out throws an exception. The `ConnectionTracker` stores all open connections in a set. The solution for this not closed connections is an extension of the `ConnectionTracker` class. The new abstract superclass is the `ConnectionValidator` (see Figure 5.26). This calls a thread which iterates in certain time intervals over the connections set in the `ConnectionTracker`. With `connection.thread` the corresponding thread is accessible and with `.isAlive()` the state can be checked. All connections are removed which corrospond to terminated threads. To keep the changes as small as possible, the extended class has abstract methods which are already implemented in the tracker class. There is no need to add any additional method only to extend it with the `ConnectionValidator`. Further it is now possible to remove connections from a thread explicit with the method `removeThreadConnection(thread)`.

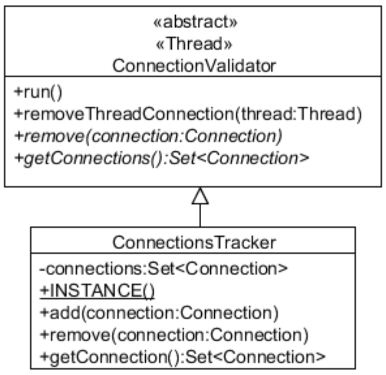

«abstract»
«Thread»
ConnectionValidator
+run()
+removeThreadConnection(thread:Thread)
+remove(connection:Connection)
+getConnections():Set<Connection>
ConnectionsTracker
-connections:Set<Connection>
+INSTANCE()
+add(connection:Connection)
+remove(connection:Connection)
+getConnection():Set<Connection>

Figure 5.26: Database connection pool manager.

5.3.5.2 JSON extension

Tricia provides an explicit `JSONAnswerStation` to send a json response which responds a `JSONObject`. This specialized type of answer stations encapsulate the transformation from a json object to a plaintext string. In Chapter 5.1.2.2 the `jQuery Dynatree Plugin` is presented which provides web-based access of hierarchical representations. This framework has an integrated lazy load interface, which reloads the directories on a click event. The interface expects a `JSONArray`, therefore an additional answer station satisfies this requirement. The name of the new station is `JsonArrayAnswerStation`.

5.3.6 Testing

This chapter describes briefly the facet import testing. Testing helps to check in a fast fashion the correctness of algorithms. Testcases define input and output data. For each test the system is initialized with the input data. The algorithm is executed and afterwards the output is compared with the defined output. In the successful case there is no difference, otherwise the test fails. This testcase implementation uses `jUnit` for testing, this is the most common testing framework. All testdata are located in xml files. Each testcase is stored in a single file.

The content of the files (Figure 5.27) describes first the input and then the desired output. Input elements are nested `directory` elements. A directory element has an attribute `name` which represents the directory name. It is really important that all testcases contain only one root directory. This is a workaround to avoid test result conflicts with already existing directories in the system like the `attachment` folder. The output related path contains for each `ContextConfiguration` a element `contextandfacets`. Within these element the context is defined with the element `context`, and each item with the element `contextelement`. This structural concept is also used for the facets. All facets are in the element `facets` which contains one or more elements of the type `facet`. Within each facet tags are defined with the element `facetelement`.

```
<testdata testdataname="testcasename">
    <input>
        <directory name="rootdirectory">
            <directory name="subdirectory"></directory>
        </directory>
    </input>
    <output>
        <contextandfacets>
            <context>
                <contextelement>tag</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>facet tag</facetelement>
                </facet>
            </facets>
        </contextandfacets>
    </output>
</testdata>
```

Figure 5.27: XML testfile datastructure.

The class diagram in Figure 5.28 illustrates the practical implementation. This diagram is simplified, only conceptual important methods and attributes are visible. All testcases are defined in the `FacetImportTestCase` class. This class extends the `FacetImportTest` class which provides all basic test handling. The `XMLFacetTestdataParser` provides encapsulated access on the testfiles. In the first step the xml testdata is parsed by the `XMLFacetTestdataParser` and converted in a internal common data structure. The database is cleaned up and initialized with the new hierarchical test file system. To import the context configuration the facet import algorithm is executed with the root directory of this testcase. After the execution the persisted resulting context configurations are compared with the defined ones. The ordering of facets are not considered in these testcases.

