

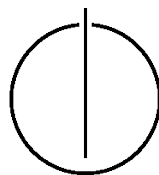
FAKULTÄT FÜR INFORMATIK

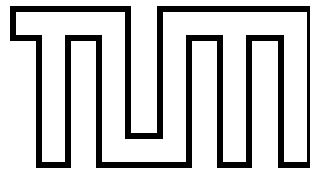
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Wirtschaftsinformatik

**Extension of an Enterprise 2.0 Platform to
Support User Activity Awareness**

Rainer Niedermayr





FAKULTÄT FÜR INFORMATIK

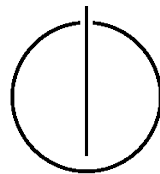
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Wirtschaftsinformatik

Erweiterung einer Enterprise 2.0 Plattform um
Funktionen für die Sichtbarkeit von Benutzeraktivitäten

Extension of an Enterprise 2.0 Platform to Support User
Activity Awareness

Author: Rainer Niedermayr
Supervisor: Prof. Dr. Florian Matthes
Advisor: Dr. Thomas Büchner
Submission Date: August 24, 2011



I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Munich, August 24, 2011

Rainer Niedermayr

Abstract

More and more companies use enterprise 2.0 platforms to enable collaboration and management of their knowledge. Most platforms have a component which allows for tracing changes. Thereby, it facilitates users to be aware of the activities of others and to keep up to date with the latest information.

This thesis is about the development of such an awareness component for the platform Tricia. It includes the detection of the requirements, the design decisions, the creation of the graphical user interfaces, and the efficient implementation in its core.

The developed component provides both pull and push functionalities: Users can watch objects and retrieve a list including the recent changes of watched or of all accessible objects (pull). Mail notifications (push) which are sent periodically and summarise changes round it off.

Keywords: Awareness, Change Tracking, Enterprise 2.0, Tricia

Contents

Abstract	iv
1 Introduction	1
1.1 Social Software	1
1.2 Event-based Awareness	2
1.3 Goal of this Thesis	3
1.4 Structure of this Thesis	3
2 Fundamentals	5
2.1 Architecture of Tricia	5
2.1.1 Model	5
2.1.2 View	8
2.1.3 Controller	8
2.1.4 Plugins	9
2.2 Model Representation of User Activity	10
2.3 Full-text Indexing	11
3 Analysis and Design	12
3.1 Requirements	12
3.2 UI Presentation of Changes	13
3.2.1 AbstractGroupMembershipChange	14
3.2.2 DomainValueChange	15
3.2.3 SimpleValueChange	16
3.2.4 TagsChange	17
3.2.5 HybridPropertyChange	18
3.2.6 RichStringChange	19
3.2.7 RoleChange	22
3.3 Aggregation of Changes	23
3.4 Dashboard	24
3.4.1 Realisations in other Systems	25
3.4.2 Planned Implementation in Tricia	25
3.5 Recent Changes	27
3.5.1 Realisation in other Systems	27
3.5.2 Planned Implementation in Tricia	30
3.6 Mail Notification	32
3.6.1 Realisations in other Systems	32
3.6.2 Planned Implementation in Tricia	32
4 Implementation	34

4.1	Presentation of Changes	34
4.1.1	Template	34
4.1.2	Substitution	35
4.2	HTML comparison with TriciaDiff	36
4.2.1	StructureAnalyzer	37
4.2.2	StructureConsolidator	39
4.2.3	HtmlGenerator	42
4.2.4	Abbreviator	43
4.3	Aggregation of Similar Change Sets	44
4.4	Change Awareness	46
4.4.1	Event Stream	46
4.4.2	New User Interfaces	49
4.4.3	Mail Notification Service	53
5	Summary	57
	Bibliography	59
	List of figures	62
	Listings	63

1 Introduction

The amount of available information grows at a rapid pace. In addition, the complexity and dynamics of any business is also on the increase. The research confirms productivity problems in information management within enterprises. These are caused by searching, duplicate work, inconsistencies, and outdated or wrong information. [infa]

Therefore, technology is needed, which facilitates successful collaboration. Social software provides the central functionalities such as information sharing, knowledge of group and individual activity, and coordination. [DB92]

1.1 Social Software

Coates (2005) describes the term social software as “software which supports, extends, or derives added value from, human social behavior”. [Coa05] Social software can be structured by its basic functionalities:

- The **identity management** allows users to present themselves on the web.
- The **relationship management** allows establishing and maintaining relationships.
- The **information management** allows handling (finding, evaluating and managing) of information available on the web.

Figure 1.1 is known as the “social software triangle”. It contains the web 2.0 application categories including blogs, wikis, social tagging and bookmarking applications, social networking, and instant messaging, which are classified by the three basic functionalities.

Enterprise 2.0 is the use of these applications within an organisation to connect the employees. McAfee summarises the six components of enterprise 2.0 technologies in the acronym SLATES (search, links, authoring, tags, extensions, signals): [McA06]

- A **search** function must exist thanks to which users should find what they are looking for.
- Web pages are connected via **links**, users must be able to add and edit links.
- **Authoring**: Users can contribute knowledge, insights, experiences, comments... The content can be cumulative (blogs: individual posts accumulate over time) or iterative (wikis: many people work on the same data).
- **Tags** are simple one-word descriptions to characterise the content. The categorisation system formed by the tags is called a “folksonomy”.
- **Extensions** provide intelligent content suggestions by mining patterns and user activity.



Figure 1.1: Social software triangle by Koch/Richter [KR07]

- **Signals** describe a component which keeps users up to date by sending them notifications if and when new or edited content of their interest appears.

[KR07] [ZDN]

1.2 Event-based Awareness

Platforms with a signal component are also called “social awareness systems”. Their purpose is to “help connected individuals or groups to maintain a peripheral awareness of the activities and the situation of each other.” [ISG09]

The users can take on the roles of senders or receivers. Senders add or update information on the platform and want to let other interested people know about that. The receivers are people or systems, who receive notifications concerning these changes. The decoupling of these two groups is an important concept behind awareness systems. Senders and the receivers are independent and don’t know have to know each other. They are separated by a system in the middle which acts as an intermediary information channeller. This system is aware of all changes and tries to match changes in the user’s interests to identify the target groups. Then it distributes the notifications to the recipients.

An event-based awareness system reduces the problems of information overloading and distribution. Information overloading means that recipients receive too much information, including irrelevant information. The term information distribution describes the problem for the sender to identify the appropriate target group (with the risk of forgetting to inform people about something important or to overload them with irrelevant information). In addition, the awareness system allows people to discover information, they may not have known existed.

Khronika was the first implementation of such an event-based notification system. It was developed in the early 1990s by Rank Xerox EuroPARC, the European research center of Xerox in Cambridge. Khronika implements a shared network server that notifies people about events. The server acts as an intermediary between senders and recipients. Senders are in charge of feeding the system with information and keeping it updated whereas the recipients determine what kind of information they are interested in. The recipients do this by submitting daemons which contain constraint sets expressing their personal interests. Khronika monitors and stores events from the senders. Then it matches the events with the recipient's interests and generates notifications which are delivered to the identified target group.

[Löv91] [DBMM93] [RM09]

1.3 Goal of this Thesis

Tricia¹ is an enterprise 2.0 software system developed by infoAsset. The platform covers the following features:

- Hybrid wiki: A wiki is a website which is collaboratively used by multiple users. It contains pages which can be edited by anyone. The concept of hybrids supplements wikis by providing possibilities to add structured content through key-value-attributes.
- Blogs: A blog contains posts which are displayed in a reverse-chronological order.
- File and directory sharing: It is possible to upload, view and download files.
- Social networking: Users can be members in groups and inherit access policies.

[infb]

These features improve the collaboration of employees in a company. However, although a prototype view which shows the latest ten user edits exists, no major awareness functionalities are available as yet.

The goal of this thesis is to extend the core of Tricia by an awareness component, which allows changes to be traced. The component will help users to be aware of the activities of others and facilitate them to keep up to date with the latest information. This thesis elucidates the detected requirements, the design decisions taken, and the implementation.

1.4 Structure of this Thesis

The structure of this thesis is as follows:

Chapter 1 gives a theoretical reflection on awareness in social software. It describes the concept of the decoupling of sender and receiver and points out the additional value of an awareness component for an enterprise 2.0 platform.

¹<http://www.infoasset.de/>

Chapter 2 covers the fundamentals of Tricia. It focuses on concepts which are relevant for the implementation of the new awareness functionalities.

Chapter 3 describes the results of the analysis and design phases. It contains the detected requirements for an awareness component embedded in Tricia, similar implementations in other enterprise 2.0 systems, design decisions, and drafts for the user interfaces.

Chapter 4 presents the implementation details of the developed components.

Chapter 5 summarises the results of this thesis and shows further possible extensions.

2 Fundamentals

This chapter gives an insight into Tricia and into the Lucene library. It focuses on concepts which are relevant for the implementation of the new awareness functionalities.

2.1 Architecture of Tricia

Tricia's architecture realises the model-view-controller pattern with the following components:

- **model:** So-called assets are mapped to an object/relational persistence database.
- **view:** A template system is in charge of the presentation.
- **controller:** Handlers are an abstraction for defining control flows.

2.1.1 Model

Tricia follows a data model driven approach. Important concepts of this approach are the entities and so called mixins.

Entities

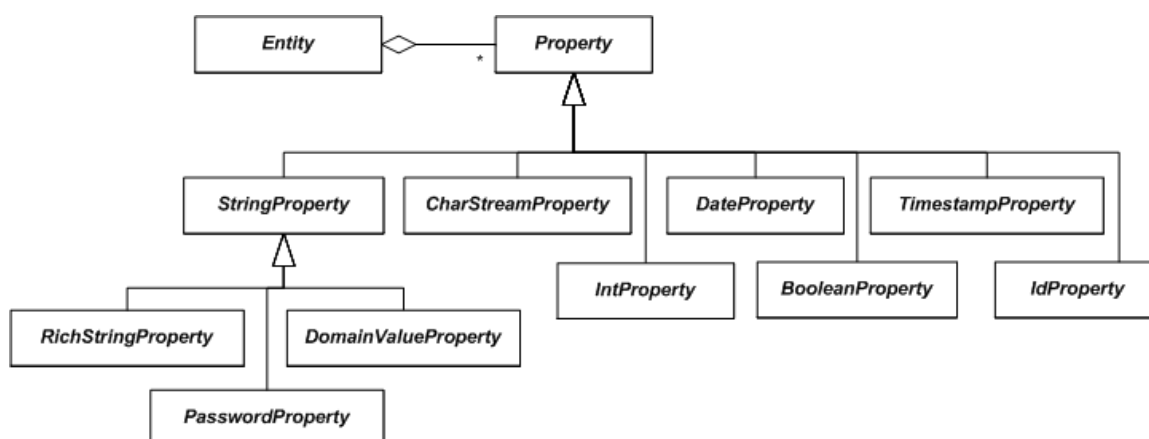
Domain objects are represented by subclasses of `Entity`. `Entity` itself inherits the more generic class `Asset`. Entities can have properties, methods and relations to other assets. Different property types such as `StringProperty`, `BooleanProperty`, `IntProperty`, `DateProperty`... exist (see figure 2.1). Relations between assets can be expressed with roles whereas a role definition in an entity specifies the other end of an association. The multiplicity is defined by the used role class: A `OneRole` association end references one single entity, a `ManyRole` models a multi-valued association.

Entities which are persisted to the database inherit the entities' subclass `PersistentEntity`. Each persistent entity class is mapped onto a database table with a column for each property. Any persisted object can be uniquely identified by its `UUID`.

Mixins

One advance of inheritance is the reuse of an implementation. However, in Java a class cannot inherit more than one super class. This limitation avoids a comprehensive reuse model as desired if it is not possible to partition the functionality into one inheritance structure. Tricia works around this limitation via so-called *mixins*. A mixin inherits `Asset`

¹<http://www.infoasset.de/file/attachments/wikis/javadoc-import-wiki/platform-documentation-entitiespart1doc/entity-property.png>

Figure 2.1: Property types in Tricia¹

and can define properties, methods and relations to other assets. It extends a base entity; the functionality can be used by adapting the base entity to the mixin type. Two types of mixins exist; the mandatory mixins are assigned statically to an entity, optional mixins can be assigned and/or removed at runtime to/from an object.

Important mandatory mixin types in Tricia are:

- **Linkable**: This mixin holds an entities' incoming and outgoing links.
- **ReadProtected**: It concerns read rights on entities.
- **Modifiable**: It concerns write rights on entities.
- **Versionable**: This mixin keeps a version history of an entity. It is assigned to each persistent entity by default.
- **InSpace**: It defines the space an entity is contained in.

Spaces

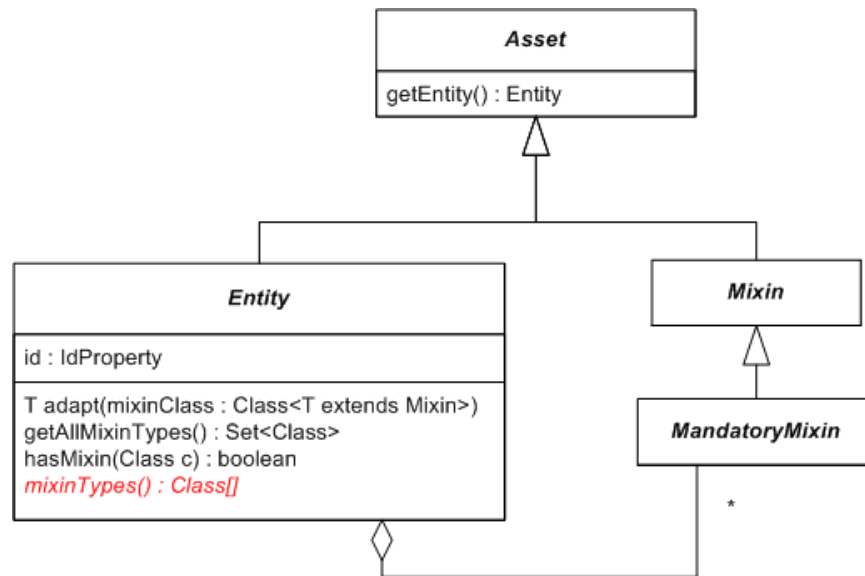
A space is a kind of a container object. It has a name and contains logically coherently data. A space is represented by a persistent entity with the mixin `Space`. Each content item has the mixin `InSpace`.

Known types of spaces are the wiki which contains wiki pages, the blog containing blog entries and the directory containing subdirectories and documents (files).

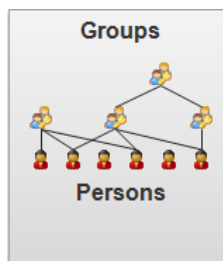
Users and Groups

The concept of groups exists in Tricia. Groups describe organizational roles and structures and help to administrate users. A group can be described as a composite; it has users

²<http://www.infoasset.de/file/attachments/wikis/javadoc-import-wiki/platform-documentation-entitiespart3doc/mixinAsset.png>

Figure 2.2: Classes Asset, Entity and Mixin²

and/or groups as members. A membership grants the member all rights which are stipulated for the group. The two built-in groups “Registered Users” and “Everybody” already exist.

Figure 2.3: Group structure in Tricia³

Access Policies

Tricia implements access control lists which allow configurable policies: Each entity has a list of editors that are able to view and edit it, and a list of readers that can only view and find it. Entities inherit the access rights of their space or their parent object (if there is any), but it is possible to add editors and to override readers. The specified readers or editors are users and/or groups.

³<http://www.infoasset.de/file/attachments/wikis/hilfe/inhalte-und-bereiche/Content%20and%20Spaces.PNG>

2.1.2 View

Tricia has its own templating language which enforces the separation of presentation and logic.

Templates

A template is an html file with formatted content which contains placeholders. The placeholders are replaced at runtime with the actual content. In the template these are marked by the character "\$".

The following example for a simple template file contains the placeholder \$time\$ which will be replaced by the handler:

```
<html>
  <head>

  </head>
  <body>
    Current time: $time$
  </body>
</html>
```

Listing 2.1: Simple template file

Substitutions

For each placeholder there is a substitution specified in the handler class. The substitution logic is expressed by inner Java classes. Different types can be used:

- The **print substitution** replaces a placeholder with a string value.
- The **conditional substitution** is used to show or to hide a block of content.
- The **list substitution** allows replacing placeholders repeatedly for all items of a list.
- The **template substitution** replaces a placeholder with the evaluation of another template. It permits the building of complex nested structures.

```
template.put("time", new PrintSubstitution()
{
    @Override
    protected String print()
    {
        return getTimeAsString();
    }
});
```

Listing 2.2: Simple print substitution

2.1.3 Controller

The third component of the MVC pattern is the controller, which is responsible for the program flow.

Handler

The controller is represented by handlers. A handler is in charge of HTTP requests. At the beginning of the handling process the session is identified and the client context initialised. Then the dispatcher identifies the responsible handler class. This class checks at first if the session is allowed to access it. After that - if the check returns a positive result - the business logic is executed. At the end, an evaluated page as response is returned or the request is forwarded to another handler.

The classes `SessionLocal` and `Parameters` offer useful static methods which allow reading out the entity of the currently logged in user, the referrer url and get parameters encoded in the url.

