Management and Security of Collaborative Web Environments

Diplomarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

David Vogel

2004

Leiter der Arbeit:

Prof. Dr. Stéphane Ducasse
Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik

Further information about this work and an *online* version of this document can be found at:
http://www.iam.unibe.ch/~scg/smallwiki/

The address of the author:

David Vogel
Huberstrasse 2
CH-3008 Bern

or

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
vogel@iam.unibe.ch

Abstract

A *Wiki* is a collaboratively-written website, also known as a Wiki space driven by Wiki server software implemented with any programming language. Wiki supports hyperlinks and has a simple text syntax for creating new pages and crosslinks between internal pages on the fly. Its *open editing concept* allows users to freely create and edit web page content using any web browser.

Like many simple concepts, *open editing* has profound and subtle effects on Wiki usage. Anybody can add, edit, and maybe delete pages of the Wiki. That is why a *vandal* is able to damage or abuse a Wiki by deleting parts of the Wiki site, defacing a page or uploading files in order to use the Wiki as an interim storage. The management of a Wiki gets clumsy if there is just one administrator who has to adjust the damage in the entire Wiki space. The assessment of these problems depends on the Wiki application area.

This diploma gives an general overview of *collaboration models* and of the *Wiki concept*, in particular of the *SmallWiki* implementation and its design. We introduce the SmallWiki Default Security Model and its enhancement - the SmallWiki Extended Security Model- in order to solve the problems of *vandalism* and of *central management*. This fine-grained security model is explained and it is shown how a *Wiki administrator* can manage the permissions for SmallWiki users at any point in the Wiki site, and how the pattern of *save delegation* is applied.

The characteristics of the new model are also described on a formal level. Additionally the security user interface of SmallWiki is depicted in detail.

This solution is validated by describing in detail common scenarios.

Acknowledgments

First I wish to thank my supervisor Prof. Dr. Stéphane Ducasse for his guidance, and the head of the group Prof. Dr. Oscar Nierstrasz for giving the opportunity to work in his group. Thanks also to all other group members for providing a pleasant working atmosphere.

I also want to thank all those people who have contributed to whatever knowledge I have of object-oriented thinking and Smalltalk programming. Many people have contributed to my education, including the authors of various books.

In particular, I'd like to thank some friends from university for special contributions: Tobias Aebi for the philosophy and information he imparted when he came in as a voluntary consultant to review the design and strategy of my work. Together we have drawn many Wiki trees. Michele Lanza for providing the patience, wisdom, and time to review my work; for always brightening up the life at the university, for having the perfect timing for recreative coffee breaks, and for being a friend. Calogero Butera for not letting me suffer alone and for sharing the worries of a diploma student; he understood well. Daniele Talerico, Frank Buchli, Tom Bühler, and all other students from the pool for the good times we shared at the perfect air-conditioned attic.

I want to thank my boss Dr. Igor Metz for his support and for giving me the opportunity to work at flexible time at Glue S.E. AG.

Most of all, I want to acknowledge and thank the people beyond university, whose lives were most affected by this work: my parents and my brothers for the support and the encouragement they gave to me, and my love and bride-to-be, Kerstin, to whom I offer the greatest thanks. On top of her own busy life she took on many additional duties as I spent long hours at the computer, and she always gave me the continuous support I needed. Thanks, I couldnt have done it without you!

David Vogel
May 2004

Contents

Chapter 1

Introduction

1.1 Wiki, a web-based Collaboration Tool

Interactive web page access occurs when the members of a group can *collectively collaborate* (view and edit) the content of a web site through a comfortable user interface. Usually the HTTP protocol is used for data transfer. The original idea of the web implied that anyone could be both *reader* and *writer*, but the technical complexity of editing and uploading HTML pages makes the publishing part of the web inaccessible to many users. The problem becomes even more complicated when a team of technically unskilled writers want to publish content, all without getting in each other's way or accidentally destroying the site's framework.

Content Management System (*CMS*) is an umbrella term for many different types of collaboration systems. Most systems have a method of revision control. Some allow only the publication of news, others provide the administrator with many different categories of content; some have no real concept of security other than *can write* and *cannot write*, others allow fine-grained classification of all users. *Wikis* are a type of content management system, working by an access-control free philosophy but nevertheless greatly simplifying the process of editing and maintaining pages. Wikis do not work with copies of the shared material; the original document is available at all times, always in its latest version.

Wiki Wiki means *super fast* in the Hawaiian language, and it is the speed of creating and updating pages that is one of the defining aspects of Wiki technology. Generally, there is no prior review before modifications are accepted, and most Wikis are *open* to the general public or at least to all persons who also have access to the Wiki server. In fact, even registration of a user account is often not required.

Many implementations have been made using different programming languages. This work focuses on *SmallWiki* [1], a Wiki implementation written in *VisualWorks Smalltalk* [2].

1.2 Problems

There exist problems for many Wiki implementations. Some are to be taken seriously, whereas others might not be very important. The assessment of this importance depends on the Wiki application area. In this section, we list major problems:

1. **Vandalism.** As a consequence of the mostly used *open editing* concept, any Wiki user is allowed to *add, edit,* and maybe to *delete* pages of the Wiki. Therefore a *vandal* is able to damage intentionally a Wiki: delete large parts of the Wiki space, deface a page or abuse the Wiki server as a file archive (*e.g.*, *MP3*-archive). It is cumbersome to fix the damage over and over, since it might be difficult to detect the changes a vandal has made to the Wiki site.