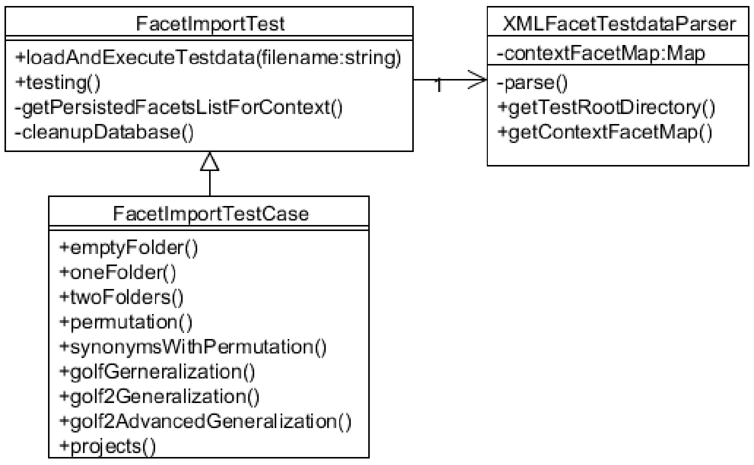


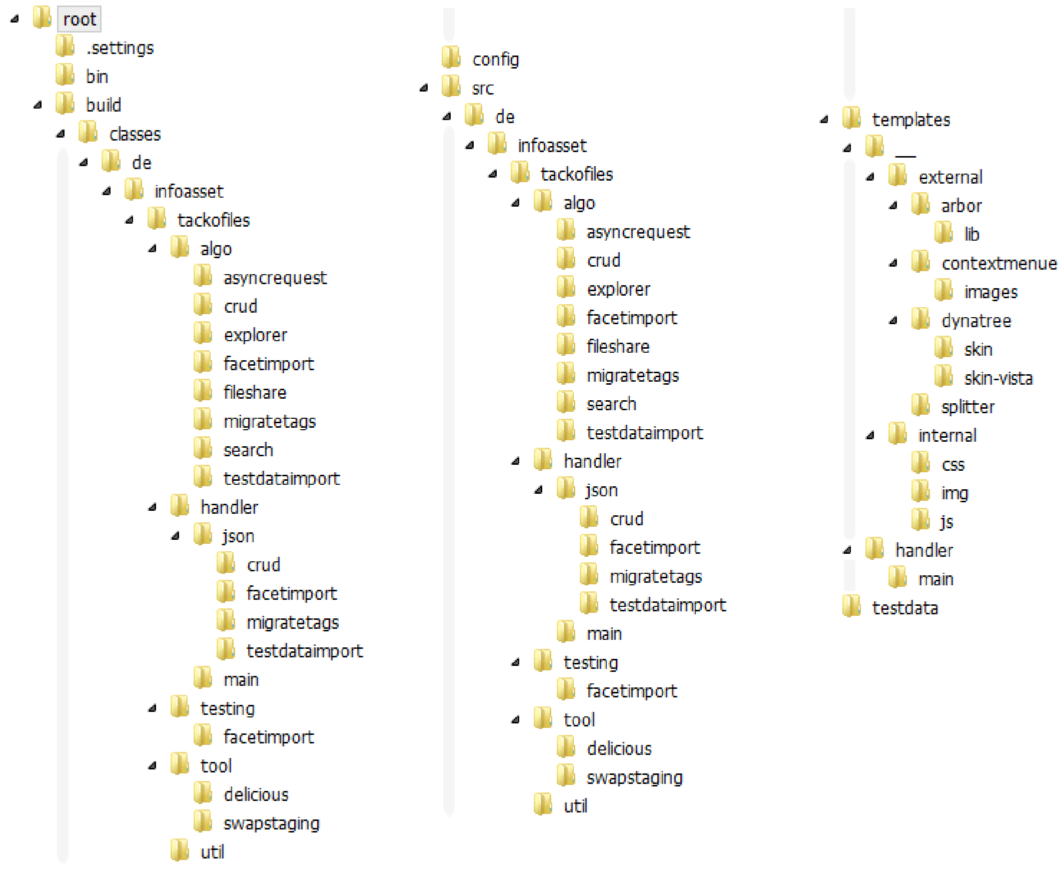
Figure 5.28: Facet import testing class diagram.

Testcases

All testcases are in the appendix. The illustrated testdata is used to test the correctness of the implemented algorithms. They cover several special cases to ensure the predicted behavior. Most testcases are strongly related to the testcase definitions in Chapter 3.6.

6 Real Data Sample

This chapter illustrates the subsumption graph and the resulting facets with a real data sample. As testdata the TACKO Files project is used (Figure 6.1). Within this hierarchical source file system are 99 directories and 345 files. The hierarchical directory tree is divided in three sections to compress it to one page. Additionally there are added some gray lines which help to find the right hierarchical level across the directory tree parts. This sample contains some special cases. Directories within the path `\root\src\` contain the uncompiled `*.java` files. The path `\root\build\classes\` contains the compiled binary resources. That means there are existing two similar subdirectory trees within different directories. Normally the `\root\bin\` path is one more similar subdirectory structure. In this sample the content of the `bin` directory is deleted to simplify it. This does not affect anything important.



```
root
    .settings
    bin
    build
        classes
            de
                infoasset
                    tackofiles
                        algo
                            asyncrequest
                            crud
                            explorer
                            facetimport
                            fileshare
                            migratetags
                            search
                            testdataimport
                        handler
                            json
                                crud
                                facetimport
                                migratetags
                                testdataimport
                            main
                        testing
                            facetimport
                        tool
                            delicious
                            swapstaging
                        util
    config
    src
        de
            infoasset
                tackofiles
                    algo
                        asyncrequest
                        crud
                        explorer
                        facetimport
                        fileshare
                        migratetags
                        search
                        testdataimport
                    handler
                        json
                            crud
                            facetimport
                            migratetags
                            testdataimport
                        main
                    testing
                        facetimport
                    tool
                        delicious
                        swapstaging
                    util
    templates
        __
            external
                arbor
                    lib
                contextmenue
                    images
                dynatree
                    skin
                    skin-vista
                splitter
            internal
                css
                img
                js
        handler
            main
    testdata
```

Figure 6.1: Hierarchical source file system.