2.1.4 Plugins

Tricia is built in a modular way. An application consists of a core and can be extended by one or more plugins. A plugin can use functionality from other plugins, therefore, these form a dependency relationship. [BMN10]

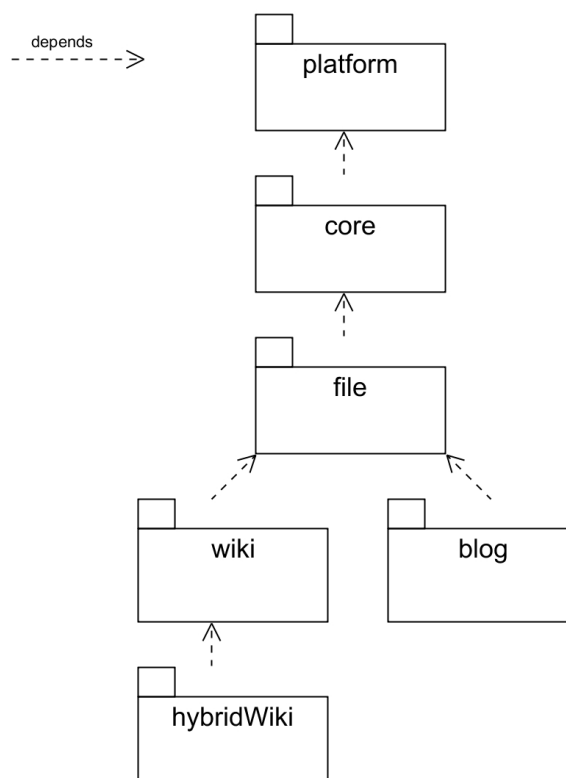


Figure 2.4: Important Tricia plugins

2.2 Model Representation of User Activity

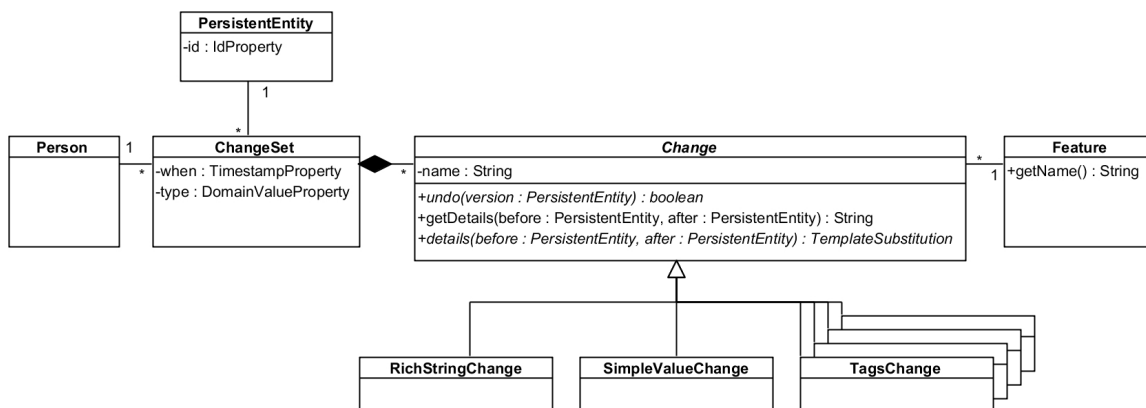


Figure 2.5: Model representation of user activity

The user activity is represented by change sets, which are created each time, a user executes a write operation on a persistent entity. The class `ChangeSet` is a persistent entity too and can therefore be identified by its ID property. The class holds the time stamp of the activity and records the action type. The action type, represented by `ChangeSetType`, is set on

- `_add`, if the user created a new persistent entity
- `_edit`, if the user edited features of an entity (examples for well known features are: content of a page, description fields, tags, hybrid attributes)
- `_remove`, if the user deleted an entity
- `_undelete`, if the user restored a deleted entity.

Furthermore, a change set has an association to the concerned changed persistent entity and to the users' entity. Both associations are realised as `OneRoles`.

If the type of the change set is `_edit`, then the set contains at least one change. All changes are held in a property as a JSON⁴ serialised string. The method `getChanges()` allows for retrieval of these by deserialising the string. It returns a list whose items are instances of the abstract class `Change`. Several concrete classes inherit it (see also figure 3.2):

- `AbstractGroupMembershipChange`: The membership of a user was edited.
- `DomainValueChange`: The value of an enumeration property was edited.
- `SimpleValueChange`: The value of a property of a primitive type was edited.
- `TagsChange`: Tags were added or removed to/from an entity.
- `HybridPropertyChange`: A key-value attribute was edited.

⁴JSON = JavaScript Object Notation; a text-based user-readable standard for data interchange

- `RichStringChange`: A formatted text was edited.
- `RoleChange`: The write and/or read access rights of an entity were edited.

Each change concerns exactly one feature and can be identified by the the combination of the change set's ID and the name of the feature. The change class holds the value of the feature before the edit. In addition, it implements the (in the super class declared) `undo()` method, which is used to revert an edit. A change set can roll an entity back to an earlier state by invoking the `undo()` method on all contained changes. Another important method which is implemented in the subclasses of `Change` is `details()`. It returns a template substitution, which is used to generate an html overlay to display the change graphically. The method `getDetails()` returns the template of `details()` as a converted html string.

The class `EventManager` is responsible for creating the change sets and for storing these in the database. For this action it uses the static method `ChangeSet.getDifferences()`, which returns the changes between two states of an entity.

2.3 Full-text Indexing

For the full-text indexing Tricia uses the open source library `Lucene`⁵ developed by the Apache Software Foundation. `Lucene` facilitates full-text indexing and search functionality. It is highly scalable and allows creating powerful queries. It was originally written in Java and was ported to many other programming languages.

The main idea behind the `Lucene` architecture is that an index exists which consists of many documents. Each document contains text fields with data. In order for information to be accessed later the field content can be stored. If the field is intended to be used as a component in a search query, it must be indexed. For indexed fields an option to analyze the data in advance exists. Then the tokens of the field's value, instead of the whole string, are indexed.⁶ The index can be searched with queries. Queries may consist of sub-queries and of terms operating on fields. The results can be ordered by their relevance.

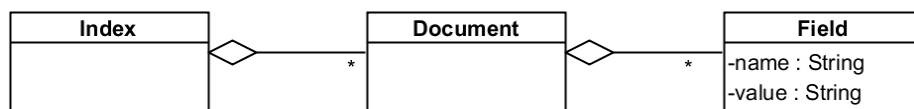


Figure 2.6: Structure of the Lucene index

⁵<http://lucene.apache.org/java/docs/index.html>

⁶http://lucene.apache.org/java/2_4_0/api/org/apache/lucene/document/Field.Index.html

3 Analysis and Design

This chapter describes the results of the analysis and design phases. It contains the detected requirements for an awareness component embedded in Tricia, similar implementations in other enterprise 2.0 systems, and design decisions and drafts for the user interfaces.

3.1 Requirements

The following requirements have been detected for the awareness component in Tricia.

- The **UI presentation of changes** should be improved. Right now, the information view of each change type shows the value before and after the change. A more user-friendly approach would be to point out the difference.
- **Aggregation of changes:** The Tricia user interface allows in-place editing, i. e. changing features of an object without leaving its view mode. However, each in-place edit of a feature causes the creation of a new change entry. To keep the history view and pages listing recent changes more clearly, similar changes could be merged.
- Users can **watch objects**. A **dashboard** allows managing the watch list and displays changes concerning these objects.
- In addition to the dashboard there should be a further view which lists **recent changes of all objects** which can be accessed by the user.
- **Mail notifications** regarding changes of watched objects will be sent periodically - if the user wishes this. The notification service can be configured by each user.

In addition, there are further non-functional requirements:

- **user-friendliness:** The design should be user-centred. The UI interfaces need to be clear and easy to use, and should be uniform. Furthermore, things should be kept simple. To achieve these aims, the development process will attach importance to the user interfaces. Mockups will be worked out in the early design phase.
- **robustness:** The implementation should be robust against incorrect user entries. Furthermore, the access rights need to be taken into account throughout the whole development process in order to avoid thrown program exceptions.
- **stability:** The component must be stable and reliable. Unit tests will help to detect the achievement of this goal.
- **efficiency:** The realisation should be efficient, i. e. use resources economically and allow a short reaction time.

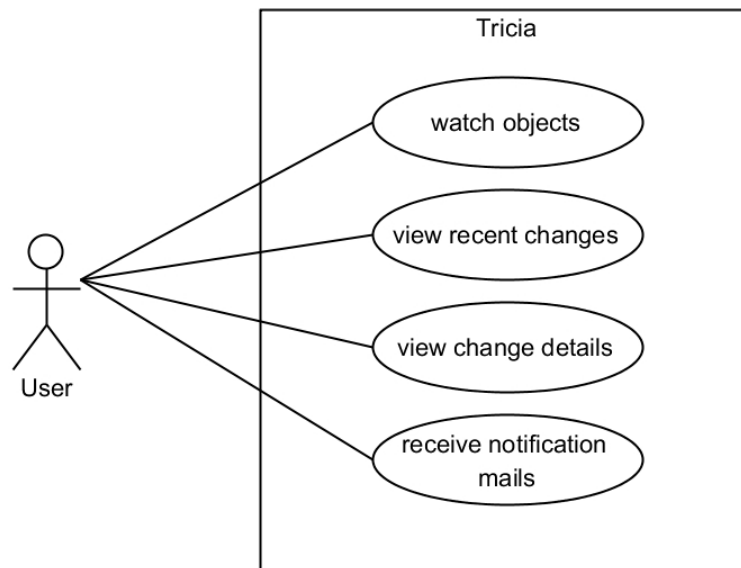


Figure 3.1: Use cases for the awareness component from the user’s point of view

- **extensibility:** It should be possible to add further functionalities in the future without great effort.

3.2 UI Presentation of Changes

For each change type a UI presentation to display details of a change exists. The existing interfaces show the state of the feature before and after the edit. As mentioned before, this could be improved. The new presentations set the focus on the difference between the two states.

The presentations are used in an overlay which pops up, when a user clicks the blue info icon next to a change entry. Right now, the info icons only exist in the version history view of the entities. The new dashboard and recent changes view will use these icons too. Moreover, the presentations are used in the compare handler of entities.

Generally, each overlay consists of a title and the content with the details. The title is “Change:” followed by the name of the changed feature in capital letters.

In the following, for each of the change types, a screenshot of the overlay in the initial system, a planned mockup, and a screenshot of the new interface are shown. It was mostly possible to realise the implementation in a very similar manner to that of the planned mockup. The implementation of the overlays is described as “exemplary” for the type `TagsChange` in chapter 4.1.

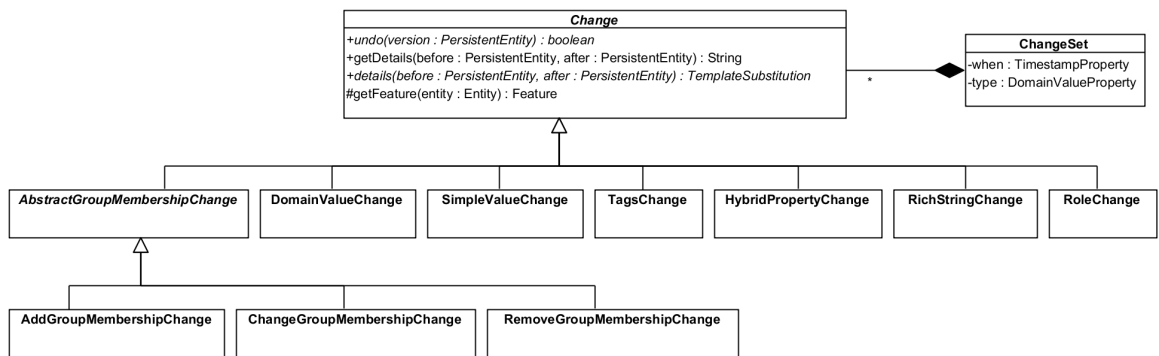


Figure 3.2: Different change types

3.2.1 AbstractGroupMembershipChange

One change type is the `AbstractGroupMembershipChange`. It concerns group memberships and is created if a membership changes. This type is a special case, because it is split up into three different presentation classes. These all inherit the abstract class `AbstractGroupMembershipChange`:

- `AddGroupMembershipChange` is used if a user or a group has been added to a group.
- `ChangeGroupMembershipChange` is used if the membership state (“active” or “applies for”) and / or the comment of a member have changed. Unlike before, only the changed property will be visible, if not both the state and the comment is changed.
- `RemoveGroupMembershipChange` is used if a user or a group has been removed from a group or deleted completely.

The overlays show the type `ChangeGroupMembershipChange`.



Figure 3.3: `ChangeGroupMembershipChange`, before: The state “applies for” has been changed to “active”, the comment has been removed.

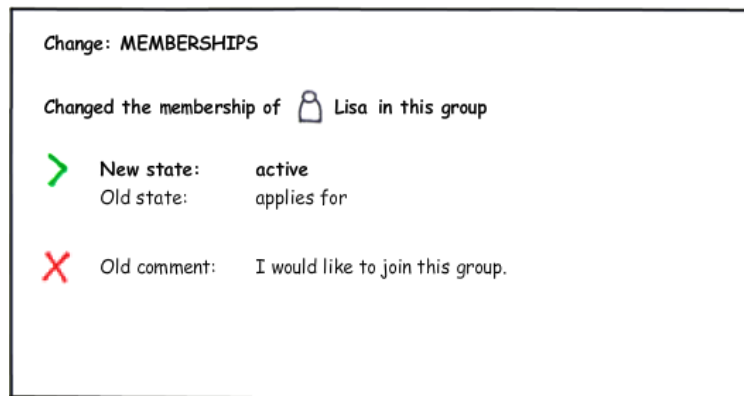


Figure 3.4: ChangeGroupMembershipChange, mockup: same data as in figure 3.3

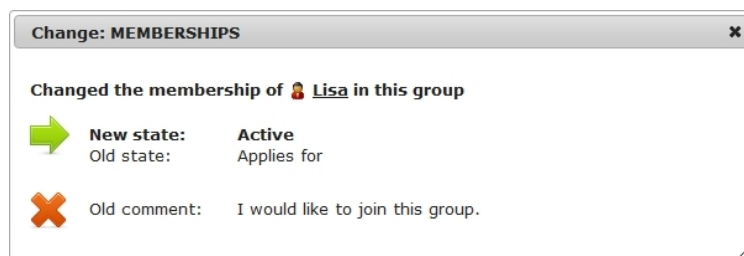


Figure 3.5: ChangeGroupMembershipChange, after: same data as in figure 3.3

3.2.2 DomainValueChange

Another type is the `DomainValueChange`. It is in charge of changed domain value properties. Domain values are attributes which have a specified range of available values. Quite similar to the interface in the initial state, the new and the old value are displayed in the new overlay.

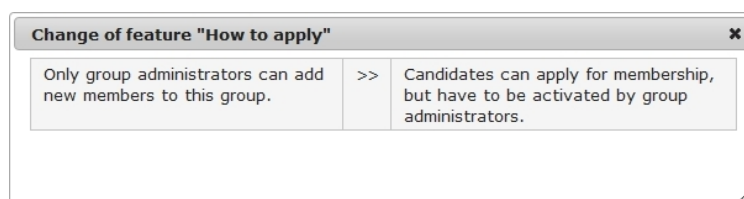


Figure 3.6: DomainValueChange, before: The feature "How to apply" of a group has changed.

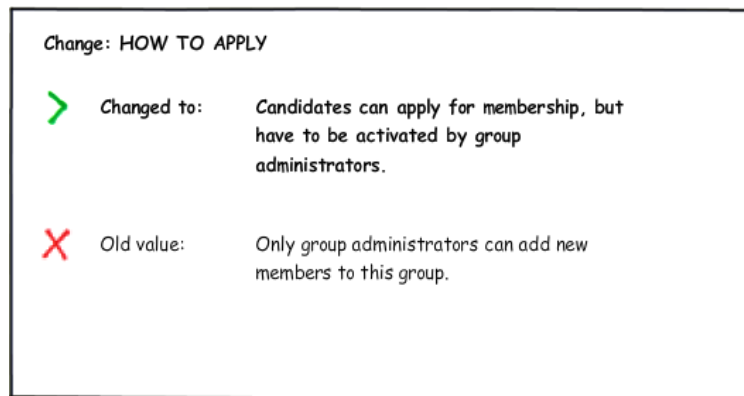


Figure 3.7: DomainValueChange, mockup: same data as in figure 3.6



Figure 3.8: DomainValueChange, after: same data as in figure 3.6

3.2.3 SimpleValueChange

This change type is used, if a simple value has changed. A simple value is either a value of a primitive type or a non-formatted string. DaisyDiff¹, an open source Java library to compare html files, was used to display the change in the initial state. However, as this library is intended to compare continuous text, the result was not very user-friendly for most data types. It was not even convincing for string values, because these are usually very short when used as simple value. (Tricia uses mostly the `RichStringProperty` for longer texts.) The new version shows the new value in bold and the old one below, very similar to the `DomainValueChange` in chapter 3.2.2. If the data type is Boolean, the old value is not shown, because it is simply the negation of the current value.

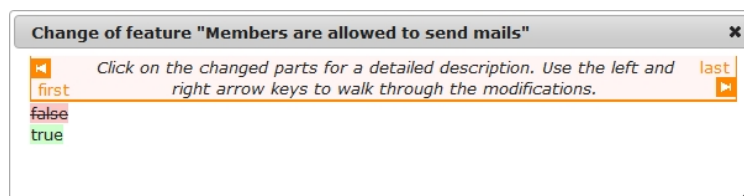


Figure 3.9: SimpleValueChange, before: The feature “Members are allowed to send mails” is a boolean property. Its value has changed to true.