2. **Web space for a Closed Groups.** There is the need to setup a Wiki space for an enclosed group of users. Thus only the members of that group should have access to the Wiki site. *E.g.*, a class of users are only able to read, others are allowed to edit the content of the site.

3. **Central Management with One Administrator.** The idea of having *one administrator for the entire site* might be acceptable for small Wiki sites with portions that do not have to be maintained perfectly. But the maintenance gets clumsy if we run a large Wiki site with just one administrator who has to adjust the damage and manage the security settings in the entire Wiki space.

1.3 Solutions

In this work, we offer solutions to the listed problems:

1. **Vandalism.** Most public Wikis avoid mandatory registration procedures. Nevertheless, many of the major Wiki engines (including MediaWiki, [3] MoinMoin [4], UseModWiki [5] and TWiki [6] already provide ways to limit write access. For small Wiki sites, a common defense against a persistent *vandal* is to simply let them deface as many pages as he wants to, and to then quickly revert the pages after the vandal has left. As an emergency measure, some Wikis allow switching the database to read-only mode, or letting only users registered up to a cutoff date continue editing. Generally speaking, however, any damage that is done by a *vandal* can be reverted rather quickly. More problematic are subtle errors inserted into pages which go undetected, for example changing of dates on an online agenda in a Wiki. To solve the problem of *vandalism*, we introduce SmallWiki, a Wiki implementation written in VisualWorks Smalltalk. Its *SmallWiki Default Security Model* provides a security management approach where we can adjust the

privileges for different classes of users for any resources in the Wiki space via a comfortable user interface.

2. **Web space for a Closed Groups.** With this security model, we can also easily setup a web space for a closed group of users.

3. **Central Management with One Administrator.** In order to make the management of the Wiki space more flexible, the site administrator should be able to *delegate* certain fields of responsibility to other administrators. This *pattern of delegation* described in Section 5.1.4 is very central to the *SmallWiki Extended Security Model* . This model is an enhancement of the *SmallWiki Default Security Model* . It encourages us to collect resources in folders and in their sub-folders together and then to design responsible administrators to manage their contents. It is the assigned responsibility of these administrators to manage the user privileges on their part of the Wiki site.

1.4 Organization of the Document

- In **Chapter 2** we give a general overview of different web-based collaboration models. We introduce Wiki as a web-based collaboration in multiuser context, in addition to being a tool to collect and cross-reference information, and we explain the Wiki concept.

- **Chapter 3** provides documentation of the SmallWiki implementation with UML-diagrams about the most important classes, their responsibilities, and their use. The basis of this chapter is the official documentation of SmallWiki [1].

- In **Chapter 4** we discuss about security terms such as risk, threat, vulnerability, user authentication and authorization. It is shown how security is guaranteed in SmallWiki. Therefore we explain the SmallWiki Default Security Model- an easy and powerful system.

- In **Chapter 5** we provide the new model *SmallWiki Extended Security Model* that fulfils the pattern of safe delegation with several local sector administrators.

- **Chapter 6** focuses more closely on administering users, building roles, mapping roles to permissions, and creating security policies with one or several administrators for a Wiki site. We build several security scenarios in order to validate the reliability of the new model.

- In **Chapter 7** we give a formal description to the SmallWiki Extended Security Model.

- **Appendix A** lists the terms used in this document.

- In **Appendix B** we give all the needed information to download, configurate and run SmallWiki with the security extension. The basis of this chapter is the official documentation of SmallWiki [1].
- In **Appendix C** we provide explanations related to the user graphical interface.

Chapter 2

Wiki, a web-based Collaboration Model

This chapter gives a general overview of different web-based *collaboration models*, such as *e-mail exchange*, *shared access*, and *interactive pages*. We illustrate their characteristics and differences. Requirements and features of web-based collaborations are shown. We introduce *Wiki* as a web-based collaboration in multiuser context, in addition to being a tool to collect and cross-reference information, and we explain the Wiki *concept*.

2.1 Collaboration

A *collaboration* is the act of combining the efforts of several parties. These may include systems used for programming, writing to each other.

Collaboration brings consensus allowing people to work towards the same objectives. When information flows through successful communication between people, the collaborative effort enables possibilities for the sharing of knowledge and better education.

The internet allows people to collaborate without physical proximity, risk of infection, and without the use of polluting fossil fuels for personal transport. Such computer-based collaborations make it possible to share know-how very quickly among people from different countries - independently of their social standing.

2.1.1 Computer-served Collaboration Models

[7] There exists different kinds of computer-based collaborations, such as:

E-Mail. *E-Mail exchange* provides direct exchanges between the members of a

collaborative group. This is very simple and requires only that members have e-mail capability. Either all members receive copies of the message or the messages are being broadcasted as a mailing list. This technology is pure *push*, and it is up to the recipient to sort, archive, and make order of the e-mail flow. A public form of e-mail is the *Internet newsgroup*, where postings are broadcasted to all participating newsgroup servers and kept there for access. The newsgroups readers can browse message headers on the newsgroup server and load selected messages to their local system to read. Unfortunately, the postings can not be edited or easily cross-linked in any way useful to the group unless they are collected with extra information/annotation into a central archive (see Figure 2.1).