Figure 6.2 illustrates the subsumption graph corresponding to the hierarchical source file system (Figure 6.1). The hierarchical source path `\root\src\de\infoasset\` and the path `\root\build\classes\de\infoasset\` are generalized in the subsumption graph to `root`→ `de`→ `infoasset`. Therefore the tags `src` and `classes` do not subsume any other tag. Another more complex generalization is illustrated by the tag `handler`. In the hierarchical source file system there are existing three related directories.

Path1: `\root\build\classes\de\infoasset\tackofiles\`**`handler`**`\`
Path2: `\root\src\de\infoasset\tackofiles\`**`handler`**`\`
Path3: `\root\templates\__\`**`handler`**`\`
Subsumption: `root`→ **`handler`**

Most java request handlers have a related htm template which is placed in the third path. Due to this path, the `handler` tag is directly subsumed by the `root` tag. Furthermore the `json` tag visualizes a special case with two incoming edges. In the hierarchical file system is also existing more than one `json` directory. The tag is also generalized, but the `handler` tag is more generalized. This causes the two incoming edges and means that all tags of the incoming edges together subsume the `json` tag. Figure 6.3 illustrates all facets with the corresponding facets for this sample. Finally Figure 6.4 shows the first level of the multifacet navigation concept, there are too many possibilities to represent it completely.

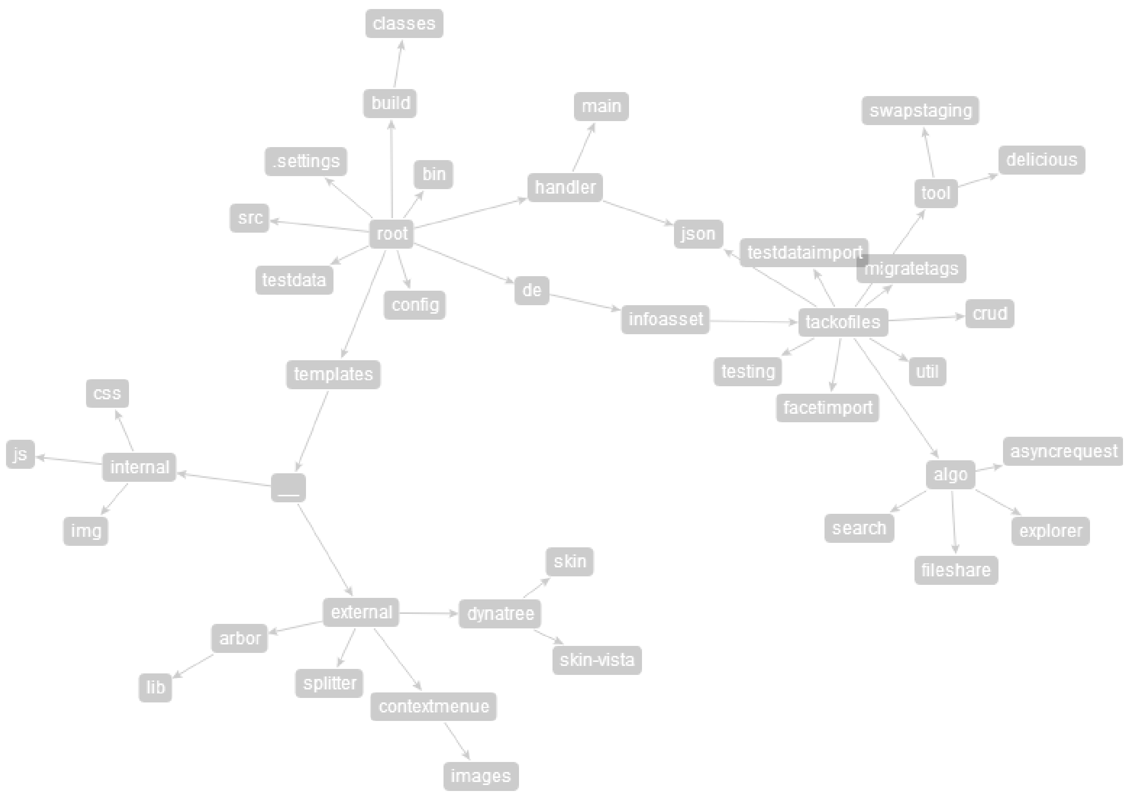
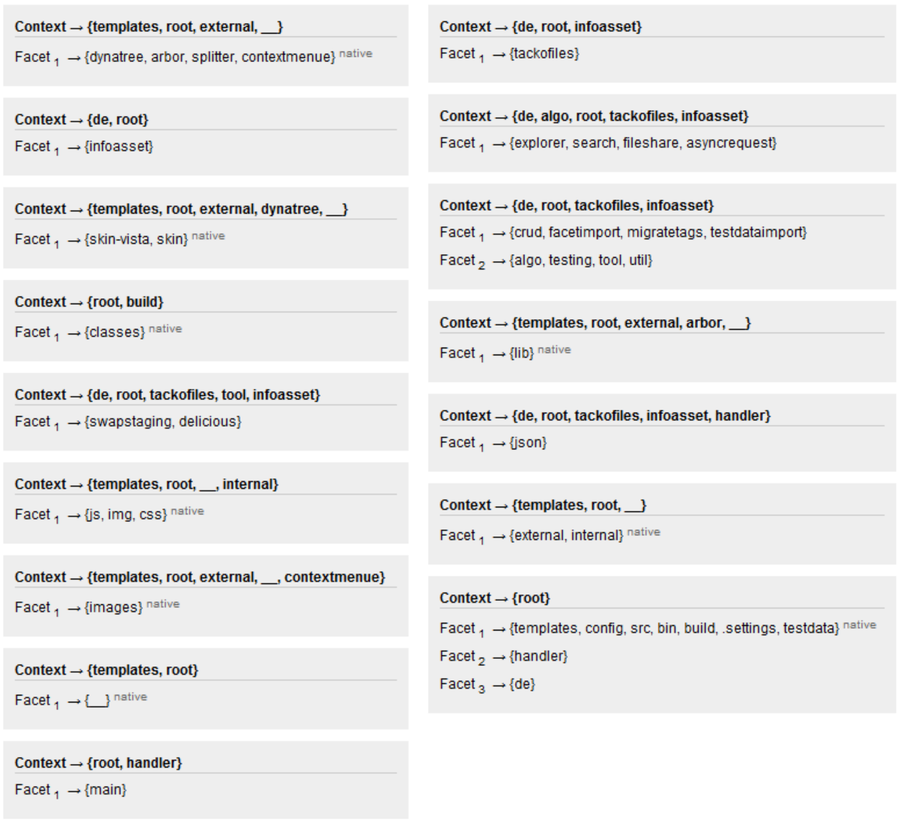