¹<http://code.google.com/p/daisydiff/>

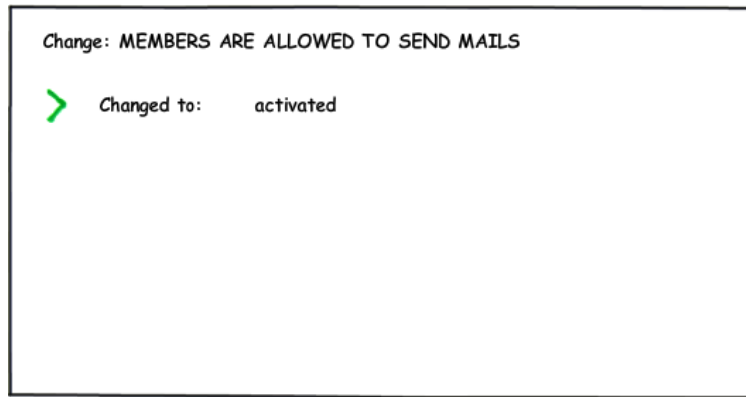


Figure 3.10: SimpleValueChange, mockup: same data as in figure 3.9

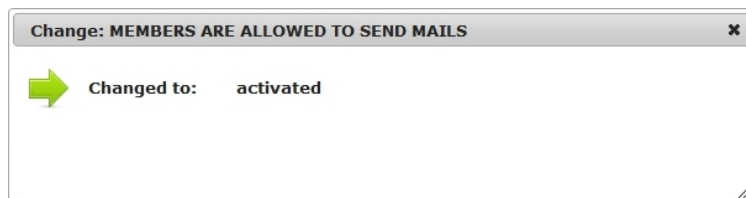


Figure 3.11: SimpleValueChange, after: same data as in figure 3.9

3.2.4 TagsChange

A `TagsChange` is used, when a standard tag or a type tag has been added to or removed from, an entity.

In the initial state, all standard and type tags² of the old and of the new entity version were displayed. This might not be the best solution, if an entity has many tags. The new change presentation contains only the difference by showing all added and all removed tags. Column headings are hidden if no tags correspond to it; e.g. in figure 3.12 there is no heading "Removed type tags", because no type tag has been removed. The headings are in singular or in plural depending on the number of tags listed below.

²A type tag is a tag, which marks wiki pages, which have the same attribute definitions within a wiki.

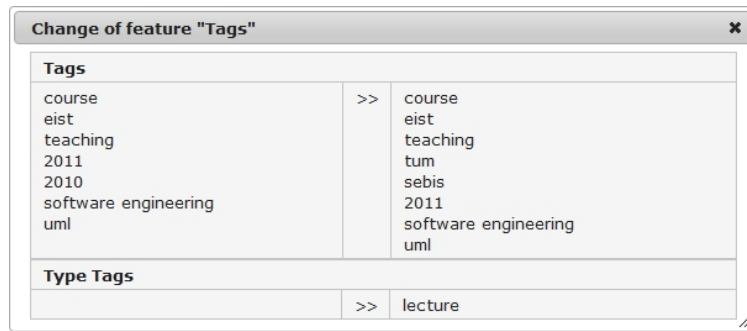


Figure 3.12: TagsChange, before: Two standard tags (“sebis”, “tum”) and a type tag (“lecture”) have been added, a standard tag (“2010”) has been removed.

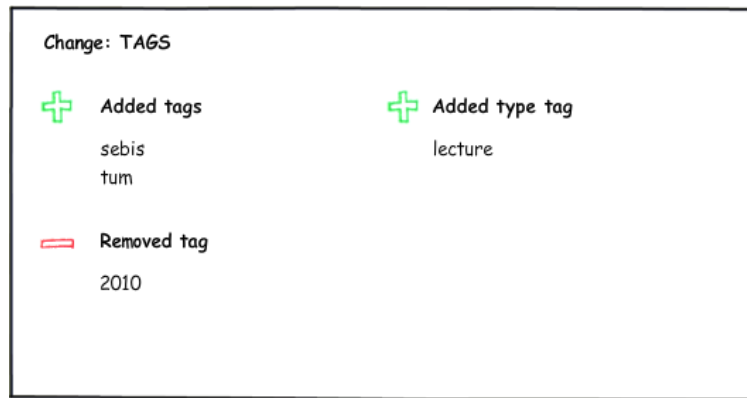


Figure 3.13: TagsChange, mockup: same data as in figure 3.12

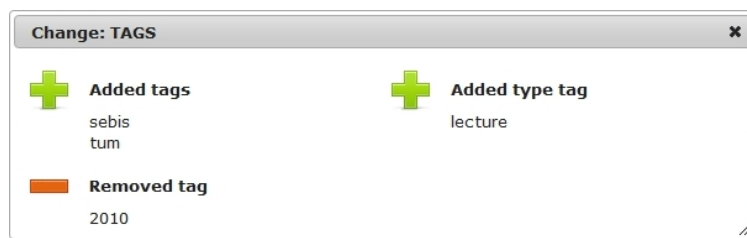


Figure 3.14: TagsChange, after: same data as in figure 3.12

3.2.5 HybridPropertyChange

The class `HybridPropertyChange` is in charge of attribute edits. The attributes are used in the structured content of pages in a hybrid wiki. Each attribute consists of a key and a value. The value can be a single one or a list and might contain links as well. There is a change, if an attribute is added to a page, removed from a page, or if the attribute’s value is edited.

As in most previous versions of the overlays, in this one the values before and after the edit were shown. Now in the new implementation it is checked, whether the old or new value is null. If the old one is null, then the property has been added and is shown with its value. If the new value is null, the property has been removed. Then the text “Removed attribute.” and the old value are shown. If both values are not null, then the ones before and the ones after are displayed.

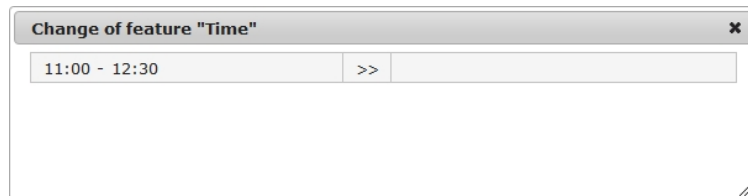


Figure 3.15: HybridPropertiesChange, before: The hybrid attribute “Time” has been removed.

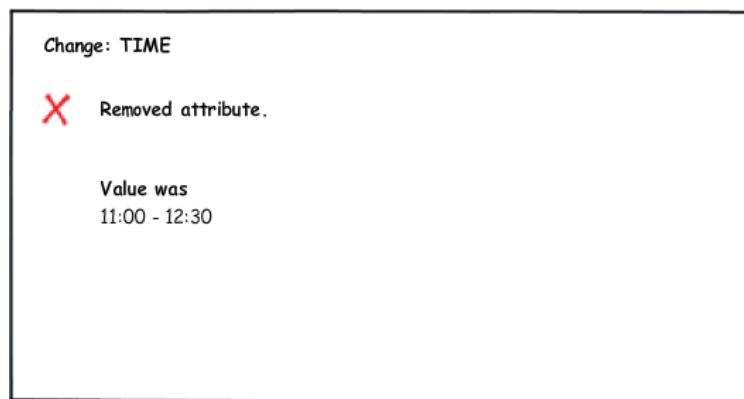


Figure 3.16: HybridPropertiesChange, mockup: same data as in figure 3.15



Figure 3.17: HybridPropertiesChange, after: same data as in figure 3.15

3.2.6 RichStringChange

The `RichStringChange` is used for edits of rich string properties. These properties hold text with formatting. An example for a feature of the type `RichStringProperty` is the content of a wiki page.

In the initial state DaisyDiff was used to display the differences between two versions of a formatted text. DaisyDiff shows the whole text with formatting and marks changed text pieces. It is possible to jump through the modifications via the next link.

If a long text has been edited it might be better to focus on the changed parts, especially if only single words have been altered. Therefore, another diff viewer called *TriciaDiff* was implemented. (DaisyDiff is still available as a second option.) *TriciaDiff* focuses on differences by picking out changed text pieces. These are displayed with their surrounding context. Text changes are:

- pieces (new words, sentences or paragraphs) which were **added**. In the new *TriciaDiff* overlay these are marked by a light green background.
- pieces which were **removed**. These are displayed with a red background and are crossed-out.
- formatting or non-visible information (e.g. the url of a link) which was **changed**. A light blue background indicates this type.
- words, sentences or paragraphs which were **replaced**. The old text is displayed as a removed piece followed by the replacement formatted like added pieces.

TriciaDiff abbreviates long changed text pieces in the centre. This is indicated by the link “[...]”. If clicked, the overlay is reloaded and all previously abbreviated (non-context) parts are expanded. The context is usually abbreviated as well, *TriciaDiff*'s abbreviation algorithm tries to preserve whole sentences (if these are short) by cutting at sentence ends.

For reasons of clarity the original formatting of the text is removed and at most the first five changes are visible, when opening the compare result. The rest of the changes can be displayed by clicking the link “all changes”. Moreover, nearby changes (e.g. two added words in different positions in one sentence) are merged into one change entry.

The use of the AJAX³ concept is being considered for the implementation of this template. When a link is clicked (e.g. the link to expand abbreviated parts), it allows reloading just the content while keeping the overlay open.

³AJAX: Asynchronous JavaScript and XML

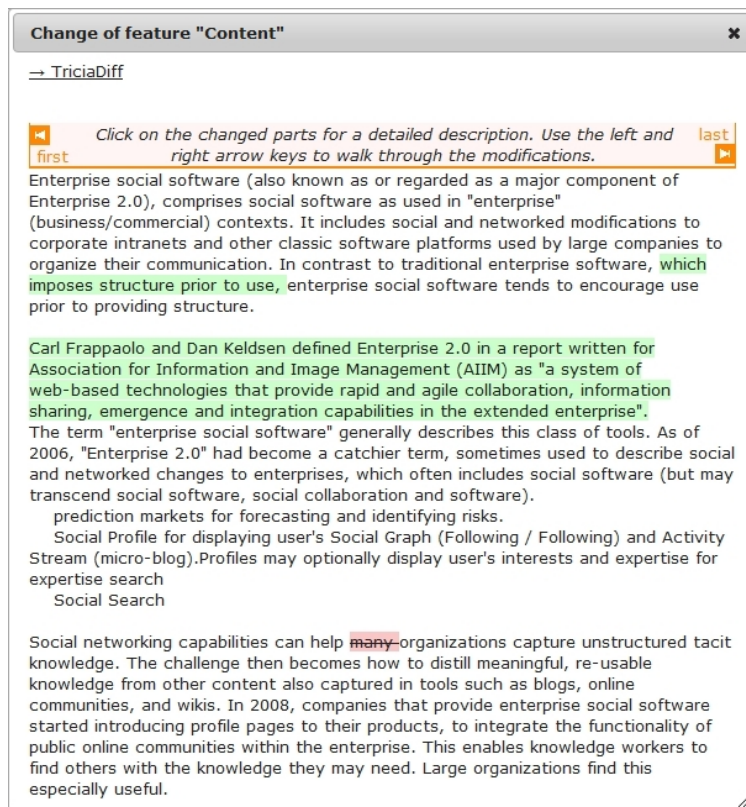


Figure 3.18: RichStringChange, before: A part of a sentence and a paragraph have been added, a single word has been removed.

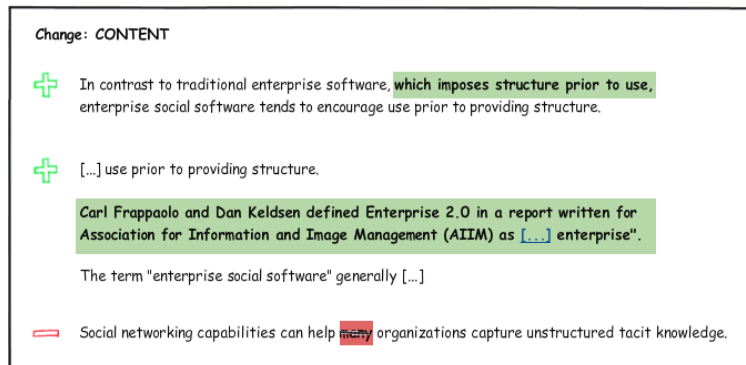


Figure 3.19: RichStringChange, mockup: same data as in figure 3.18

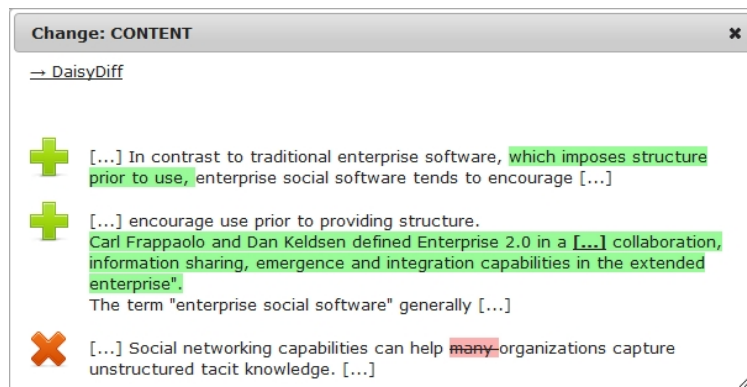


Figure 3.20: RichStringChange, after: same data as in figure 3.18. The second insertion was abbreviated in the centre.

For the implementation details see the chapter 4.2 (HTML comparison with TriciaDiff).

3.2.7 RoleChange

The last viewed type is the `RoleChange`. An instance of it is created, when one of the following use cases occurs:

- A user or a group is added as an editor or as a reader to an entity (e.g. a wiki page).
- A user or a group is removed as an editor or as a reader from an entity.
- A wiki page is moved to another wiki (its space changes).
- A wiki's main page is exchanged with another page.

In the initial state all editors/readers before the change (in the left column of the table) and after it (in the right column) was shown. In the new implementation only added and removed editors/readers are mentioned. Nevertheless, all editors/readers can be found in the details view of the entity.

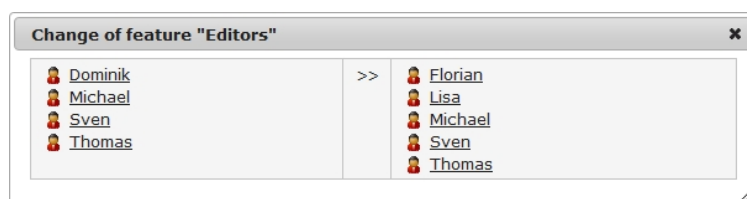


Figure 3.21: RoleChange, before: Two users have been added as editors, one user has been removed.

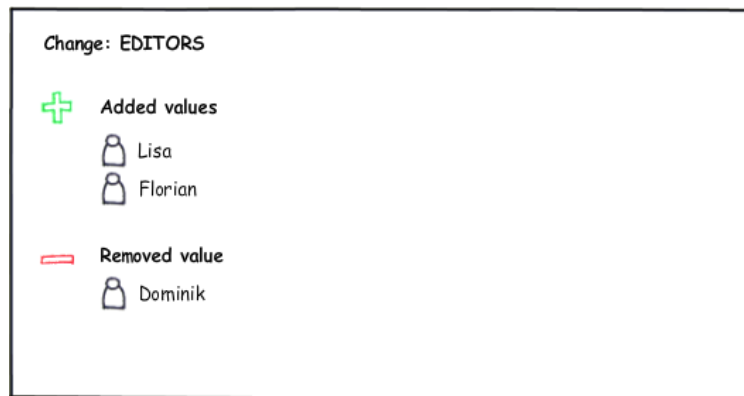


Figure 3.22: RoleChange, mockup: same data as in figure 3.21



Figure 3.23: RoleChange, after: same data as in figure 3.21

3.3 Aggregation of Changes

Each persistent entity has a version history which enables comparisons with earlier states. The history contains change sets. As already mentioned, every in-place edit leads to the creation of a new change set. On this account many little change sets containing just one change exist. An aggregation of similar sets would keep the history and pages listing changes more clearly.

Two change sets can be considered to be similar if both

- were carried out by the same user
- are less than ten minutes between (the time span can be set in a constant)
- concern the same entity
- are of the type "edit".

View Details Attachments Versions

Versions of [Advanced seminar](#) ⚠

(0 - 13 of 13).

Who	Properties	When	
Max Mustermann	Tags i	1 minute ago	
Max Mustermann	Tags i	1 minute ago	Compare with current version
Max Mustermann	Content i	1 minute ago	Compare with current version
Max Mustermann	Organizer i	2 minutes ago	Compare with current version
Max Mustermann	Language i	2 minutes ago	Compare with current version
Max Mustermann	Module No. i	2 minutes ago	Compare with current version
Max Mustermann	Lecturer i	2 minutes ago	Compare with current version
Max Mustermann	Readers i	2 minutes ago	Compare with current version
Max Mustermann	Tags i	3 minutes ago	Compare with current version
Max Mustermann	Content i	5 minutes ago	Compare with current version
Max Mustermann	Content i	8 minutes ago	Compare with current version
Max Mustermann	Tags i	9 minutes ago	Compare with current version
Max Mustermann		9 minutes ago	Compare with current version

1 (total result pages: 1)

Figure 3.24: An entities' version history with a lot of single changes

3.4 Dashboard

In general, a dashboard is a form of presentation. It displays the most important information for the user on a single-screen page and allows monitoring of what is going on. The information on a dashboard is updated regularly. [Tid11]

3.4.1 Realisations in other Systems



Figure 3.25: Dashboard of BusinessWiki⁴

In most other systems the information is usually presented as a combination of text and graphics. Related data is often grouped in titled sections or panels which can be moved, collapsed, or enlarged. The dashboard must fit on a single screen, so that it all can be seen at a glance. [Few06]

Figure 3.25 shows the dashboard of the enterprise 2.0 system BusinessWiki⁵ as an example. It contains configurable panels for ones own watch list, the latest changes, ones own edits, the calendar, the rss feeds...