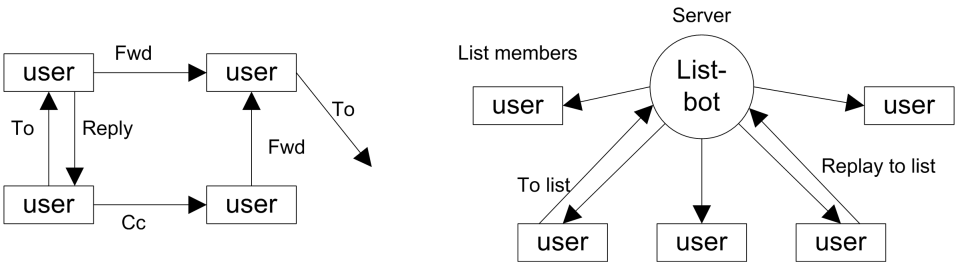


Figure 2.1: Model of e-mail exchange and of a mailing list.

Shared Files. *Shared access* means that members can directly access the same files - or a copy of them - of a repository on a server, basically via FTP (*File Transfer Protocol*) or SSH (*Secure Shell*). The files can be of any type. This might be the most common model for corporate network collaboration. Shared access is almost always combined with some form of *access control*: different members or different groups of members have different permissions on certain parts of the shared file-system. This is achieved by creating different kinds of roles, such as administrator, contributor, reviewer and reader (see Figure 2.2).

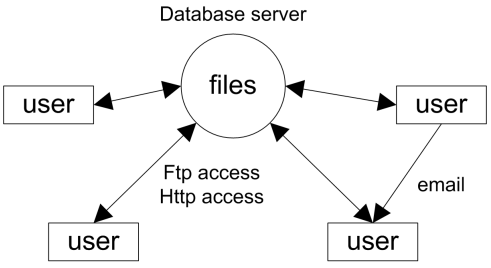


Figure 2.2: Model of shared shared files, *e.g.*, a database.

Peer-to-Peer (*P2P*) file sharing. *Peer-to-peer computing* connects together a large number of different computers, thereby pooling their resources. It provides a very efficient way of storing and accessing large amounts of data. *E.g.*, individual people store files that they want to share on their hard disks and share them directly with other people. The users run a piece of software that makes this sharing possible. Each user machine becomes a mini server. However, the way *P2P* networks are currently organised means that when the network involves a very large amount of machines, searching becomes very slow. This can be solved by using centralised servers to take care of key tasks, but this means the whole network crashes if those machines fail, unless a client becomes a server (supernode) *dynamically* for a certain time (see Figure 2.3 and Figure 2.4).

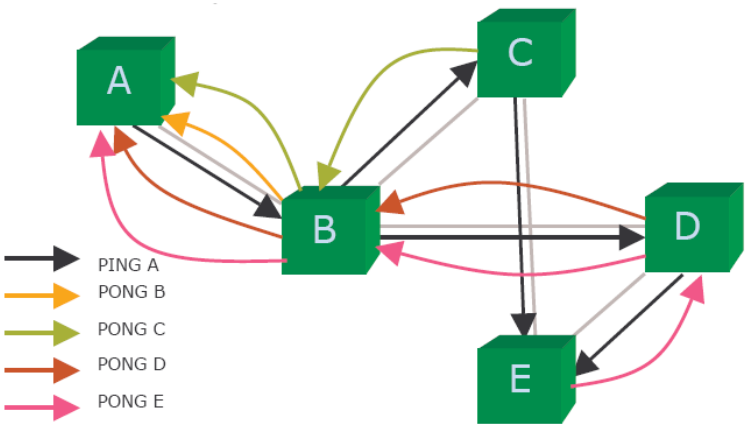


Figure 2.3: **Model of decentralised Peer-to-Peer file sharing.** The client *A* broadcasts the request to all other nodes, gets answers from all nodes, and chooses the one with best transfer rate.

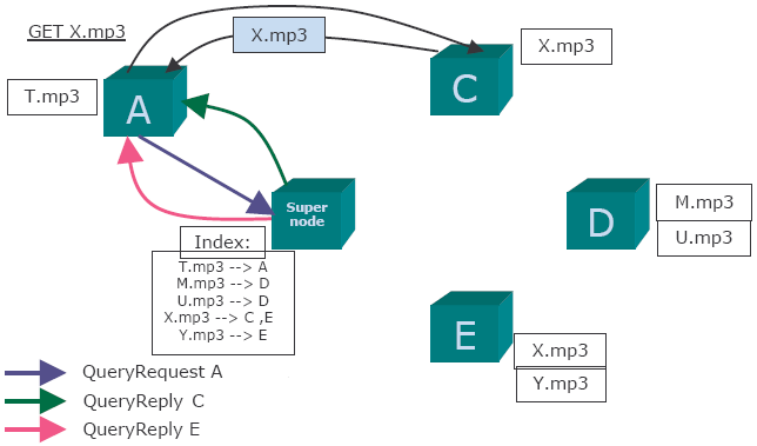


Figure 2.4: **Model of centralised Peer-to-Peer file sharing with a supernode.** The client *A* sends keywords to search with to the supernode, the supernode returns a list of hosts - $< ip\_address, portnum >$, the client pings these nodes in order to find out their transfer rates, and sends the request to the host *C* with the best transfer rate.