Figure 6.2: Tag subsumption graph.



Context → {templates, root, external, __}
Facet$_1$ → {dynatree, arbor, splitter, contextmenue}^{native}

Context → {de, root}
Facet$_1$ → {infoasset}

Context → {templates, root, external, dynatree, __}
Facet$_1$ → {skin-vista, skin}^{native}

Context → {root, build}
Facet$_1$ → {classes}^{native}

Context → {de, root, tackofiles, tool, infoasset}
Facet$_1$ → {swapstaging, delicious}

Context → {templates, root, __, internal}
Facet$_1$ → {js, img, css}^{native}

Context → {templates, root, external, __, contextmenue}
Facet$_1$ → {images}^{native}

Context → {templates, root}
Facet$_1$ → {__}^{native}

Context → {root, handler}
Facet$_1$ → {main}

Context → {de, root, infoasset}
Facet$_1$ → {tackofiles}

Context → {de, algo, root, tackofiles, infoasset}
Facet$_1$ → {explorer, search, fileshare, asyncrequest}

Context → {de, root, tackofiles, infoasset}
Facet$_1$ → {crud, facetimport, migratetags, testdataimport}
Facet$_2$ → {algo, testing, tool, util}

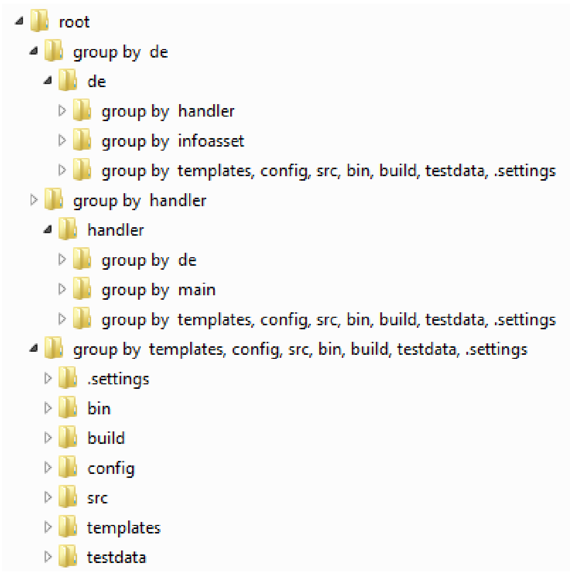
Context → {templates, root, external, arbor, __}
Facet$_1$ → {lib}^{native}