3.4.2 Planned Implementation in Tricia

The dashboard of Tricia will focus initially only on the awareness functionality. The planned user interface consists of two areas:

- a box with watched objects
- a box containing changes belonging to the objects

In addition, there will be a link to the settings page which allows some configurations.

⁴<http://onbusinesswiki.de/screen/bwi1.png> Visited on July 18th 2011.

⁵<http://onbusinesswiki.de/>

Welcome, Michael!

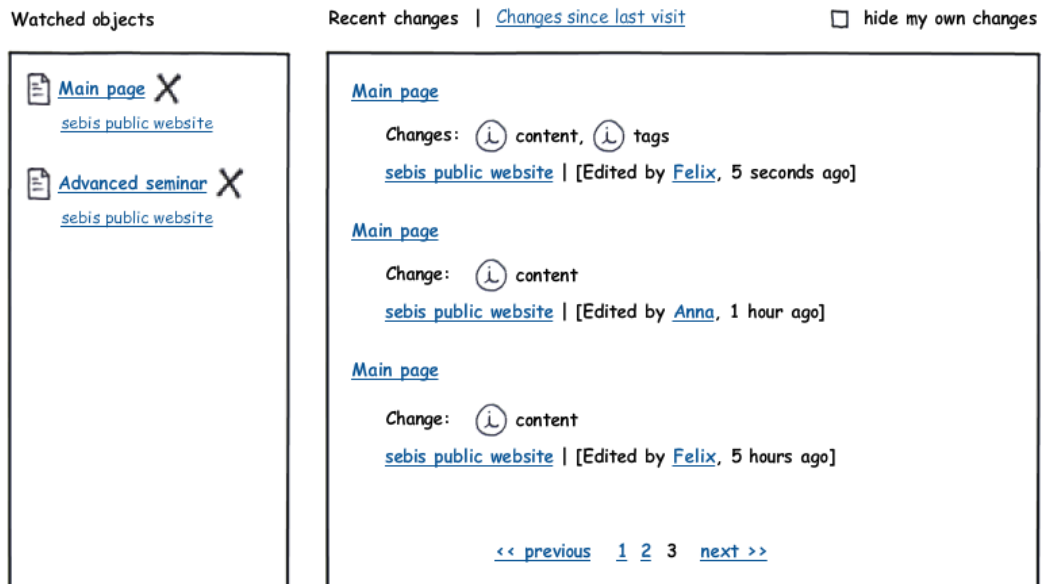
[Settings](#)


Figure 3.26: Mockup of the planned dashboard

Watched Objects

This box on the left contains a list of all objects⁶ watched. For each object the name is shown as a link, so it can be opened by clicking it. The link contains an icon to indicate the object type. Next to the link, there is a button (represented by a cross icon) to stop watching the object. (The object is then removed from the list.) If the object is in a space, the name of it is shown below. Confirmation messages will affirm the impact of user actions.

Changes

In the centre there is the main content of the dashboard.

There are planned two different views of changes: one called “Recent changes” (of watched objects) and the other one “Changes since last visit”. Both views contain a number of change entries. In order to keep the Tricia interface uniform, the appearance of the entries is based on the presentation of results in the search view. The entries are sorted by the time stamp of the change.

At the bottom of both views there is a navigation control to show further results. It contains a “next” link and some page numbers. There is also a “previous” link to move one page back - of course, it is not visible when the first results are displayed. The number of available pages is not visible, unlike in the search results, as it is not relevant here.

⁶The word “object” is used as an acronym for an instance of any content type. Content types are: wikis, wiki pages, blogs, blog entries, directories, documents, comments...

View “Recent changes” The view contains a list of recently stored change sets which concern the user’s watched objects. This view is visible by default when the dashboard is opened.

Each change set is presented through an information block: In the first line there is the name of the object as a link. Below there is the list of the edited features. Like in the version view of an entity, next to each feature there is the blue info icon. By clicking it, an overlay is opened which shows further information on the change. Moreover, each entry contains the editor’s name and a time specification. The presentation of the time depends on how much time passed since the edit has been carried out. If the change was made less than an hour ago, the number of passed minutes is shown; if less than 24 hours passed, the number of hours is visible. Otherwise “yesterday” or the date is displayed.

In this view, the option “hide own changes” is available. It allows hiding own edits in the list, so that the user can concentrate on what others did.

View “Changes since last visit” The second view, “Changes since last visit”, can be shown by clicking the equally named link on the top.

This view lists all watched objects, which were changed since the user’s last visit to them. Each object altered appears at most once in the list, regardless of the number of changes. Each entry contains the object’s name and the features involved. As in the other view, an overlay with further details can be shown for each changed feature. In addition, the user can see how often the object was changed since his last visit and who the last editor was.

The purpose of this view is to show the difference between the current and the user’s last known state of an object. It is less important here to get to know, when and by whom it was edited. The option to hide own changes is not appropriate in this view, therefore, it is not available.

3.5 Recent Changes

While the dashboard is only about objects watched, a second page exists whose aim it is to show recent changes of all objects, which can be accessed by the user.

3.5.1 Realisation in other Systems

Most enterprise 2.0 systems have functionality to list changes made a short time ago. Three examples were picked out and are described below.

Wikipedia

The encyclopaedia project Wikipedia⁷ uses MediaWiki as the underlying wiki software. In Wikipedia each page is represented by a source code. It contains the text, tables, outgoing links, and categories. Therefore, each change concerns the code.

To view the activity of other users the page “Recent changes” exists. It is often used by administrators to detect vandalism.

⁷<http://en.wikipedia.org/>

Changes are described by a line of text and contain the following information:

- name of the wiki page (as a link)
- time when the change happened
- accumulated number of added and removed characters to indicate the change size
- editor (with a link to his profile)
- short summary of the change (title of the changed paragraph and/or a comment by the user) (optional)

In addition, there are two links: “diff” links a page, which displays the difference (before-after-comparison), “hist” links the version history of the wiki page.

Some filtering options, such as hiding minor edits, hiding changes of anonymous or logged-in users, selecting namespaces... are offered.

18 July 2011

- (diff | hist) . . Foreign policy of Ollanta Humala; 10:54 . . (+259) . . Lihaas (talk | contribs) (→Americas tour:)
- (diff | hist) . . Wikipedia:Village pump (proposals); 10:54 . . (+759) . . Nil Einne (talk | contribs) (→Require Captcha for Special:EmailUser:)
- (diff | hist) . . Lufthansa Cargo Flight 8460; 10:54 . . (+75) . . 79.77.223.18 (talk)
- (diff | hist) . . Sam Perrett; 10:54 . . (0) . . 110.175.223.179 (talk)
- (diff | hist) . . Santa Catarina Ixtahuacan; 10:54 . . (+18) . . Nstannik (talk | contribs)
- (diff | hist) . . Media of Iran; 10:54 . . (+58) . . 109.162.198.30 (talk) (→External links: +video)
- (diff | hist) . . Cricket in Western Australia; 10:54 . . (+1,068) . . IgnorantArmies (talk | contribs) (→ Venues)
- (diff | hist) . . N Edge routing; 10:54 . . (+329) . . Rich Farmbrough (talk | contribs) (→Created page with “Edge routing” may refer to: * Routing a moulding on the edge of a piece of timber or other material. * Network routing at the edge of a network (th...))
- (diff | hist) . . 1. FC Bamberg; 10:54 . . (+10,287) . . Calistemon (talk | contribs) (convert to article)
- (diff | hist) . . Monsters in the Closet; 10:54 . . (+4) . . 79.177.26.42 (talk)
- (diff | hist) . . Philippine cuisine; 10:54 . . (-26) . . 71.237.197.56 (talk) (-restructure tag)
- (diff | hist) . . User:Taelus/Sandbox; 10:54 . . (-623) . . Taelus (talk | contribs) (→National Research University “Moscow Aviation Institute”: remove, done)

Figure 3.27: Recent changes view of the English Wikipedia⁸

Ubuntu Wiki

The wiki of the web portal ubuntuusers.de⁹ contains tutorials on and trouble-shooting for the operating system Ubuntu. The wiki is running on a python application called Inyoka.

⁸<http://en.wikipedia.org/wiki/Special:RecentChanges> Visited on July 18th 2011.

⁹<http://wiki.ubuntuusers.de/>

Letzte Änderungen

2011-07-18		
20:04	AqBanking (1x)	<ul style="list-style-type: none"> • aqbanking wizard nur bis Maverick (von kaputtnik)
19:53	ubuntuusers/intern/Wiki-Team/Arbeitsablauf (1x)	<ul style="list-style-type: none"> • Flaggen, Smilies und InterWikiLinks (von cornix)
19:41	Baustelle/Spiele/Dark Horizons: Lore (1x)	<ul style="list-style-type: none"> • natty (von march)
18:56 - 19:04	Baustelle/KMyMoney (2x)	<ul style="list-style-type: none"> • Bildtest (von kaputtnik) • test (von kaputtnik)
19:04	Baustelle/KMyMoney/kmymoney_logo.png (1x)	<ul style="list-style-type: none"> • von kaputtnik
18:59	KMyMoney/Trash/kmymoney-icon (2x)	<ul style="list-style-type: none"> • Umbenannt von KMyMoney/kmymoney-icon (von kaputtnik) • Ohne Endung, wird nicht verwendet (von kaputtnik)
11:21 - 18:58	Baustelle/OpenERP (3x)	<ul style="list-style-type: none"> • Schritte eingefügt. (von orgel) • Fertigstellungsdatum geändert. (von orgel) • beschreibung (von Philipp B)
18:53	Baustelle/KMyMoney/kmymoney.png (1x)	<ul style="list-style-type: none"> • von kaputtnik
18:52	Anwendertreffen/Stuttgart (1x)	<ul style="list-style-type: none"> • changed (von frankhe)
17:38 - 17:42	Soundkarten konfigurieren/HDA (2x)	<ul style="list-style-type: none"> • Zuletzt geändert durch lucid-dream, Eintrag für Lenovo G555 (von lucid-dream) • Eintrag Lenovo G555 Codec geändert. (von lucid-dream)
15:09 - 17:02	Hardwaredatenbank/Eingabegeräte (2x)	<ul style="list-style-type: none"> • Logitech G400 Desktop-Maus ergänzt (von wireded) • Maus Roccat pyra aktualisiert (von Crossfader)

Figure 3.28: Recent changes view of the wiki of ubuntuusers.de¹⁰

As pictured in figure 3.28, changes are displayed in a three-column table with the following details:

- time (or time span) when the change was made
- path of the concerned wiki page composed by its namespace and name. Next to it, there is a number in brackets which indicates the number of single edits.
- editor's comment and his name (for each single edit)

A special characteristic is that edits within a specified time span are grouped.

Wikispaces

Wikispaces¹¹ hosts millions of wikis for private users, companies and institutions. Recent edits are contained in a table view. Compared with the other two views, it has the best filter functionality: it allows selecting changes by the type of the object (page, message, comment, file, tag, member), to specify a date range, and to filter by user name.

Next to the name of the page changed there is either its content (if it is not extensive) or the link "(view changes)".

¹⁰http://wiki.ubuntuusers.de/Wiki/Letzte_%C3%84nderungen Visited on July 18th 2011.

¹¹<http://www.wikispaces.com/>

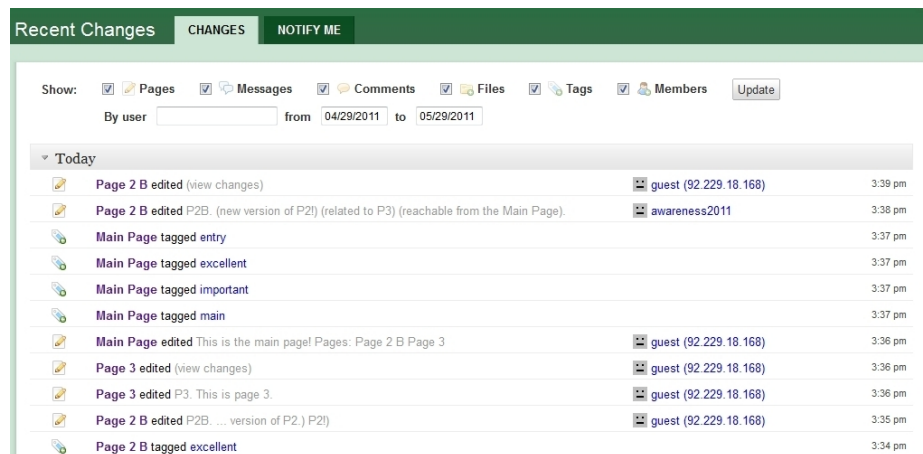


Figure 3.29: Recent changes view of Wikispaces' wikis¹²

3.5.2 Planned Implementation in Tricia

After having considered the interfaces of other systems, the following interface has been developed. Its layout is intended to be similar to the search result page. On the left there is a sidebar with filter options to restrict the result set. In the centre further restrictions can be selected (at the top), below there are the results.

Sidebar

The sidebar itself is composed of fold-away boxes. Each box can be open or collapsed, so that only its heading is visible.

Space Some object types (wiki pages, blog entries) are held in a space. It is possible to select a space by clicking its name. Then it is typed in bold and only objects of it appear in the results. Like in the Tricia search at the most one space can be chosen. The selection can be dropped by a second click on it. If no space is chosen, the filter is not applied.

Date It is possible to restrict entries based on the time stamp of the edit. A time interval can be determined by filling the field for the start date ("after") and/or the field for the end date ("before"). Each interval border is only active, if the corresponding check box on the left is checked. This filter is disabled, if both check boxes remain unchecked.

Change Type A filter to view only add events, or only edits, or only delete events can be used. The default selection of this combo box is "all change types".

Other Like in the dashboard it is possible to filter out own changes. By default the box is unchecked. A change of the check state takes effect immediately.

¹²<http://enterprisetwozero.wikispaces.com/space/changes> Visited on May 29th 2011.

Recent changes



Figure 3.30: Mockup of the planned recent changes page

Centre

In the centre there are further filters and the results:

Further Filters, Sorting In addition to the already mentioned filters, results can be limited by the object's name. The name can be entered in a text box, which shows suggestions while typing. Right next to that there is a box to specify the sort order; the available alternatives order by change date or by name.

Tags are another way to constrain the results. The tags filter control is hidden by default; it can be uncovered by clicking the button "Tag Filter". It is composed of two boxes, a green and a red one. Tags placed in the green box are required, out of it only entries having all of the required tags are included in the results. The red box holds the tags which must not appear in the result set. Below there is a tag cloud with frequently used tags which is even visible if the tag control is hidden. The control supports typing in a box and drag and drop.

Results Below the tag cloud there are the change events, which are sorted as selected above. Ten results are visible per page.

The presentation is very similar to the dashboard's "recent changes" view. By contrast, next to the name of each object (except deleted ones) there is a "watch" link by which it

can easily be added to the personal watch list - or, if it is already in there, removed from it (In this case the link is labelled "stop watching").

3.6 Mail Notification

Another message channel for awareness purposes, beside the dashboard and the recent changes page, is the mail notification.

3.6.1 Realisations in other Systems

Most enterprise 2.0 systems have the functionality to monitor pages. A change of a monitored page usually leads to a notification mail. However, nearly all systems send the mail immediately after the change. This results in many mails sent, which then flood inboxes. Because of that, users might watch less objects or don't use this feature.

Ubuntu Wiki

The previously viewed Ubuntu wiki offers such a notification service. The sent mails contain for each page altered its name and the editor's name. In addition, there are links to get to the version comparison page and a short summary text describing the change (if the editor specified one). At the bottom of the mail there is a link to unsubscribe from the notification service.



Figure 3.31: Mail notification sent by the Ubuntu wiki

3.6.2 Planned Implementation in Tricia

The implementation in Tricia will follow another approach: Instead of sending a mail after each change, a mail is sent in regular intervals (e.g. once a day) containing a summary of the latest changes of watched objects.

The challenge is to inform the user about what happened without overburdening him with information. Therefore, the mail should have a reasonable length. Of course, the subject of the mail should already make clear what the mail is about. Moreover, there are further restrictions, e.g. it does not make sense to use images, because most mail clients don't allow embedding external images by default for security reasons.

Some drafts for notification mails were worked out. These differ slightly in the level of details. One draft looks as follows: There is a short block for each changed object. The

block contains the object's name as a link, so that the user can get there with one single click. Next to it, there is another link to get directly to the compare handler. In addition, for each object the time stamp of the last edit and the last editor are given as information. A short preview of the changed features is included in the mail. At the end of the mail there it is specified how to unsubscribe from the notifications.