Shared Web Server Pages, Content Management Systems (*e.g.*, Wiki). Interactive page access occurs when the members of a group can *collectively collaborate* (view and edit) the content of a site through a comfortable user interface. Usually the HTTP protocol is used for the data transfer. The original idea of the web implied that anyone could both be *reader* and *writer*, but the technical complexity of editing and uploading HTML pages made the publishing part of the web inaccessible to many users. The problem becomes more complicated when a team of technically unskilled writers wants to publish content, without getting in each others way or accidentally destroying the site's framework.

Content Management System (*CMS*) is an umbrella term for many different types of collaboration systems. Most systems have a method of revision control. Some allow only the publication of news, others provide the administrator with many different categories of content; some have no real concept of security other than *can write* and *cannot write*, others allow fine-grained classification of all users. Even *Wikis* are a type of content management system, working by an access-control free philosophy but nevertheless greatly simplifying the process of editing and maintaining pages. Wikis do not work with copies of the shared material; the original document is available at all times, always in its latest version.

A few server models, such as *Zope* [8] go even further by making the entire server infrastructure accessible to the users. They can modify not just content but also server behavior by editing actual components/scripts. Security

mechanisms are needed to manage the permissions, such as for accessing a server component (see Figure 2.5).

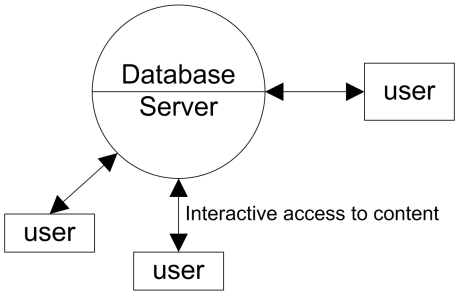


Figure 2.5: Interactive server model with collaborative content. The members of a group can collectively collaborate on the content of a site.

2.1.2 Requirements of Web-Collaborations

In order to implement a web based collaboration, the following requirements must be fulfilled:

Session management. A session is a group of application instances currently working together in a collaborative mode. All applications belonging to the same session exchange information and share behavior. A collaboration server may manage more than one session simultaneously.

User authentication and authorization. Most collaboration systems provide means for user *authentication* and *authorization*. All user data is kept in a database and is used by the main server, which acts as a router of all control and application messages. It is desirable to associate certain application functionality with certain level of *privileges*.

Event logging. One of the important system capabilities is the recording of system activities. Since all system and application messages must go through the main server, all of them can be recorded in a database. This data can be then retrieved later and the whole session activity can be asynchronously reviewed.

Simplicity. A successful collaboration server must be easy for its participants to use and provide all basic functions. All other potential power features should be optional, especially if they require extra software to be installed.

2.1.3 Features of Web-Collaborations

Web based collaborations have great qualities:

Free and easy Access. Th only required thing is only a connection to the internet and a web client such as the Mozilla [9] web-browser in order to access the resources from anywhere. The level of free accessibility depends on the security policy.

Up-to-date versions. With centrally updated or interactive pages, always the latest version is available.

Hyperlinking. The shared material can exist in a rich set of links (to collaborators, to resources, to comments, to older versions, and so on). This requires some sort of maintenance: links should always be up to date.

Content markup. Content markup makes content machine-searchable and more easily to be converted to other media.

2.2 Wiki: a web-based Collaboration Tool

The characteristics of Wiki server software are listed:

Open source solution. Wiki is a open-source technology within the context of existing servers, clients, protocols and standards.

Many implementations. Since the basic concept is simple, there are many implementations in different languages.

Easy to use. The basic Wiki interactivity consists of people dropping by, browsing and reading, editing and adding content.

Easy to set up. Usually Wiki server software is easy to configure and to integrate with other server tools.

Light solution. Compared to the size of its code, a Wiki server contains a surprising amount of functionality and is easy to extend.

2.2.1 Wiki Concept

A *WikiWikiWeb* enables documents to be authored collectively in a simple markup language using a web browser. Because most Wikis are web-based, the term *Wiki* is usually sufficient. A single page in a Wiki is referred to as a *Wiki page*, while the entire body of pages, which are usually highly interconnected, is called *the Wiki*.

Wiki Wiki means *super fast* in the Hawaiian language, and it is the speed of creating and updating pages that is one of the defining aspects of Wiki technology. Generally, there is no prior review before modifications are accepted, and most Wikis are *open* to the general public or at least to all persons who also have access to the Wiki server. In fact, even registration of a user account is seldom required.

Pages and editing. In traditional Wikis, every page has two representations: the form in which it is displayed (usually HTML which is rendered by a web browser) and the form in which it is edited (a simplified markup language, the style and syntax of which varies from Wiki to Wiki).

The reasoning behind this design is that HTML, with its large library of nested tags, is too complicated to allow fast-paced editing, and distracts from the actual content of the pages. It is also viewed as beneficial that users cannot use all the functionality that HTML allows, such as *JavaScript* and *Cascading Style Sheets*, because of the consistency in look and feel that is thereby enforced.

Nevertheless, some recent Wiki engines provide WYSIWYG (*What You See Is What You Get*) editing, usually requiring *ActiveX controls* or *plugins* that translate graphically entered formatting instructions like *bold* and *italics* into HTML tags that are then transparently submitted to the server. In these cases, users who do not have the necessary plugin can only edit the page in its HTML source.