Context → {de, root, tackofiles, infoasset, handler}
Facet$_1$ → {json}

Context → {templates, root, __}
Facet$_1$ → {external, internal}^{native}

Context → {root}
Facet$_1$ → {templates, config, src, bin, build, .settings, testdata}^{native}
Facet$_2$ → {handler}
Facet$_3$ → {de}

Figure 6.3: Context with corresponding facets.



```
◢ root
  ◢ group by de
    ◢ de
      ▷ group by handler
      ▷ group by infoasset
      ▷ group by templates, config, src, bin, build, testdata, .settings
  ▷ group by handler
    ◢ handler
      ▷ group by de
      ▷ group by main
      ▷ group by templates, config, src, bin, build, testdata, .settings
  ◢ group by templates, config, src, bin, build, testdata, .settings
    ▷ .settings
    ▷ bin
    ▷ build
    ▷ config
    ▷ src
    ▷ templates
    ▷ testdata
```

Figure 6.4: First levels of the multifacet navigation.

The subsumption graph (Figure 6.5) of the tricia repository (Figure 6.6) visualizes another sample . The represented graph is not helpful to find a subsumption of a certain tag, but it illustrates the general structure. The root node is marked with an orange cycle. The name is not like expected the name of the root directory, because the graph layouting algorithm has covered the root tag with another tag. The example hierarchical file system contains approximately 5,500 folders and 28,000 files. The files do not affect the subsumption graph. Calculating this subsumption graph together with the facet import takes more than one minute. According to the graph structure, it is recognizable that a few tags subsume a huge amount of tags.

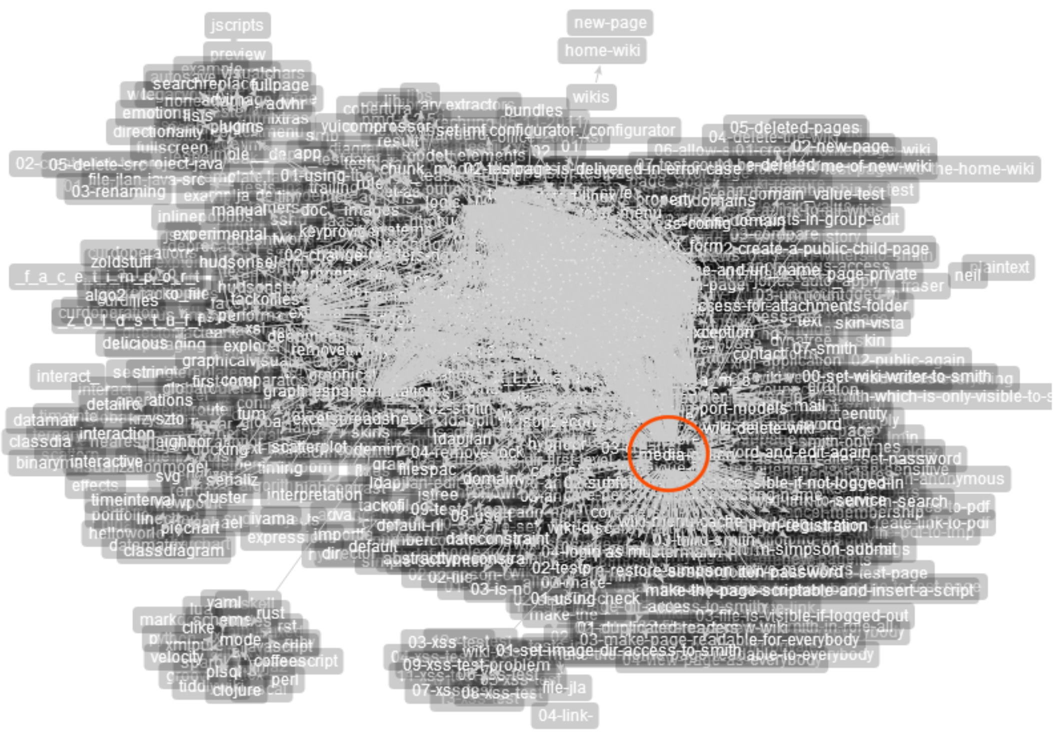
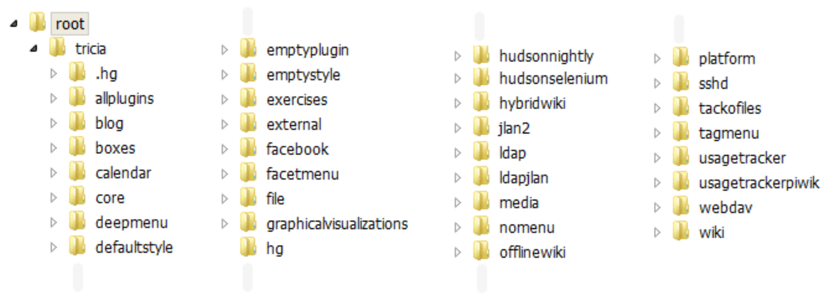


Figure 6.5: Subsumption graph of the tricia repository.