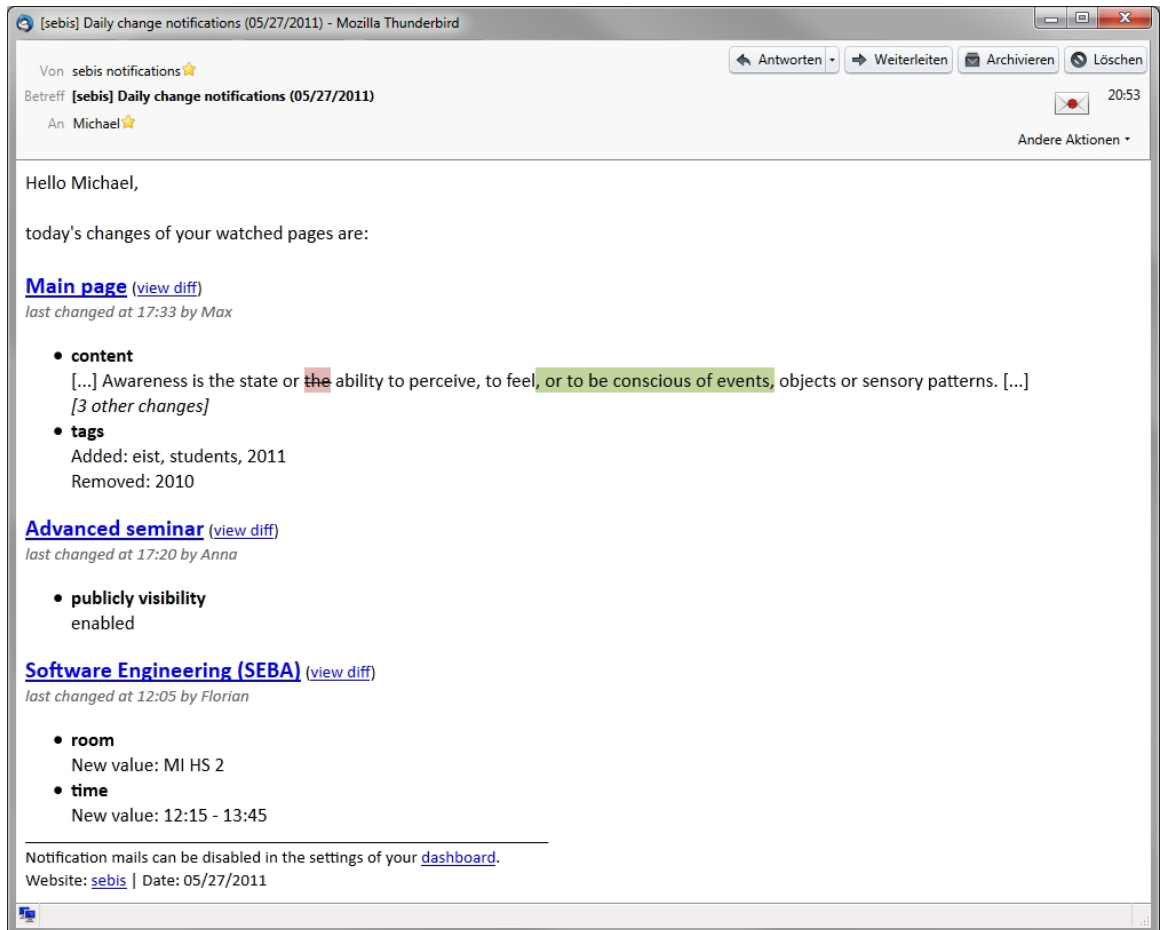


Figure 3.32: Planned look of the mail notification

4 Implementation

This chapter contains the implementation details.

4.1 Presentation of Changes

Firstly, the implementation of the UI presentation of changes is explained exemplary for the type `TagsChange`. The implementation of each presentation includes the development of the html template and of the corresponding handler, which dynamically substitutes placeholders with data.

4.1.1 Template

As viewed in chapter 3.2.4, the tags change's presentation informs about added and removed tags (both for standard and type tags). The following is a simplified excerpt of its template code:

```
[...]
${hasAddedStdTags$
  <tr>
    <td>
      
    </td>
    <td>
      <b>${TEXT_AddedStdTags$}</b>
    </td>
  </tr>
  <tr>
    <td/>
    <td>
      <ul>
        ${addedStdTags t$
          <li>${t.value$}</li>
        }
      </ul>
    </td>
  </tr>
${hasAddedStdTags}$
[...]
```

Listing 4.1: `TagsChange`: excerpt of the template code

The excerpt shows the part which is responsible for the added standard tags. `${hasAddedStdTags$}` is a conditional template which surrounds other template code. The surrounded code is being suppressed, if no standard tags are added. Otherwise it defines two table rows: The upper row incorporates an image with a plus icon and shows the heading. The heading is provided by a print substitution (`${TEXT_AddedStdTags$}`),

so that it can be set dynamically depending on the number of tags listed below (singular/-plural). The second row contains a list substitution for the names of the added standard tags.

The subsequent code of the removed standard tags and of the type tags looks very similar.

4.1.2 Substitution

The class `TagsChange` is in charge of the substitutions. It has two variables of the type `Set<String>` holding the standard tags and the type tags before the change. When the user wants to see the details of the change, the method `details()` of this class is invoked to calculate the content of the overlay. This method returns a template substitution:

```
@Override
public TemplateSubstitution details(PersistentEntity before, PersistentEntity after)
{
    return new TemplateSubstitution()
    {
        @Override
        public void init() { /* ... */ }

        @Override
        public void putSubstitutions(Template template) { /* ... */ }
    };
}
```

Listing 4.2: `TagsChange`: method `details()`

In the overridden `init()` method of the inner substitution class the tags of the new state are retrieved (lines 4 and 5) and then the before-after-comparison is started:

```
@Override
public void init()
{
    Set<String> tagsAfter = Sets.newHashSet(after.adapt(Taggable.class).
        getStandardTagNamesWithoutTypeTags());
    Set<String> typeTagsAfter = Sets.newHashSet(after.adapt(Taggable.class).
        getTypeTagNames());

    diff = new TagsDiff(tagsBefore, tagsAfter, typeTagsBefore, typeTagsAfter);
}
```

Listing 4.3: `TagsChange`: template substitution's `init()` method

The inner class `TagsDiff` calculates the difference: first, four hash sets are instantiated. All standard tags from the state after are added to the set `addedStandardTags` and then all tags from the state before are removed from it. This is done using the native methods of `HashSet`, which are performed highly efficiently. As a result, in `addedStandardTags` remain only the added standard tags. This is also done for the removed standard tags and for the type tags. At the end, the sets are converted to lists and sorted.

After that, the template's placeholders are replaced with the data worked out. This happens in the `putSubstitutions()` method of the template class. `SimpleListSubstitution` is a self written class, which encapsulates parts of the iteration logic in order to improve the reusability. The print substitutions for the headings are extracted to the method `putTextSubstitutions()` in order to keep this method more convenient.

```

@Override
public void putSubstitutions(Template template)
{
    putTextSubstitutions(template);

    template.put("addedStdTags", new SimpleListSubstitution()
    {
        @Override
        protected Iterable getValues()
        {
            return diff.getAddedStandardTags();
        }
    });

    //lots of further substitutions ...
}

```

Listing 4.4: TagsChange: excerpt of the method putSubstitutions()

4.2 HTML comparison with TriciaDiff

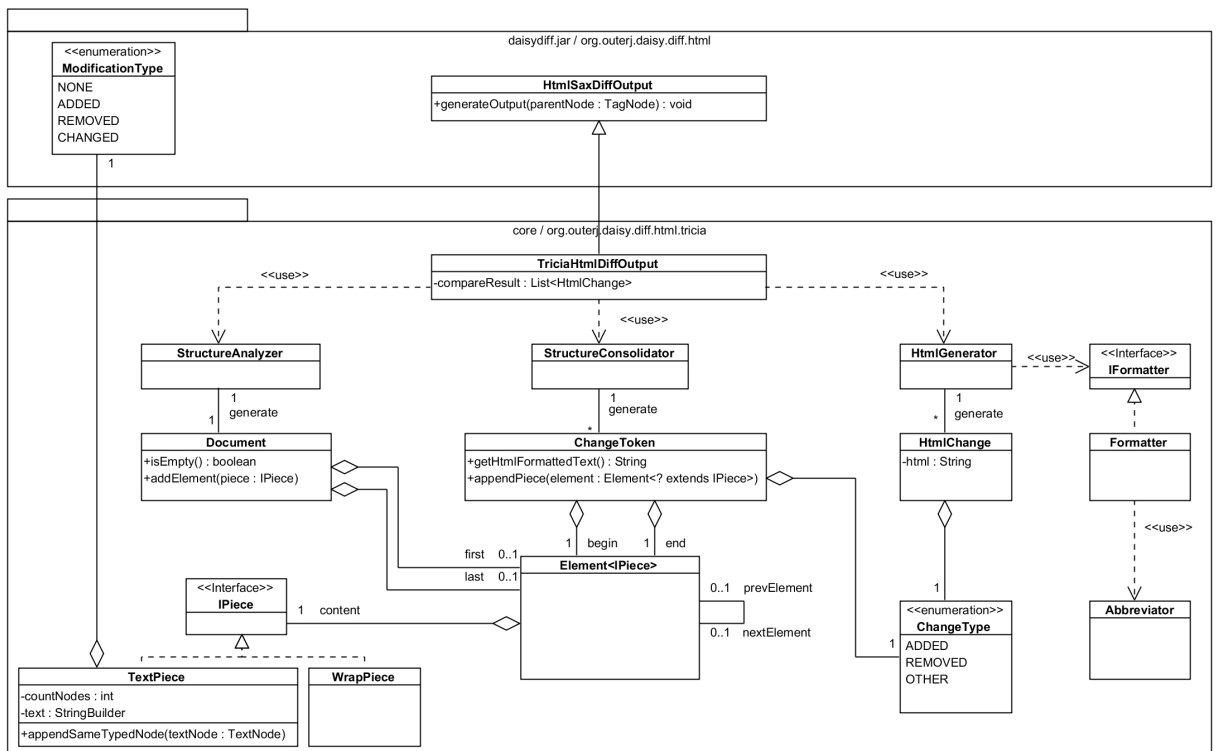


Figure 4.1: Overview of TriciaDiff

This section describes the implementation of the developed component *TriciaDiff*. Its purpose is to compare two html strings. The comparison result is used in the overlay of the `RichStringChange`¹ (see chapter 3.2.6). *TriciaDiff* builds on the open source project

¹The `RichStringChange` concerns changes of properties of the type rich string. In *Tricia*, rich strings are

DaisyDiff².

The implementation is described by viewing the three steps of the calculation. The calculation is started in the method `generateOutput()` of the class `TriciaHtmlDiffOutput`.

- Step 1: The `StructureAnalyzer` converts the input, which was partially preprocessed by `DaisyDiff`, to an internal structure.
- Step 2: The `StructureConsolidator` merges changes, which are in close proximity.
- Step 3: The `HtmlGenerator` computes html results which can be displayed in the overlay. This step includes the abbreviation of the results.

Each process step is first described generally and then for the following example:

- html content before: "This is the first sentence. There is another sentence. The first one was not changed.
"
- html content after: "This is the first sentence. There is another sentence. The first one was changed short time ago.
"

The formatting of the words "This is" has been changed to bold, the word "not" has been removed, and "short time ago" has been inserted.

4.2.1 StructureAnalyzer

At the beginning, the `StructureAnalyzer` restructures the input data for the following processing steps. It flattens the tree structure of the input and already merges text pieces of the same change type directly following one another.

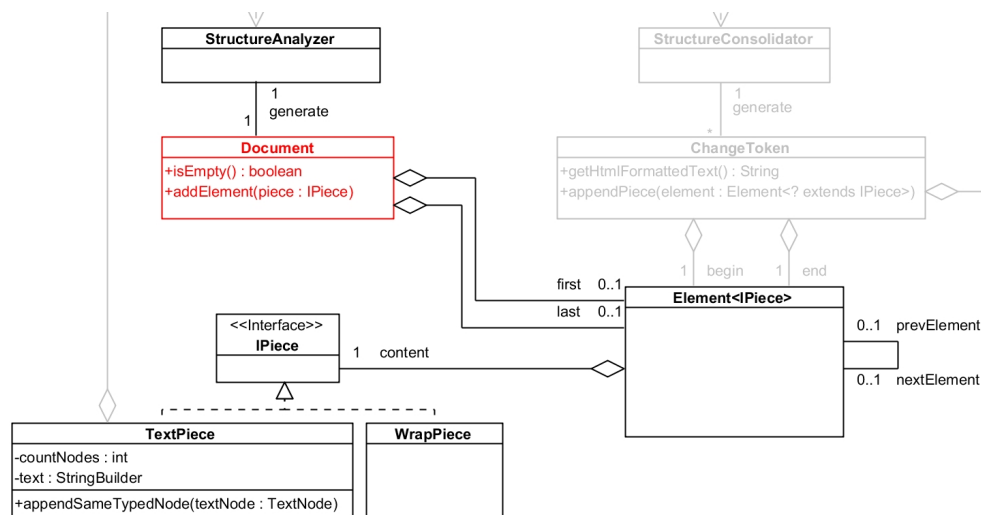


Figure 4.2: TriciaDiff: StructureAnalyzer

¹internal represented as html.

²<http://code.google.com/p/daisydiff/>

Input

The given input for this process step is a `TagNode` representing the root of a tree. Each child of the root can either be a `TagNode` (and contains children again) or a `TextNode`. A text node contains a token, which is a word, a whitespace, or a punctuation mark. In addition, each `TextNode` has a modification type, which is represented as an enumeration class. The enumerators are:

- **ADDED:** The token was added, i. e. the token is part of the second input, but is missing in the first one.
- **REMOVED:** The token was removed; it is no more in the second input.
- **CHANGED:** The formatting of the token or non-visible html tag value (e.g. the url of a hyper reference) was changed.
- **NONE:** The token is equal in both input values.

The class diagram in figure 4.3 illustrates the input. The classes form a *composite pattern* with the abstract super class `Node` as component, the class `TagNode` as composite, and the `TextNode` as leaf. [GHVJ09]

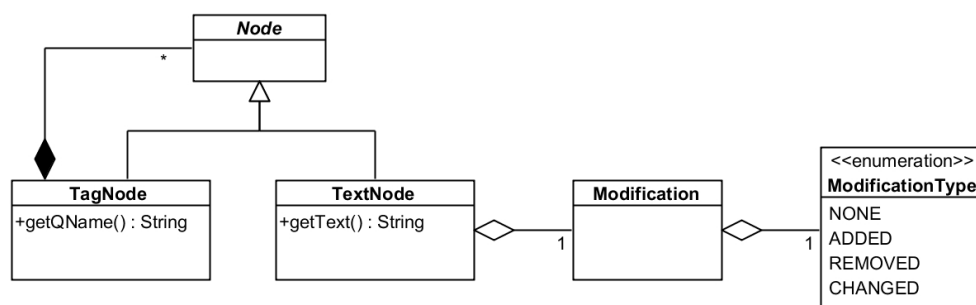


Figure 4.3: Input of the StructureAnalyzer

Output

The result is an instance of `Document`, whereas this class can be seen as a *linked list* containing items. An item is of the generic class `Element` and has a reference to the previous and to the next item. The content of the item is either a `TextPiece` (containing text) or a `WrapPiece` (indicating that a paragraph ended).

Calculation

The analyzer starts by iterating over the children of the root node.

- If a child is of the type `TagNode`, then its tag name is retrieved and compared with a collection of known html structure tags³. Due to performance reasons, the known tags are stored in a static hash set. If the name is contained in the set, a wrap piece is

³e.g. br, p, h1, h2...

added to the document. In case the current method is then recursively invoked for all children of this `TagNode`.

- If a child is a `TextNode`, then the method `addText()` is invoked to handle it. This method uses the private class variable `openTextPiece` (type: `TextPiece`), which holds the latest processed text of preceding nodes. If the change type of the child is the same as the one of `openTextPiece`, then its text is added to it. If the child has another type, the open piece is committed and a new one of the type of the child is created. After having processed all nodes, the last open piece is committed too. Then the document is returned as the result.

```
private void addText(TextNode node)
{
    if (openTextPiece != null && openTextPiece.getChangeType() == node.getModification()
        .getType())
    {
        openTextPiece.appendSameTypedNode(node);
    }
    else
    {
        closeCurrentTextNode();
        openTextPiece = new TextPiece(node);
    }
}
```

Listing 4.5: StructureAnalyzer: method `addText()` to append text of the same type

Example

The document generated for the example input (see the beginning of chapter 4.2) contains seven elements. The first six elements have a text piece as content, the last one has a wrap piece, because the sentence ended with `
`. Text piece `t1` shows the merge result of the directly consecutive text nodes “This”, “ ” and “is”, which were all changed in the formatting. The variable `countNodes` reveals how many nodes were combined.

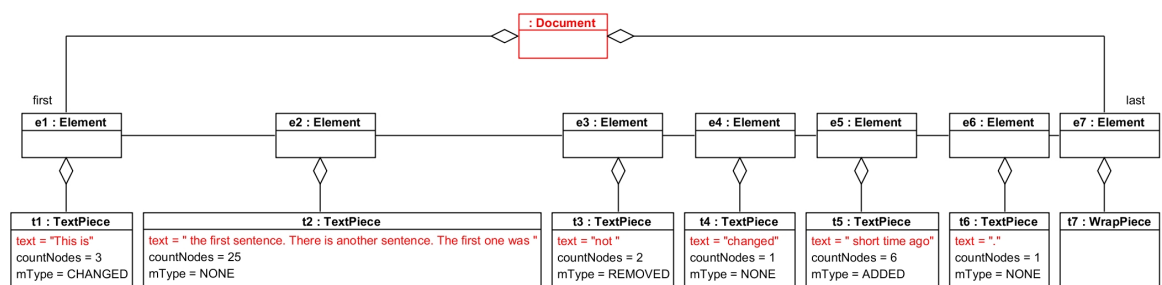


Figure 4.4: Objects of the StructureAnalyzer calculation result for the example

4.2.2 StructureConsolidator

After having restructured the input, the next step is carried out: The `StructureConsolidator` merges changes which are in close proximity.