The *formatting instructions* allowed by a Wiki vary considerably depending on the Wiki implementation that is used. Simple Wikis only allow basic text formatting, whereas more complex ones have support for tables, images, formulas, or even interactive elements like games. Because of that, there is now an effort trying to define a *Wiki markup standard* [10].

Linking and creating pages. Wikis are a true hypertext medium, with non-linear navigational structures. Each page typically contains a large number of links to other pages; hierarchical navigation pages often exist in larger Wikis, but do not have to be used. Links are created using a specific syntax, the so-called *link pattern*. Originally, most Wikis use *CamelCase* as a link pattern, produced by capitalizing words in a phrase and removing the spaces between them (the word *CamelCase* is itself an example of CamelCase). While CamelCase makes linking very easy, it also leads to links which are written in a form that deviates from the standard spelling. CamelCase-based Wikis are instantly recognizable from the large number of links with names such as *TableOfContents* and *EseTutorial*. CamelCase has many critics, and Wiki developers looked for alternative solutions. The first Wiki that introduced so called *free links* was *Cliki* [11]. Various Wiki engines use *single brackets*, *curly brackets*, *underscores*, *slashes*, *asterisk* (*e.g.*, SmallWiki) or other characters as a link pattern. Links across different Wiki communities are possible using a special link pattern called *InterWiki* - the idea of having one unified Wiki system distributed across many servers.

Creating a new page in a Wiki is usually done strictly through the same process as linking to it: a link is created on a topically related page; if the link does not exist, it is in some way emphasized as a *broken link*. Following that link opens an editor window, which then allows the user to enter the title and text for the new page. This mechanism ensures that *orphan* pages (which have no links pointing to them) are rarely created, and a generally high level of connectivity is retained.

Recent changes, RSS feeds, and revision history. Wikis generally follow a philosophy of making it easy to fix mistakes instead of making it hard to make them. Thus, while Wikis are very open, they also provide various means to verify the validity of recent additions to the body of pages.

The most prominent one on almost every Wiki is the *recent changes* page. It is simply a list of either a specific number of *recent edits* or a list of *all edits* that have been made within a given timeframe. Some Wikis allow filtering the list to exclude edits that have been marked *minor* or which were made by automatic importing scripts (*bots*).

Some Wikis like SmallWiki provide Rich Site Summary (*RSS*) files, which are based on the Extensible Markup Language (*XML*). RSS is a format for syndicating news and the content of news-like sites. Anything that can be broken down into discrete items can be syndicated via RSS: a changelog of CVS checkins, the revision history of a book and actually the *recent changes* page of a wiki. Once information about each item is in RSS format, an RSS-aware program can check the feed for changes and react to the changes in an appropriate way.

From the *change log*, two other functions are accessible in most Wikis: the *revision* history, which shows previous versions of the page, and the *diff* feature, which can highlight the changes between two revisions. The *revision* history allows opening and saving a previous version of the page and thereby restoring the original content. The *diff* feature can be used to decide whether this is necessary or not: A regular user of the Wiki can view the *diff* of a change listed on the *recent changes* page and, eventually load the history to restore a previous revision.

In case unacceptable edits are missed on the *Recent changes page*, some Wikis provide additional control over content. *Tavi* [12] by Scott Moonen introduced *subscribed changes*, a form of internal bookmarking that is used to generate a list of recent changes to a set of specific pages only.

In extreme cases, many Wikis allow *protecting* pages from being edited. Protected pages on SmallWiki, for example, can only be edited by *authorised* users, which own a special set of privileges. This is generally considered to violate the basic philosophy of WikiWiki and therefore is mostly avoided.

Chapter 3

SmallWiki Relevant Design Aspect

This chapter describes the most important classes of the SmallWiki implementation. The class descriptions are accompanied by UML (*Unified Modeling Language*) [14] diagrams. We describe also the classes of the SmallWiki Extended Security Model, that is introduced in Chapter 5. These classes are marked with footnotes.

The most widely used design patterns used in SmallWiki are the Composite [15] and the Visitor [15] patterns.

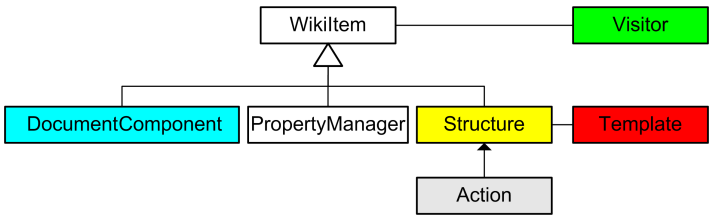


Figure 3.1: **The Core Design.** MVC paradigm: the subclasses of `WikiItem` represent the *model*, the rendering (*view*) is done within different visitors, and the controller is represented by the hierarchy below the `Action` class.

All the classes seen in Figure 3.1 are abstract. Their concrete subclasses will be discussed in the following subsections. The subclasses of `WikiItem` represent the model in the MVC (*Model View Controller*) paradigm and might be visited using subclasses of the `Visitor` hierarchy. As all the rendering is done within different visitors, this part can be seen as the *view*. At last we have the controller, represented by the hierarchy below the `Action` class. Actions are used to do modifications on the model and to start the different visitors to generate the appropriate views.