- root
  - tricia
    - .hg
    - allplugins
    - blog
    - boxes
    - calendar
    - core
    - deepmenu
    - defaultstyle
    - emptyplugin
    - emptystyle
    - exercises
    - external
    - facebook
    - facetmenu
    - file
    - graphicalvisualizations
    - hg
    - hudsonnightly
    - hudsonselenium
    - hybridwiki
    - jlan2
    - ldap
    - ldapjlan
    - media
    - nomenu
    - offlinewiki
    - platform
    - sshd
    - tackofiles
    - tagmenu
    - usagetracker
    - usagetrackerpiwik
    - webdav
    - wiki

Figure 6.6: First directory level of the tricia repository.

7 Summary and Outlook

This chapter summarizes this thesis (see Chapter 7.1) and gives a briefly outlook (see Chapter 7.2).

7.1 Summary

Aim of this work was to design and implement a TACKO Files model which couples tag-based systems with hierarchical file systems. This summary is structured according to TACKO Files process. First of all, the tags are imported from the hierarchical file system. There are two algorithms, one for a simple tag import and another one for the facet import. The simple one works incremental. A facet import considers the whole hierarchical file system for every import.

All imported facet tags are browsable with the native facet navigation or with the multifaceted navigation concept. In the default case, the native facet navigation is the best choice. The represented structure is compact and easy to understand. The multifaceted navigation provides an advanced navigation concept. If it exists more than one facet for a certain context, all context related documents can be represented corresponding to the selected facet. This allows regarding the resources form different perspectives. Selecting the facets for a certain context needs an additionally synthetic layer in the navigation structure. In real data samples the folder names of the synthetic layer are partly really long. This influences the usability negative. On the other hand the concept is really powerful a huge amount of navigation paths are offered. Resources are structured more according to the needs of the users and can be found faster. Additionally a document count option per tag is offered. This option is possible in both navigation concepts. The option can be really helpful for searching tags which are assigned to a huge amount of documents. To keep the concept as simple as possible, this option is deactivated by default.

The described concepts are accessible via a mountable network device and a web interface. Additionally all CRUD operations are provides in the web-based interface. They can be selected with a context menu. A right click on the desired tag or document opens the context menu. Every operation is mapped to the hierarchical source file system and executed there. After an operation the facets are recalculated.

In summary it is possible to couple a tag-based system with a hierarchical files system with some restrictions. Especially some CRUD operations based on the multifaceted view are not clearly mappable back to the hierarchical source file system. The new tag-based navigation concepts offer an additional fashion to browse a hierarchical file system in an effective way. It combines the advantages of hierarchical based navigation with the advantages of the tag-based navigation. The single location restriction of the hierarchically concept is solved. Furthermore the tags are clearly hierarchical structured. All concepts are implemented prototypically in the TACKO Files plugin and need to be proved with more real data samples.

7.2 Outlook

In the future, the facet import algorithm could be modified that in a way it is possible to consider only change directories of the file system. With an increasing amount of directories within the hierarchical source file system, this leads to performance advantages. Implementing the CRUD operations also in the mountable network device is one more step to a fully integrated TACKO Files plugin. To evaluate and improve the usability, third person's usability tests are necessary. Currently only documents are browsable resources. In general the concept is easily extensible, that also other resources could be optionally included in these navigation concepts.

8 Glossary

CRUD operation	Create, read, update and delete operations.
path	Describes the placement of a recourse in a hierarchical file system.
path tags	Path tags are the corresponding tags of the tag path and ordered like tag path.
SMB	Server Message Block protocol it's a common communication protocol for mounting network devices.
TACKO	TAg-based Content dependent Knowledge Organisation.
TACKO Files	Hierarchical file system extension for the TACKO model.
tag path	Represents a path of a tag based navigation concept.
Tricia	A commercial web-based enterprise collaboration platform. http://www.infoasset.de/
UI	User Interface.