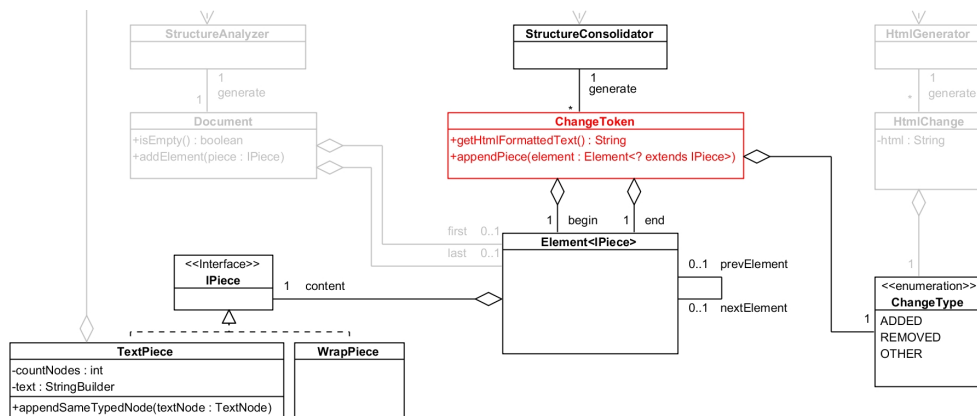


Figure 4.5: TriciaDiff: StructureConsolidator

Input

The instance of `Document` (the calculation result of the `StructureAnalyzer`) is the input of the `StructureConsolidator`.

Output

The output of this pass is a list of `ChangeTokens`. For each (merged) change it contains one or more list elements of the document. Each `ChangeToken` has a `ChangeType`: It is `ADDED` if all elements have the modification type `ADDED`, it is `REMOVED` if all elements have the modification type `REMOVED` and it is `OTHER`, if there is an element with the modification type `CHANGED` or if not all elements have the same type.

Calculation

The calculation starts by iterating over the (list) elements of the document. The change piece of each element is retrieved and appended to the class variable `currentChange` (type `ChangeToken`) (which holds the current interim result). If the previously added changes in `currentChange` have another type than the viewed piece, these are committed before and a new empty change token is prepared. The execution of this logic slightly varies depending on the type of the element content:

TextPiece If the content is an instance of `TextPiece`, the method `handleText()` is called: The method adds the piece to `currentChange`, if it concerns an edit (`ModificationType` is unequal `NONE`) or if it is not an edit but short⁴ and between two edits. Otherwise `completeChange()` is invoked to commit the current open change. Listing 4.6 shows the code of this method.

⁴The length can be set in a constant variable. The default value is 14 text nodes (approx. 7 words separated by spaces).

WrapPiece If the content is an instance of `WrapPiece`, `handleWrap()` is invoked. If the instance's previous and following text piece has the same modification type, then the wrap is seen as a part of the change and added to it. Otherwise `completeChange()` is invoked.

```
private void handleText(Element<TextPiece> currElement)
{
    TextPiece t = (TextPiece) currElement.getContent();

    if (t.getChangeType() == ModificationType.NONE)
    {
        if (isOpenCurrentChange())
        {
            if (t.getNumberOfNodes() < LENGTH_OF_NOCHANGE_PIECE_FORCING_CHANGE_SPLIT
                && currElement.getNextElement() != null
                && currElement.getNextElement().getContent() instanceof TextPiece)
            {
                addToCurrentChange(currElement);
            }
            else
            {
                completeChange();
            }
        }
    }
    else
    {
        addToCurrentChange(currElement);
    }
}
}
```

Listing 4.6: StructureConsolidator: method `handleText()`

Example

This process step added instances of `ChangeToken` to the object structure. The change tokens combine near changes: In the example, the change `c1` is only formed by `t1`, since there is no other near change. However, `c2` units the tokens `t3`, `t4` and `t5`, because the distance between the `t3` (removed text) and `t5` (added text) is low (`countNodes` of `t4` is only 1).

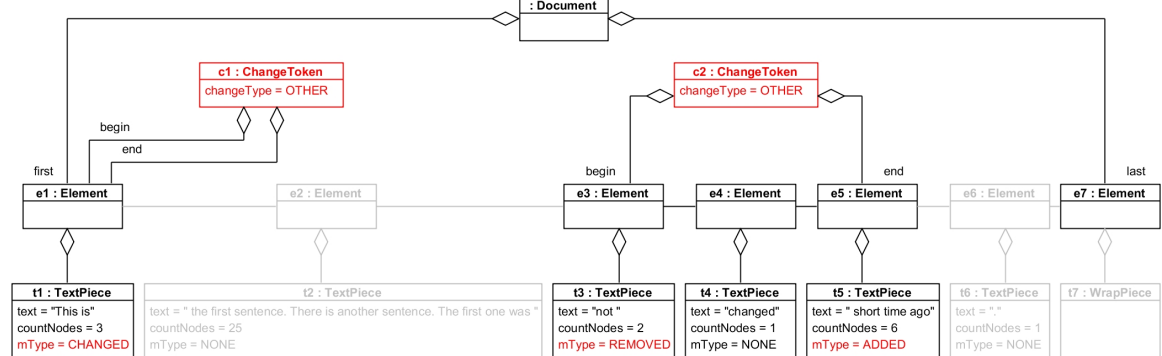


Figure 4.6: Objects of the StructureConsolidator output for the example

4.2.3 HtmlGenerator

In step 3 html code is generated as output of the whole calculation.

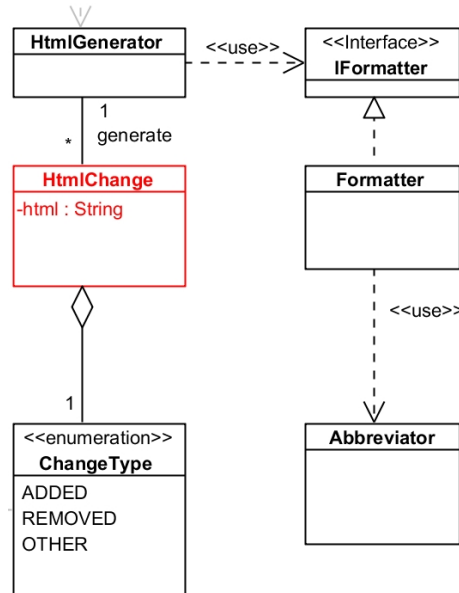


Figure 4.7: TriciaDiff: HtmlGenerator

Input

The HtmlGenerator receives the list of change tokens and an instance of IFormatter as input.

Output

The output of this processing step is a list of html pieces which are represented as HtmlChange. This data class contains the html string and the change type (ADDED, REMOVED or OTHER). The type is needed later in the overlay to choose the appropriate icon.

Calculation

For each change token of the input the method `getHtmlFormattedText()`, whose return result is used to instantiate a `HtmlChange`, is invoked. The implementation is as following (the used methods are described below):

```

public String getHtmlFormattedText(Formatter formatter)
{
    return getLeftContext(formatter) + getContent(formatter) + getRightContext(
        formatter);
}
  
```

Listing 4.7: HtmlGenerator: method `getHtmlFormattedText()`

Retrieving the Context The left and right context of a piece is gathered with the corresponding methods. Firstly, `getLeftContext()` retrieves the first element of the current change token. An element has a reference to the element on the left and on the right (see figure 4.6). Then the reference to the left is used to get the prior element. Subsequently, the prior element's left reference is used to move further left in the linked list. This is repeated until

- a specified number of nodes is reached, or
- a wrap node is identified and the already gathered text is not empty, or
- the list doesn't contain anymore elements.

This calculation for the right context works analogically. Before the text is returned, it is cut by the abbreviation algorithm. This is described in chapter 4.2.4.

Retrieving the Content The method `getContent()` retrieves the content of a change by concatenating the text of all elements of the change token. Before formatting the text, it is abbreviated in the centre if its length exceeds the specified one. Then the class `DefaultFormatter` carries out the formatting by adding html tags. This class implements the interface `IFormatter`. The *strategy pattern* makes it possible to use other formatting strategies by switching the formatter class.

Then the `HtmlGenerator` returns the html changes.

Example

The object diagram below shows the result of the html generation for the example.

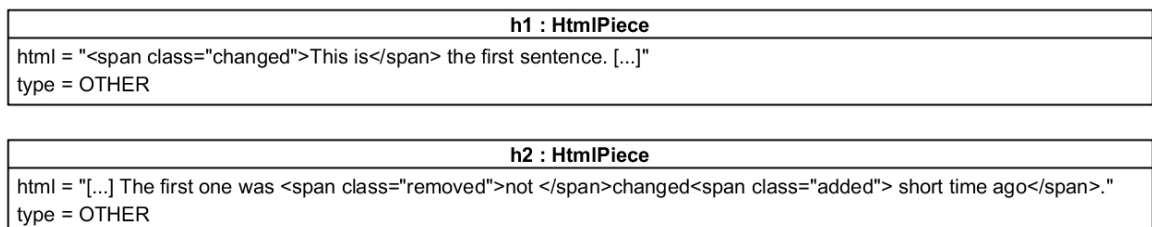


Figure 4.8: Objects of the `HtmlGenerator` calculation result for the example

4.2.4 Abbreviator

The class `Abbreviator` is used by the `DefaultFormatter` to abbreviate the content (if the length exceeds the specified value) and the context of a change piece. The behaviour of the abbreviation algorithm can be configured by some constants:

```

CUT_INSIDE_MAX_WORDS
  [maximum length of the content of a change piece until it is
  abbreviated in the centre]
  default value = 22
  
```

```
unit = words
CUT_OUTSIDE_MIN_WORDS
  [minimum length of the context on one side (might be fallen short off
  if not enough words are available)]
  default value = 3
  unit = words
CUT_OUTSIDE_DESIRED_WORDS
  [number of words used for the context if no preferred cut point
  (sentence end or beginning) is detected]
  default value = 6
  unit = words
CUT_OUTSIDE_MAX_WORDS
  [maximum length of the context]
  default value = 9
  unit = words
CUT_OUTSIDE_MAX_CHARS
  [maximum number of characters the context can have (does usually not
  take effect with the default value of CUT_OUTSIDE_MAX_WORDS)]
  default value = CUT_OUTSIDE_MAX_WORDS * 12
  unit = characters
```

Cutting the Content

If the content of a change is longer than `CUT_INSIDE_MAX_WORDS` words, text is cut out from the centre. At the cutting point “[...]” is inserted as a link. (By clicking the link a non-abbreviated version will be shown.)

Cutting the Context

When cutting the context, the abbreviation algorithm tries to keep whole sentences or to cut between subordinate clauses. Therefore, it attempts to identify the beginning of a sentence (respectively a sentence end if it is the right context) between the words on position `CUT_OUTSIDE_MIN_WORDS` and `CUT_OUTSIDE_MAX_CHARS`. This is mainly done by searching characters such as “.”, “,”, “!” ... If a preferred cutting point is found, then it is used to cut, otherwise the context is shortened at the position `CUT_OUTSIDE_DESIRED_WORDS`.

4.3 Aggregation of Similar Change Sets

In chapter 3.3 criteria for similar change sets were defined. Another technical constraint has to be added to these criteria: A (merged) change set can contain at the most one change of the type `GroupMembershipChange`.

If a user edits a persistent entity, the entities’ `applyPersist()` method is invoked. It updates the data in the database, creates a new change set and hands the set over to the `EventManager`.

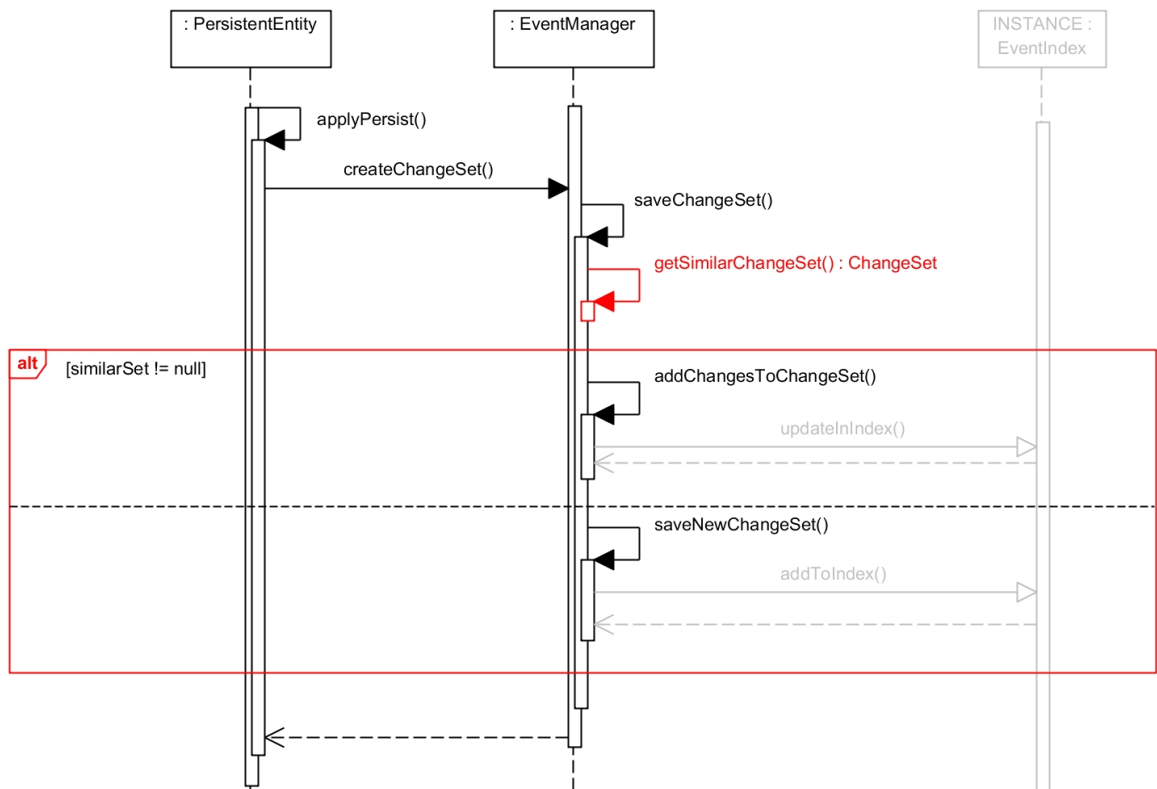


Figure 4.9: Sequence diagram showing the persist pass of change sets⁵

Now the method `getSimilarChangeSet()` of the event manager checks whether an earlier change set exists, which fits the similarity constraints. As the similarity check is realised after every edit, only the last change set has to be reviewed.

If exists one, it is returned and merged with the new change set: To do so, the edited entity is retrieved and a writable copy of it is created. All changes of both change sets are undone on the copy to get to the state before the changes. Then the static method `ChangeSet.getDifferences()` is invoked to get the difference of the two states as single changes. These are assigned to the earlier change set. Then the time stamp is adjusted and the set is updated in the database. There is the special case that there is no difference between both entity states, because the second edit reverted the first one. Then the previous change set which is already in the database is removed.

If there is no similar change set which fits the constraints the method returns null and the new set is persisted.

The sequence diagram in figure 4.9 illustrates this procedure. The following code gives an insight how the similarity check is carried out.

```

private static ChangeSet getSimilarChangeSet(Person person, PersistentEntity entity)
{
    Calendar calendar = Calendar.getInstance();
    calendar.add(Calendar.MINUTE, - TIME_DIFFERENCE_IN_MINUTES_CONFORMING_SIMILARITY);
    Timestamp tenMinutesAgo = new Timestamp(calendar.getTimeInMillis());
  
```

⁵Note: Return messages of recursive calls are not modelled.

```

Query q01 = new QueryEquals(ChangeSet.SCHEMA.prototype().entityUid, entity.getUid()
);
Query q02 = new QueryGreater(ChangeSet.SCHEMA.prototype().when, tenMinutesAgo);
Query q03 = new QueryEquals(ChangeSet.SCHEMA.prototype().type, ChangeSetType._edit)
;

Query q1 = new QueryAnd(q01, q02);
Query q = new QueryAnd(q1, q03);
q.addSortingCriterion(new Descending(ChangeSet.SCHEMA.prototype().when));

for (ChangeSet c : ChangeSet.SCHEMA.queryEntities(q, 1))
{
    if (c.person.get() != null && c.person.get().equals(person))
    {
        return c;
    }
}

return null;
}

```

Listing 4.8: EventManager: method to retrieve similar change sets

The screenshot shows a web interface with tabs for 'View', 'Details', 'Attachments', and 'Versions'. Below the tabs, it says 'Versions of Advanced seminar' with a warning icon. Below that, it says '(0 - 2 of 2)'. The table has three columns: 'Who', 'Properties', and 'When'. The first row shows 'Max Mustermann' with properties 'Content', 'Tags', 'Lecturer', 'Module No.', 'Language', 'Organizer', and 'Readers', and a timestamp of '1 minute ago'. The second row shows 'Max Mustermann' with the property 'Created' and a timestamp of '2 minutes ago'. A link 'Compare with current version' is visible in the second row. At the bottom, it says '1 (total result pages: 1)'.

Who	Properties	When
Max Mustermann	Content, Tags, Lecturer, Module No., Language, Organizer, Readers	1 minute ago
Max Mustermann	Created	2 minutes ago Compare with current version

Figure 4.10: A version history with the same edits as in figure 3.24 but with aggregation

4.4 Change Awareness

This section is about the implementation of the *signals* component. It describes the indexing and retrieving of events, the dashboard and the recent changes view, and the mail notification service with its settings.