3.1 Server

The basic serving is done with the chain-of-responsibilities design pattern [15] in the serving protocol. Incoming requests are passed to the first possible candidate that is able to handle it. The request is analysed and processed within this structure and if necessary processed or passed to one of its children (see Figure 3.4).

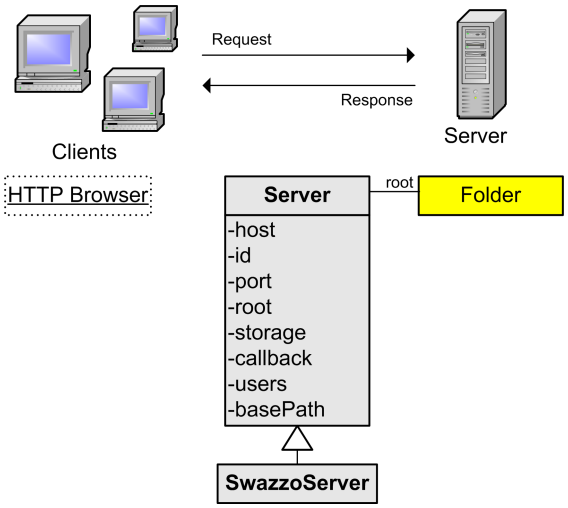


Figure 3.2: Server Setup

The server class has been designed to be subclassed and to provide a common interface to different server implementations (see Figure 3.2). A server might get started using the messages `#start, #startOn:, #startOn:host:ip:` or by simply instantiating using the message `#new`, configuring and starting manually. The server is not a singleton, so there might be multiple instances running within the same image.

```
server := SwazooServer startOn:8080.
```

The instance variable *root* represents the root-entity of the Wiki tree, which is usually a folder. When starting a new server, a default configuration will be created. The write-accessor for the root on a running Wiki should not be called accidentally, as all the subentries will be destroyed immediately without the possibility of going back. In order to have a look at the model of the Wiki, the following expression can be evaluated:

```
server root inspect.
```

The default server has no automatic storage mechanism assigned; this is basically useful when developing for SmallWiki and saving the image manually. When us-

ing the Wiki in a production environment, a working storage-strategy should be assigned and tests should be done extensively [1].

- `server storage:ImageStorage new`
  fast and secure persistence.
- `server storage:nil`
  no persistence.

The responsibility to pass the request to the root node of the Wiki is taken by the server. The exceptions are caught and displayed as a stack-dump on the client side (see Figure 3.3). The link *Open Debugger* can be used to open the debugger in VisualWorks within the context that caused the error and thus investigate the problem further.

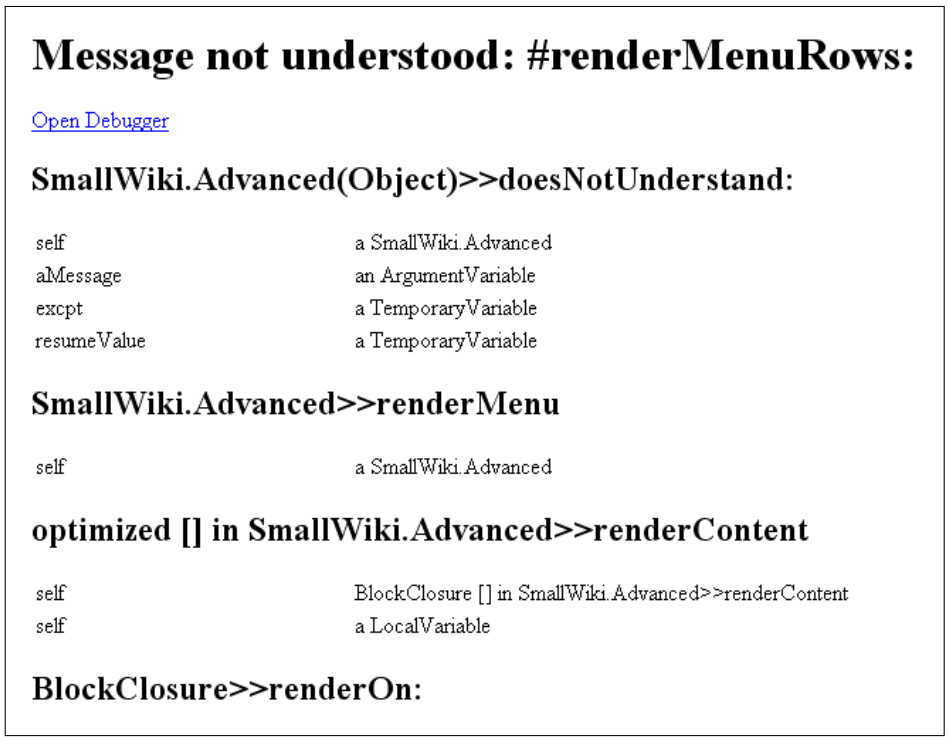


Figure 3.3: **Stack Dump in the Web-Browser.** The exceptions are caught and displayed as a stack-dump on the client side.

[1] If someone develops other storage strategies, he should let us know as we are interested to integrate them into the main-distribution.