Appendix

Testdata

```
<?xml version="1.0"?>
<testdata testdataname="oneFolder">
    <input>
        <directory name="root">
            <directory name="projects"></directory>
        </directory>
    </input>
    <output>
        <contextandfacets>
            <context>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>projects</facetelement>
                </facet>
            </facets>
        </contextandfacets>
    </output>
</testdata>
```

Figure 8.1: One folder xml textdata.

```
<?xml version="1.0"?>
<testdata testdataname="twoFolder">
    <input>
        <directory name="root">
            <directory name="projects">
                <directory name="internal"></directory>
            </directory>
        </directory>
    </input>
    <output>

        <contextandfacets>
            <context>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>projects</facetelement>
                </facet>
            </facets>
        </contextandfacets>

        <contextandfacets>
            <context>
                <contextelement>root</contextelement>
                <contextelement>projects</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>internal</facetelement>
                </facet>
            </facets>
        </contextandfacets>

    </output>
</testdata>
```

Figure 8.2: Two folder xml testdata.

```
<?xml version="1.0"?>
<testdata testdataname="permutation">
    <input>
        <directory name="root">
            <directory name="lectures">
                <directory name="projects"></directory>
            </directory>
            <directory name="projects">
                <directory name="lectures"></directory>
            </directory>
        </directory>
    </input>
    <output>

        <contextandfacets>
            <context>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>projects</facetelement>
                    <facetelement>lectures</facetelement>
                </facet>
            </facets>
        </contextandfacets>

    </output>
</testdata>
```

Figure 8.3: Permutation xml testdata.

```
<?xml version="1.0"?>
<testdata testdataname="identicalNamesWithPermutation">
    <input>
        <directory name="root">
            <directory name="finance">
                <directory name="projects">
                    <directory name="projectbudget"></directory>
                </directory>
            </directory>
            <directory name="projects">
                <directory name="finance">
                    <directory name="projectbudget"></directory>
                </directory>
            </directory>
        </directory>
    </input>
    <output>

        <contextandfacets>
            <context>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>finance</facetelement>
                    <facetelement>projects</facetelement>
                </facet>
            </facets>
        </contextandfacets>

        <contextandfacets>
            <context>
                <contextelement>finance</contextelement>
                <contextelement>projects</contextelement>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>projectbudget</facetelement>
                </facet>
            </facets>
        </contextandfacets>

    </output>
</testdata>
```

Figure 8.4: Identical names with permutation xml testdata.

```
<?xml version="1.0"?>
<testdata testdataname="golfGeneralization">
    <input>
        <directory name="root">
            <directory name="cars">
                <directory name="golf"></directory>
            </directory>
            <directory name="sport">
                <directory name="golf"></directory>
            </directory>
        </directory>
    </input>
    <output>
        <contextandfacets>
            <context>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>cars</facetelement>
                    <facetelement>sport</facetelement>
                </facet>
                <facet>
                    <facetelement>golf</facetelement>
                </facet>
            </facets>
        </contextandfacets>
    </output>
</testdata>
```

Figure 8.5: Golf generalization xml testdata.

```
<?xml version="1.0"?>
<testdata testdataname="golf2Generalization">
    <input>
        <directory name="root">
            <directory name="cars">
                <directory name="golf">
                    <directory name="golf2"></directory>
                </directory>
            </directory>
            <directory name="sport">
                <directory name="golf"></directory>
            </directory>
        </directory>
    </input>
    <output>

        <contextandfacets>
            <context>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>cars</facetelement>
                    <facetelement>sport</facetelement>
                </facet>
                <facet>
                    <facetelement>golf</facetelement>
                </facet>
            </facets>
        </contextandfacets>

        <contextandfacets>
            <context>
                <contextelement>cars</contextelement>
                <contextelement>golf</contextelement>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>golf2</facetelement>
                </facet>
            </facets>
        </contextandfacets>

    </output>
</testdata>
```

Figure 8.6: Golf2 generalization xml testdata.

```xml
<?xml version="1.0"?>
<testdata testdataname="golf2AdvancedGeneralization">
    <input>
        <directory name="root">
            <directory name="cars">
                <directory name="golf">
                    <directory name="golf2"></directory>
                </directory>
            </directory>
            <directory name="sport">
                <directory name="golf"></directory>
            </directory>
            <directory name="geographie">
                <directory name="golf">
                    <directory name="aden"></directory>
                    <directory name="mexiko"></directory>
                    <directory name="nepal"></directory>
                </directory>
            </directory>
        </directory>
    </input>
    <output>

        <contextandfacets>
            <context>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>cars</facetelement>
                    <facetelement>sport</facetelement>
                    <facetelement>geographie</facetelement>
                </facet>
                <facet>
                    <facetelement>golf</facetelement>
                </facet>
            </facets>
        </contextandfacets>

        <contextandfacets>
            <context>
                <contextelement>cars</contextelement>
                <contextelement>golf</contextelement>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>golf2</facetelement>
                </facet>
            </facets>
        </contextandfacets>

        <contextandfacets>
            <context>
                <contextelement>geographie</contextelement>
                <contextelement>golf</contextelement>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>aden</facetelement>
                    <facetelement>mexiko</facetelement>
                    <facetelement>nepal</facetelement>
                </facet>
            </facets>
        </contextandfacets>

    </output>
</testdata>
```