4.4.1 Event Stream

Each time a user makes an edit, a change set is persisted or updated in the database. After that, the set is passed to the singleton class `EventIndex` which inherits `LuceneWrapper`. This class is responsible to keep the Lucene index concerning the changes up to date. In addition, it offers query functionalities.

Indexing

When an event has to be indexed, the `addToIndex()` method of `EventIndex` is invoked. There exist also methods to update or remove events. These methods call `getDocument()` of `Event` to create a Lucene document. The following fields are added to the document:

EVENT_ID
[contains the ID of the change set and is used to identify the document]
stored = true
indexed = true
analyzed = false

ENTITY_UID
[contains the UID (unique identifier) of the changed object]
stored = true
indexed = true
analyzed = false

PERSON_ID
[contains the ID of the editor (if a index rebuild is conducted, it is possible that the person is null)]
stored = true
indexed = true
analyzed = false

TIMESTAMP_FOR_SORTING
[contains a time stamp which is used to sort the events when querying these]
stored = false
indexed = true
analyzed = false

READ_ACCESS
[contains a string used to restrict the access]
stored = false
indexed = true
analyzed = true

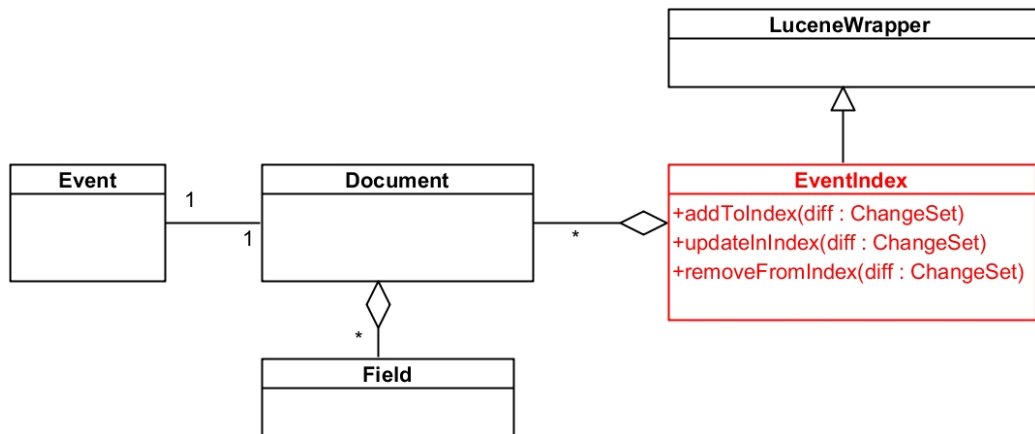
EVENT_TYPE
[contains the change set type (NEW, EDIT, REMOVE, UNDELETE)]
stored = false
indexed = true
analyzed = false

ENTITY_NAME
[contains the name of the changed entity (this field is used to display the name, even if the entity is no longer in the database (e.g. when showing a delete event))]
stored = true
indexed = true
analyzed = true

SPACE_NAME
[contains the name of the entities' space (if the entity has the mixin InSpace)]
stored = true
indexed = true
analyzed = true

TAGS
[contains the tags of the entity as a concatenated string; used for tag filtering]
stored = false
indexed = true
analyzed = true

Then the document is put into the index using the methods of the super class `LuceneWrapper`. A few persistent entities exist, whose objects are not supposed to be indexed, because they shouldn't appear in the event stream: These include instances of `ContentWatch`, `Settings`, `EventIndex`. These classes implement the marker interface `DontIndexChangeEvents` and will, therefore, be skipped in the indexing methods.

Figure 4.11: Class `EventIndex`

Querying

A lucene query consists of basic terms, which can be combined to complex queries. The combination of terms and/or sub-queries can be done by adding these to a Boolean query. When adding a clause to a Boolean query, the occurrence has to be specified: `Occur.MUST` defines that the clause must occur, `MUST_NOT` that the clause must not occur, and `SHOULD` expresses that at least one of the sub-clauses has to be evaluated to be true.

The default maximum length of the query is 1024 clauses. If it is necessary, this value can be increased by invoking `BooleanQuery.setMaxClauseCount()`.

To retrieve events about watched objects for the dashboard, both the database and the Lucene index are needed. However, an overlapping join query is not possible. Therefore, all watched objects need to be loaded from the database and subsequently, the Lucene query is built: It adds for each object a term which works on the entity ID to an OR clause. If the watched object is a space, then changed items in the space should appear, too. To do so, a further OR clause operating on the space field is added. (Listing 4.9) contains the code which adds these clauses.) Further clauses are added if necessary (e.g. a `MUST_NOT` term to exclude the user's own changes).

For events requested for the "Recent changes" page, a clause for each active filter is appended to the query.

At the end an access rights filter is added to the query to avoid exceptions, which could be thrown later (when reloading further data from the database) due to denied read access.

```

public void addConstraintWatchedObjects(Person person)
{
    BooleanQuery bQ = new BooleanQuery();

    for (ContentWatch currentWatch : person.watches.getAssets())
  
```

```

{
    Entity watchedObject = currentWatch.watchedObject.get().getEntity();

    Term t;

    if (watchedObject.hasMixin(Space.class))
    {
        t = new Term(Event.FIELD_SPACE_NAME, watchedObject.adapt(Space.class).
            getName());
    }
    else
    {
        t = new Term(Event.FIELD_ENTITY_UID, watchedObject.getUid());
    }

    TermQuery tQ = new TermQuery(t);
    bQ.add(tQ, Occur.SHOULD);
}

addSubQuery(bQ);
}

```

Listing 4.9: EventQueryBuilder: method which adds clauses to get events of watched objects

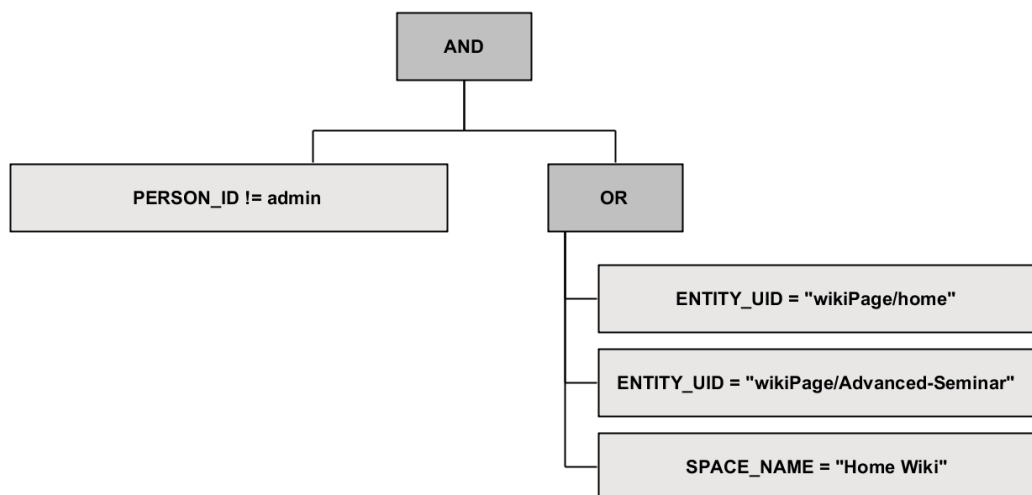


Figure 4.12: A sample query for a dashboard: own changes are hidden, the pages “Home” and “Advanced Seminar” and the space “Home Wiki” are watched

4.4.2 New User Interfaces

New user interfaces were developed for the awareness component: The dashboard visualises edits which concern watched objects, whereas the page “Recent changes” shows all edits.

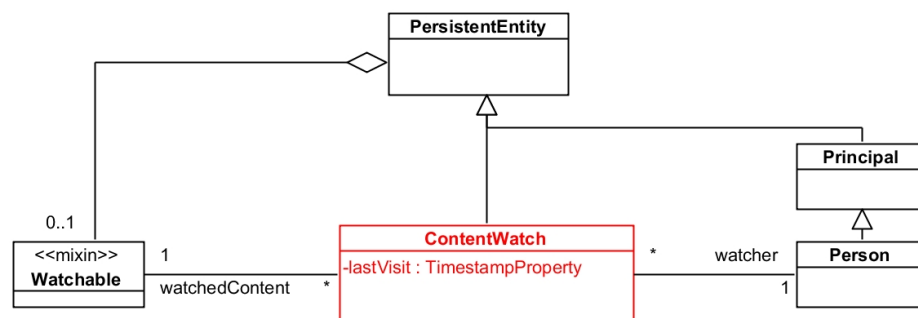
Watch Action

In order to be able to use the dashboard, the watch list must contain some items. Tricia allows now watching entities of the following types:

- wiki page
- wiki
- blog
- user group
- document⁶
- directory

The classes representing these types have the mandatory mixin `Watchable`. In the presentation view of objects there is a link to the `WatchHandler`, which is invoked with the entities' ID. The action (start or stop) can be specified to enforce it, however, it is optional. If not specified, the current state is reversed. It is not necessary to provide the id of the user, because it can be read out by the handler using the method `SessionLocal.getUser()`. If a space is watched, all elements of the space are considered to be watched implicitly, too.

The execution logic either creates a new instance of `ContentWatch`, which holds an association to the mixin of the entity and to the user, and stores it in the database, or deletes the existing object. An object can be watched by many users and each user can watch many. A deletion of the user or of the object is cascaded to the content watch. Figure 4.13 shows the class diagram to this issue.

Figure 4.13: Class `ContentWatch`

Dashboard

The dashboard can be accessed via the main menu, which is visible on every page. However, the user has to be logged in to see the link and to be able to use the dashboard.

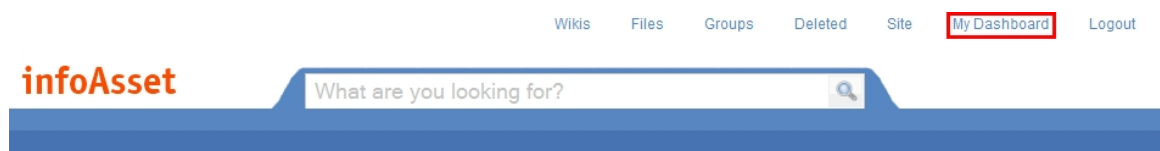


Figure 4.14: Link to the dashboard

⁶A document is a file in a directory.

The dashboard links the page “All recent changes” and the notification mail settings. In addition, there is a link to view the profile of the current user (because the dashboard link in the menu replaced this link).

If the user watches no objects, a help page will be shown. It gives an introduction on how to put objects on the watch list and to use the dashboard. Otherwise the dashboard’s UI interface, illustrated in figure 4.15, is shown (see chapter 3.4.2 for the design decisions of the mockup).

The implementation consists of the handler class and an html template. The template is composed of:

- a list substitution for the **watched objects**: The user’s watched pages are queried from the database and then used in the substitution method.
- a template substitution for the **change events**: For first, the Lucene index is queried for the change events (of the watched objects). The number of requested results is `RESULTS_PER_PAGE7 + 1` in order to see, if the next link to a further page is appropriate. Then the `ChangeEventHandler` is instantiated, the events are handed over to it and the template evaluation method is invoked. The result of the evaluation replaces the event placeholders.

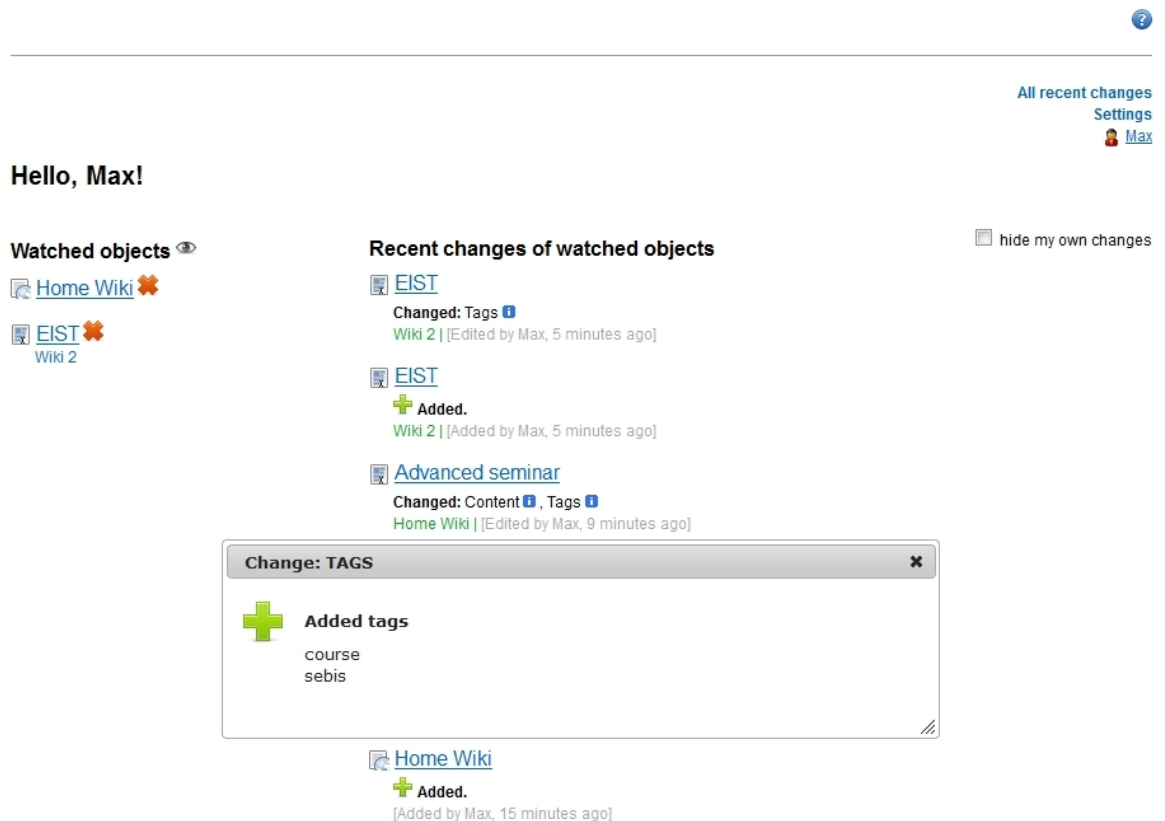


Figure 4.15: Screenshot of the dashboard

⁷RESULTS_PER_PAGE is 10 by default.

The dashboard was not exactly implemented as planned. The view to show changes since the last visit was not implemented. As this is the first iteration, the user acceptance should be awaited first. However, it can be extended easily later, because major parts of its logic are already implemented in other functionalities and can be used.

Recent Changes

In addition, the “All recent changes” page has been developed. Its template contains a placeholder for the side bar with the filter options and for the change events. Both are replaced with template substitutions.

Like in the dashboard, the replacement of the events is carried out using the `ChangeEventHandler`. However, the presentation is slightly different. (For instance, these events have a link to add an (unwatched) object to the personal watch list.) The data handed over to the change event handler is retrieved by running a Lucene query, which contains a clause for each (active) filter.

The page is reloaded when a filter changes. The filters are passed as get parameters, therefore the selections can be bookmarked.

The user has to be logged in to see this page.

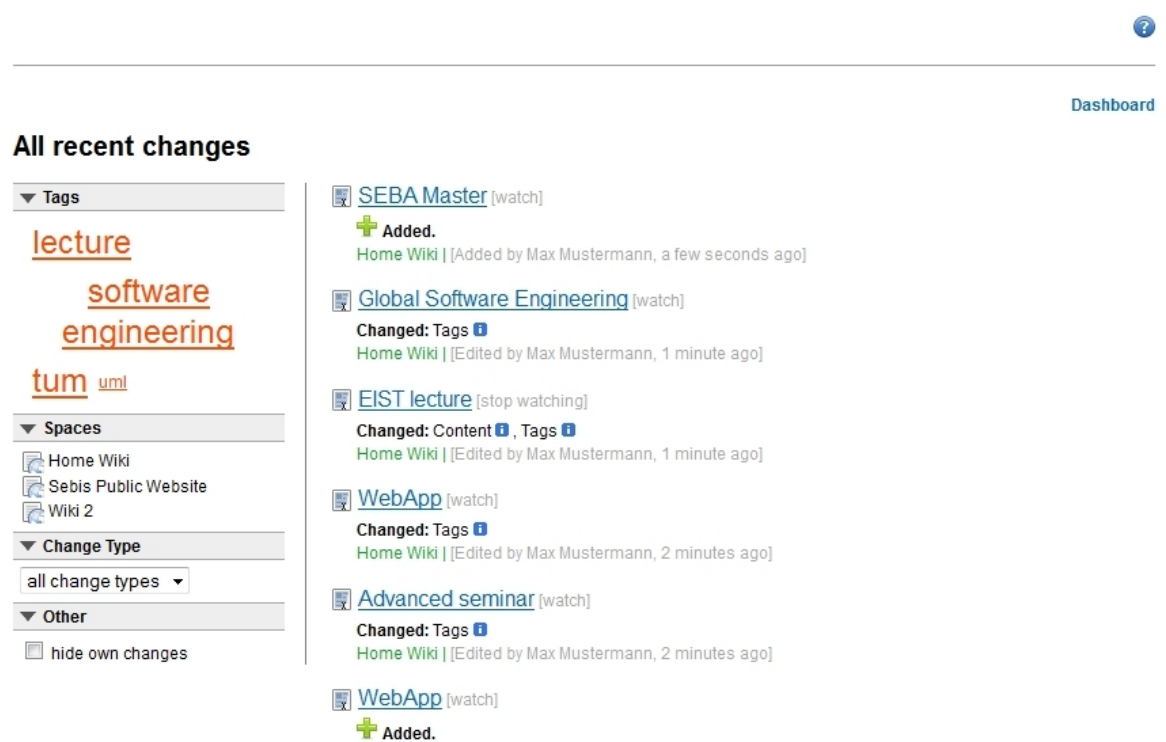


Figure 4.16: Screenshot of the recent changes page

ChangeEventsHandler

Both awareness pages unify the presentation of the events in the class `ChangeEventsHandler`. This handler is invoked with an index query result. Then the template evaluation is run.