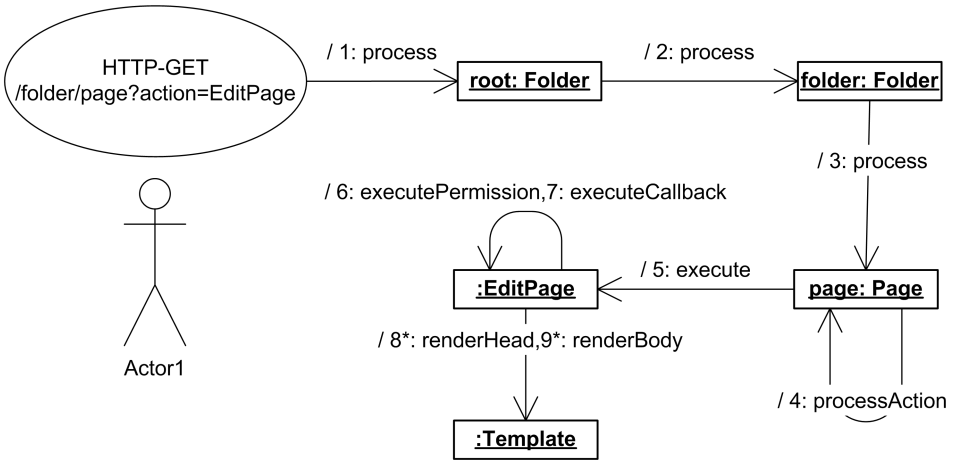


Figure 3.4: **Chain of Responsibility.** Content serving after the user sent the request *http://www.smallwiki.org/folder/page?action=EditPage*.

In order to edit the page *page* contained in the folder *folder* a user enters an URL such as

```
http://www.smallwiki.org/folder/page?action=EditPage
```

Then the following steps, as seen in the collaboration diagram in Figure 3.4, are taken:

1. The web-server gets the request emitted by the client and starts the look up process by delegating it to the root folder.
2. Because the target is not the root structure itself, the request is delegated to the folder called *folder*.
3. As in the previous step, in this folder the request is delegated to the page called *page*.
4. There is no one else that could be interested in this request, it is therefore processed by extracting the parameters and determining the action that should be executed. In this example the class `PageEdit` will be instantiated, initialised, and the message `#execute` will be sent.
5. The action first checks the permissions of the user and evaluates the callbacks. See Section 3.8 for further information.
6. The action asks all the template components to emit their HTML-header and their HTML-body. See Section 3.9 for further information.

3.2 SecurityInformation

The abstract class `SecurityInformation` represents the security-information in the system (see Figure 3.5). Its responsibility is to check if the current user has a certain permission. There is also the possibility to assert the presence of a Permission in the current session. If no such permission is present, an *UnauthorizedError* is thrown and an error page will be rendered instead of the one of the current action.

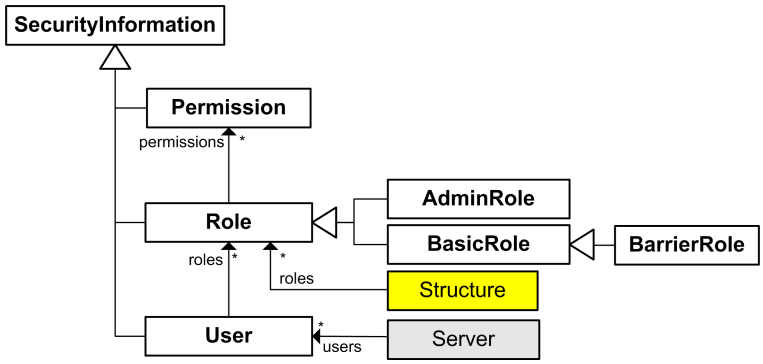


Figure 3.5: **The Security Hierarchy.** A role is a container of a set of permissions. A role can be assigned to a user and also to a structure.

3.3 Permission

A permission represents a privilege in the system and is the basic entity for the permission management. A permission will be granted if it is equal to the given permission. For a list of the permissions available in SmallWiki we refer to Table 4.1.

A permission will be usually used in conjunction with `User>>hasPermission:aPermission` or `User>>assertPermission`.

3.4 Role

Multiple permissions might get assigned to any role. A role can be assigned to a user and can be attached to a structure. The roles on the structures are important in order to change/update the roles of a user that is visiting this structure. A role grants a certain permission if this permission is present in the set of the permission of the role. For a list of the roles available in SmallWiki we refer to Table 4.2.

AdminRole. The AdminRole is used in conjunction with the administrator named *admin* and grants any permission in the system: the message `Role>>hasPermission:aPermission` always returns true.

BasicRole. This role is relevant for all users except for the site administrator named *admin*. A role grants a certain permission if this permission is present in the set of the permission of the role.

BarrierRole. The *BarrierRole*[2] is used to stop adopting permissions from parent structures - this does not apply to any roles of a user, who owns an *admin permission*, *e.g.*, if the BarrierRole owns the *permissions x*, then none of the roles on the same structure will adopt this *permissions x* from their parents' structure roles (see Section 5.1.2).

3.5 User

Multiple roles might be assigned to any user. A certain permission is granted if any of the roles grants this permission.

3.6 Structure

The structure is the basic entity of SmallWiki, representing the model of a single page. A structure is identified by exactly one URL and is usually included in a composite-tree of other structures (see Figure 3.6). The three concrete subclasses of `Structure` are: `Page` and `Resource` as components and the `Folder` as composite. In fact `Structure` should not only be the subclass of `WikiItem`, but also of `Model`. As the visiting aspect, however, is far more important, the messages provided in Model have been copied from this system class.