Figure 8.7: Golf2 advanced generalization xml testdata.

```xml
<?xml version="1.0"?>
<testdata testdataname="projects">
    <input>
        <directory name="root">
            <directory name="projects">
                <directory name="bayern">
                    <directory name="augsburg">
                        <directory name="2011"></directory>
                        <directory name="2012"></directory>
                    </directory>
                    <directory name="munich">
                        <directory name="2011"></directory>
                        <directory name="2012"></directory>
                    </directory>
                </directory>
                <directory name="niedersachsen">
                    <directory name="braunschweig">
                        <directory name="2011"></directory>
                        <directory name="2012"></directory>
                    </directory>
                </directory>
            </directory>
        </directory>
    </input>
    <output>
        <contextandfacets>

            <context>
                <contextelement>projects</contextelement>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>niedersachsen</facetelement>
                    <facetelement>bayern</facetelement>
                </facet>
                <facet>
                    <facetelement>2012</facetelement>
                    <facetelement>2011</facetelement>
                </facet>
            </facets>
        </contextandfacets>

        <contextandfacets>
            <context>
                <contextelement>projects</contextelement>
                <contextelement>niedersachsen</contextelement>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>braunschweig</facetelement>
                </facet>
            </facets>
        </contextandfacets>

        <contextandfacets>
            <context>
                <contextelement>projects</contextelement>
                <contextelement>bayern</contextelement>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>augsburg</facetelement>
                    <facetelement>munich</facetelement>
                </facet>
            </facets>
        </contextandfacets>

        <contextandfacets>
            <context>
                <contextelement>root</contextelement>
            </context>
            <facets>
                <facet>
                    <facetelement>projects</facetelement>
                </facet>
            </facets>
        </contextandfacets>

    </output>
</testdata>
```

Figure 8.8: Projects xml testdata.

Bibliography

[Arr03] Manuel Arriag. *Going beyond the hierarchical file system: a new approach to document storage and retrieval.* 2003.

[ASB11] Seyyed Hamidreza Afzali Zahra Zabardast Ali Sajedi Badashian, Mehregan Mahdavi. *Supporting Multiple Categorization using Conceptual File Management*. American Journal of Scientific Research, 2011.

[DC11] Samizdat Drafting and Co. Arbor introduction. `http://arborjs.org/introduction`, 2011. [Online; accessed 13th of June 2012].

[GS10] Jon Atle Gulla Geir Solskinnsbakk. *A Hybrid Approach to Constructing Tag Hierarchies*. Norwegian University of Science and Technology, 2010.

[Hea06] Marti A. Hearst. *Design Recommendations for Hierarchical Faceted Search Interfaces*. School of Information, UC Berkeley, 2006.

[HR07] Marti A. Hearst and Daniela Rosner. *Tag Clouds: Data Analysis Tool or Social Signaller?* University of California, Berkeley, 2007.

[jF12] The jQuery Foundation. jquery. `http://jquery.com/`, 2012. [Online; accessed 15th of June-2012].

[JPGB05] William Jones, Ammy Jiranida Phuwanartnurak, Rajdeep Gill, and Harry Bruce. *Don't Take My Folders Away! Organizing Personal Information to Get Things Done*. The Information School, University of Washington, 2005.

[MNdbD06] Cameron Marlow, Mor Naaman, danah boyd, and Marc Davis. *HT06, Tagging Paper, Taxonomy, Flickr, Academic Article, ToRead*. Yahoo! Research Berkele and UC Berkeley School of Information, 2006.

[MNS12] Florian Matthes, Christian Neubert, and Alexander Steinhoff. *Multi-faceted context-dependent knowledge organisation with TACKO*. 12th International Conference on Knowledge Management and Knowledge Technologies - i-KNOW 2012, Messe Congress Graz, Austria, 2012.

[MTT$^+$09] Gary Marchionini, Daniel Tunkelang, Michael Thelwall, Michael G. Christel, Ryen W. White, Resa A. Roth, and R. David Lankes. *Synthesis Lectures on Information - Concepts, Retrieval, and Services*. University of North Carolina, Chapel Hill, 2009.

[Rud06] Christiane Rudlof. *Handbuch Sofware-Ergonomie - Usability Engineering*. Unfallkasse Post und Telekom, 2006.

[SAZ12] Ali Sajedi, Seyyed Hamidreza Afzali, and Zahra Zabardast. *Can You Retrieve a File on the Computer in your First Attempt? Think to a New File Manager for Multiple Categorization of Your Personal Information*. Personal Information Management - PIM 2012, 2012.

[SB06] Max Völkel Stephan Bloehdorn. *TagFS - Tag Semantics for Hierarchical File Systems*. Institute AIFB, University of Karlsruhe, 2006.

[Smi08] Gene Smith. *Tagging: People-Powered Metadata for the Social Web*. New Riders, Berkeley, CA 94710, 2008.

[Wis11] Kilian Wischer. *Design and Implementation of a File-based Solution to Provide Offline Access to an Enterprise 2.0 Platform (bachelor thesis)*. Technical University of Munich, 2011.

List of Figures

Listings