In order to have all necessary information, the entity and the change set of each event are loaded from the database (using the IDs in the event data). Afterwards the placeholders of the template are substituted by the data. The time information is prepared in a user-friendly format. This is done in the self developed `DateUtil` class, which generates strings such as “a few seconds ago”, “8 minutes ago”, or “yesterday”.

Below the ten results per page, a navigation bar is shown, if further pages are available.

4.4.3 Mail Notification Service

Supplementary to the user interfaces (pull concept) characterised above, a notification service (push concept) exists, which sends mails summarising the latest changes of watched objects. Its logic is implemented in the class `NotificationMailSender`, which is periodically invoked by a timer. Its run method iterates over all registered users. For each one it checks, whether notifications are enabled and what the chosen time interval is. Available time intervals are “once an hour”, “once a day”, and “once a week”.

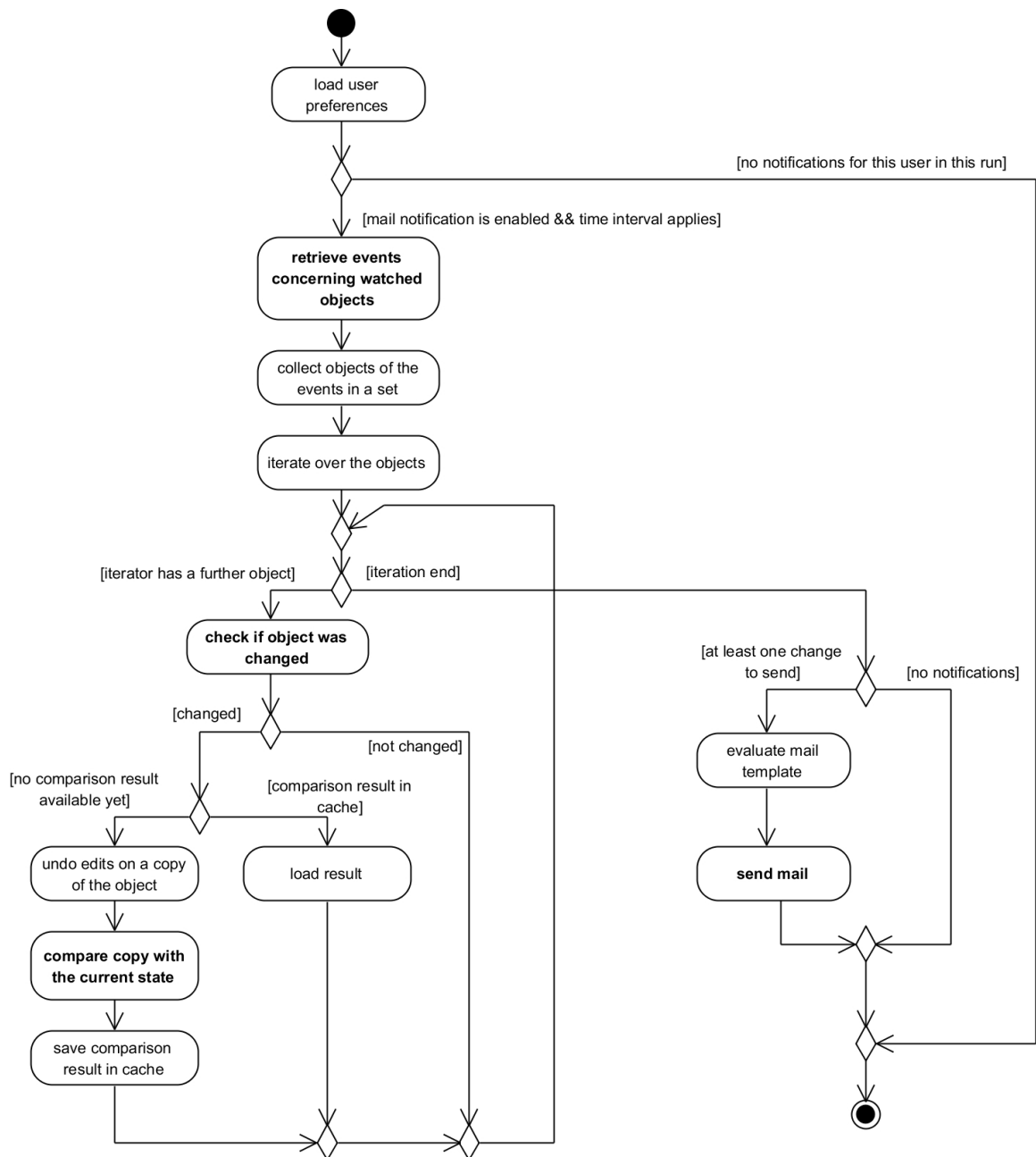


Figure 4.17: Activity diagram illustrating the creation of mail notifications

If the user's chosen time interval applies to this run, all change events for his watched objects, which were edited in the time span since the last sent mail, are retrieved from the Lucene index. Then a method goes through the events, gets the IDs of the concerned changed entities and loads these from the database. The result is held in a set, so that each entity is contained at the most once.

Then the method `getCalculatedContainer()` is invoked for all objects in the set. It checks, if the hash map `objectChangesCache` contains already a change comparison

result of the object for the given time interval. If so, the result is retrieved from the map, otherwise the change set is calculated. The calculation is as follows: all change sets concerning this object, which have a time stamp in the interval, are loaded from the database and then reverted on a copy of the watched object. Subsequently, the difference between the copy (which represents the state before all changes of the time interval) and the current state of the entity is computed. This is done via the static `getDifferences()` method of the class `ChangeSet`. Afterwards the result is cached in the hash map to be reused for other users with the same notification interval.

If there was at least one watched object, which was changed, the method `sendMail()` is invoked with a list of the change sets as a parameter. This method calls `getMail()` of the `NotificationMail` class, which evaluates the mail html template. The presentation of the changes described by this template is not exactly the same as the one used in the overlays of the version history. This template is optimised for the use in mails, on that account it doesn't contain any images and the changes are generally displayed more compact. Different versions of it exist; these differ in the level of detail. The user can set his preferred level in the mail notification settings (see chapter 4.4.3).

At the end, the computed html string is used in the body of the mail and subsequently, the mail is sent. After that, the method runs the same procedure for the next user.

The notification mails were implemented exactly as planned, therefore the mockup of figure 3.32 can be considered as a screenshot of the implementation, too.

```
private void sendMail(Person person, List<ChangeSetContainer> changesForThisMail)
{
    Mail currentMail = notificationMail.getMail(person, changesForThisMail);

    currentMail.init(person.login.get());

    currentMail.sendMail();
}
```

Listing 4.10: `NotificationMailSenderTask`: method `sendMail()`

```
private ChangeSetContainer getCalculatedContainer(PersistentEntity entity)
{
    final String uid = entity.getUId();

    if (! objectChangesCache.containsKey(uid))
    {
        objectChangesCache.put(uid, calculateChangeSet(entity));
    }

    return objectChangesCache.get(uid);
}
```

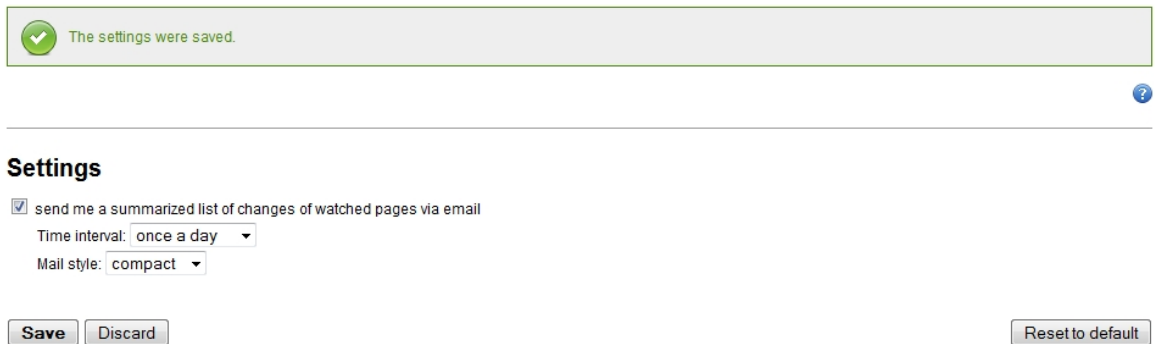
Listing 4.11: `NotificationMailSenderTask`: method which gets the notification mail change set for an entity

Settings

The settings page, which can be reached by a link in the dashboard, allows the configuration of the mail notification service. At the moment it offers three options:

- **Mail notifications** can be **enabled or disabled**. The default value is disabled. The following two options are only applied, if notifications are enabled.

- The **time interval** of the notifications can be specified. Available options are “once an hour”, “once a day” (default value) and “once a week”.
- There exist four different **mail styles** which differ in the level of details. The choice “minimal” only lists the changed objects, but does not provide any information about what changed. In contrast, “compact” names the changed features, but still suppresses the details. The next more meaningful choice “dynamic” shows details of some change types, which are considered to be relevant to the user, because they contain information about the object. The chosen types are `RichStringChange` and `HybridPropertyChange`.⁸ In contrast, for instance the tags just support the categorisation, but are not supposed to contain important information which the user has to be aware about. Finally, the choice “complete” contains the details of each changed feature not regarding the type. “dynamic” is preselected by default.



✓ The settings were saved.

Settings

send me a summarized list of changes of watched pages via email

Time interval:

Mail style:

Figure 4.18: Settings page to configure the mail notifications

⁸The choice can be changed in a constant class variable in the code without any effort.

5 Summary

The result of this thesis is the implementation of awareness functionalities in the core of Tricia. These functionalities allow for tracing changes:

- A user can see all recent changes concerning objects that can be accessed with his access rights. The list of the changes can be filtered.
- It is now possible to watch objects. Changes of these objects appear in the dashboard view (pull concept) and can be signalled to the user via notification mails (push concept).
- The aggregation of changes merges similar edits. This keeps the version history of objects and the dashboard / recent changes page more clearly.
- The presentation of changes has been improved. It focuses on the difference between the two states of an entities' changed feature.

Further developments could be:

- A second dashboard view, which shows one summarised change entry for each watched object, which was edited since the user's last visit of it, could be added.
- The platform could be supplemented by a functionality to detect and display recently added internal incoming links to an entity.

Appendix

Bibliography

- [BMN10] T. Büchner, F. Matthes, and C. Neubert. Data model driven implementation of web cooperation systems with tricia. In *3rd International Conference on Objects and Databases (ICOODB)*, 2010.
- [Coa05] T. Coates. Website. http://www.plasticbag.org/archives/2005/01/an_addendum_to_a_definition_of_social_software/, 2005. Visited on August 4th 2011.
- [DB92] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work 1992*, 1992.
- [DBMM93] P. Dourish, V. Bellotti, W. Mackay, and C. Y. Ma. Information and context: Lessons from a study of two shared information systems. In *Proceedings of the conference on Organizational computing systems 1993*. ACM Press, 1993.
- [Few06] S. Few. *Information Dashboard Design: The Effective Visual Communication of Data*. O'Reilly Media, 2006.
- [GHVJ09] E. Gamma, R. Helm, J. Vlissides, and R. Johnson. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 2009.
- [infa] infoAsset. Product presentation of tricia. <http://www.infoasset.de/file/attachments/wikis/infoasset/tricia/110602%20infoAsset%20Tricia%20EN.pdf>. Visited on August 11th 2011.
- [infb] infoAsset. Website. <http://www.infoasset.de/tricia-product-features>. Visited on June 1st 2011.
- [ISG09] J. Ibáñez, O. Serrano, and D. García. Emotinet: A framework for the development of social awareness systems. In *Awareness Systems*, pages 291–311. Springer Verlag London, 2009.
- [KR07] M. Koch and A. Richter. *Enterprise 2.0: Planung, Einführung und erfolgreicher Einsatz von Social Software in Unternehmen*. Oldenbourg Wissenschaftsverlag, 2007.
- [Löv91] L. Lövstrand. Being selectively aware with the khronika system. In L. Bannon, M. Robinson, and K. Schmidt, editors, *Proceedings of the Second European Conference on Computer-Supported Cooperative Work*, pages 265–277. Kluwer Academic Publishers, 1991.

Bibliography

- [McA06] A. P. McAfee. Enterprise 2.0: The dawn of emergent collaboration. Technical report, MIT Sloan School of Management, 2006.
- [Ras00] J. Raskin. *The humane interface: new directions for designing interactive systems*. Addison-Wesley Professional, 2000.
- [RM09] M. Rittenbruch and G. McEwan. An historical reflection of awareness in collaboration. In *Awareness Systems*, pages 3–48. Springer Verlag London, 2009.
- [Tid11] J. Tidwell. *Designing Interfaces*, volume 2. O’Reilly Media, 2011.
- [ZDN] ZDNet. Website. <http://www.zdnet.com/blog/hinchcliffe/the-state-of-enterprise-20/143>. Visited on August 3rd 2011.

List of Figures

1.1	Social software triangle by Koch/Richter	2
2.1	Property types offered by Tricia	6
2.2	Classes <code>Asset</code> , <code>Entity</code> and <code>Mixin</code>	7
2.3	Group structure in Tricia	7
2.4	Important Tricia plugins	9
2.5	Model representation of user activity	10
2.6	Structure of the Lucene index	11
3.1	Use cases for the awareness component from the user's point of view	13
3.2	Different change types	14
3.3	<code>ChangeGroupMembershipChange</code> , before	14
3.4	<code>ChangeGroupMembershipChange</code> , mockup	15
3.5	<code>ChangeGroupMembershipChange</code> , after	15
3.6	<code>DomainValueChange</code> , before	15
3.7	<code>DomainValueChange</code> , mockup	16
3.8	<code>DomainValueChange</code> , after	16
3.9	<code>SimpleValueChange</code> , before	16
3.10	<code>SimpleValueChange</code> , mockup	17
3.11	<code>SimpleValueChange</code> , after	17
3.12	<code>TagsChange</code> , before	18
3.13	<code>TagsChange</code> , mockup	18
3.14	<code>TagsChange</code> , after	18
3.15	<code>HybridPropertiesChange</code> , before	19
3.16	<code>HybridPropertiesChange</code> , mockup	19
3.17	<code>HybridPropertiesChange</code> , after	19
3.18	<code>RichStringChange</code> , before	21
3.19	<code>RichStringChange</code> , mockup	21
3.20	<code>RichStringChange</code> , after	22
3.21	<code>RoleChange</code> , before	22
3.22	<code>RoleChange</code> , mockup	23
3.23	<code>RoleChange</code> , after	23
3.24	An entities' version history with a lot of single changes	24
3.25	Dashboard of BusinessWiki	25
3.26	Mockup of the planned dashboard	26
3.27	Recent changes view of the English Wikipedia	28
3.28	Recent changes view of the wiki of ubuntuusers.de	29
3.29	Recent changes view of Wikispaces' wikis	30
3.30	Mockup of the planned recent changes page	31

3.31	Mail notification sent by the Ubuntu wiki	32
3.32	Planned look of the mail notification	33
4.1	Overview of TriciaDiff	36
4.2	TriciaDiff: StructureAnalyzer	37
4.3	Input of the StructureAnalyzer	38
4.4	Objects of the StructureAnalyzer calculation result for the example	39
4.5	TriciaDiff: StructureConsolidator	40
4.6	Objects of the StructureConsolidator output for the example	41
4.7	TriciaDiff: HtmlGenerator	42
4.8	Objects of the HtmlGenerator calculation result for the example	43
4.9	Sequence diagram showing the persist pass of change sets	45
4.10	A version history with the same edits as in figure 3.24 but with aggregation	46
4.11	Class <code>EventIndex</code>	48
4.12	A sample query for a dashboard: own changes are hidden, the pages “Home” and “Advanced Seminar” and the space “Home Wiki” are watched	49
4.13	Class <code>ContentWatch</code>	50
4.14	Link to the dashboard	50
4.15	Screenshot of the dashboard	51
4.16	Screenshot of the recent changes page	52
4.17	Activity diagram illustrating the creation of mail notifications	54
4.18	Settings page to configure the mail notifications	56

Listings

2.1	Simple template file	8
2.2	Simple print substitution	8
4.1	TagsChange: excerpt of the template code	34
4.2	TagsChange: method details()	35
4.3	TagsChange: template substitution's init() method	35
4.4	TagsChange: excerpt of the method putSubstitutions()	36
4.5	StructureAnalyzer: method addText() to append text of the same type	39
4.6	StructureConsolidator: method handleText()	41
4.7	HtmlGenerator: method getHtmlFormattedText()	42
4.8	EventManager: method to retrieve similar change sets	45
4.9	EventQueryBuilder: method which adds clauses to get events of watched objects	48
4.10	NotificationMailSenderTask: method sendMail()	55
4.11	NotificationMailSenderTask: method which gets the notification mail change set for an entity	55