A structure provides basic navigational accessors to its parents, children and siblings in the Wiki tree. The basic serving is done with the chain-of-responsibilities design-pattern in the serving protocol. The resolving protocol provides messages to look up other structure items using their name.

All the structures have a title, a back-reference to their parent, and might contain user-defined properties, *i.e.*, something like a dictionary containing symbols as keys and any other objects as values. Structures are versioned automatically using a reference pointing to the previous version of the same page. The message `#postCopy` should be overridden to make it work correctly, since some subclasses of `Structure` have a dictionary with objects whereas others do not.

[2] used only in the SmallWiki Extended Security Model

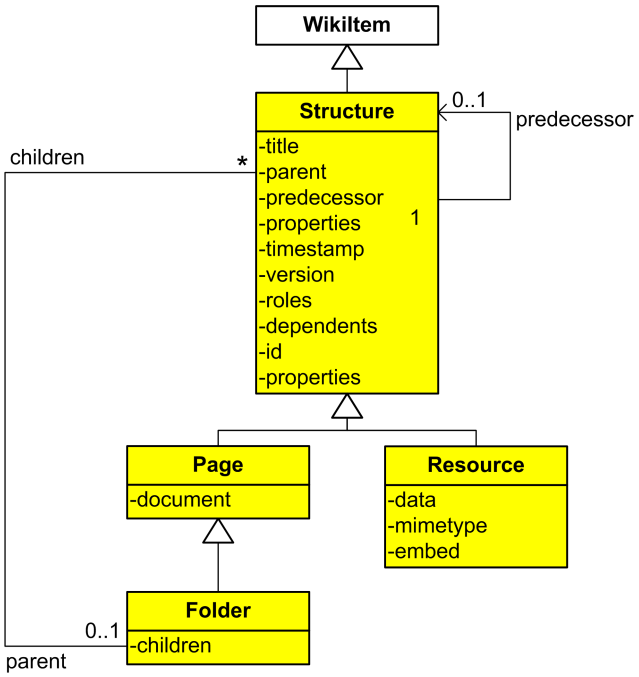


Figure 3.6: Structure Composite

Resource. A resource might contain any data, like images, videos, sounds, pdf or zip files. In fact it can be anything that someone wants to include with the pages, or wants to provide as a possibility to download.

The MIME-TYPE (*Multipurpose Internet Mail Extensions Type*, a specification for formatting non-ASCII data so that it can be sent over the Internet) of the data is used to determine how the given resource should be rendered. As an example images and videos should be displayed inside the html document, whereas zip-files are only references as a link to allow the user to download the file.

Page. A page is the most important and probably the most used class of the Structure hierarchy. As a sole entity it contains a composite of documents modeling the contents of the page that the user entered using the Wiki syntax. When initializing the instance a default document will be created to make the user aware of the newly created page.

Folder. The folder groups a number of children. `Folder` is a subclass of `Page`, therefore it also contains a document that might be used to describe the contents.

3.7 Document

The document hierarchy describes the content of a Wiki page. It includes all the basic elements to represent a text such as paragraph, table, list, links, *etc.* (see Figure 3.7).

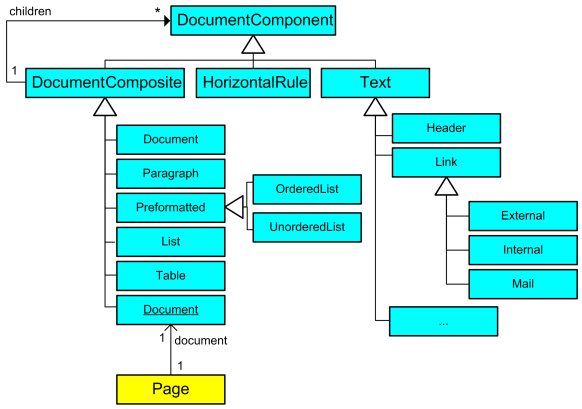


Figure 3.7: The Document Hierarchy

General notes.

- All the subclasses of `DocumentComponent` must overwrite the message `postCopy` to make a deep copy of all its content.
- All concrete subclasses of `DocumentComponent` should override the message `accept:aVisitor` to visit this instance.

Remarks concerning Links.

- `MailToLink` holds a string with the e-mail address.
- `ExternalLink` holds a string with the URL. External links are also able to point to internal resources, but they do not update when the target is renamed or moved.
- `InternalLink` holds a reference to a structure and updates automatically title and reference when the target is renamed or moved. An internal link might point to a non existing structure and will be created automatically when being accessed.

When the user enters a text using the Wiki syntax, the text is parsed using *SmaCC* [16] and the abstract syntax tree is stored within the page.

As shown on Table B.1, the syntax of SmallWiki is similar to *SqueakWiki* [17] or *WikiWorks* [18]. Changing the grammar of the parser is no big deal, if somebody is more familiar with a different one and wants to support that. However, as for all other parsers, it is difficult to write extensions that can be added and removed independently in order to parse new document entities.

3.8 Action

Actions are instantiated by a structure and they are initialized with that structure and the current request using the constructor method `#request:structure:`. Actions have basically two tasks: first to perform the action itself, and second to initiate or to render the GUI. Actions represent also the context in which a page is rendered as they know about their structure, the request, the response, and the security status (see Figure 3.8).

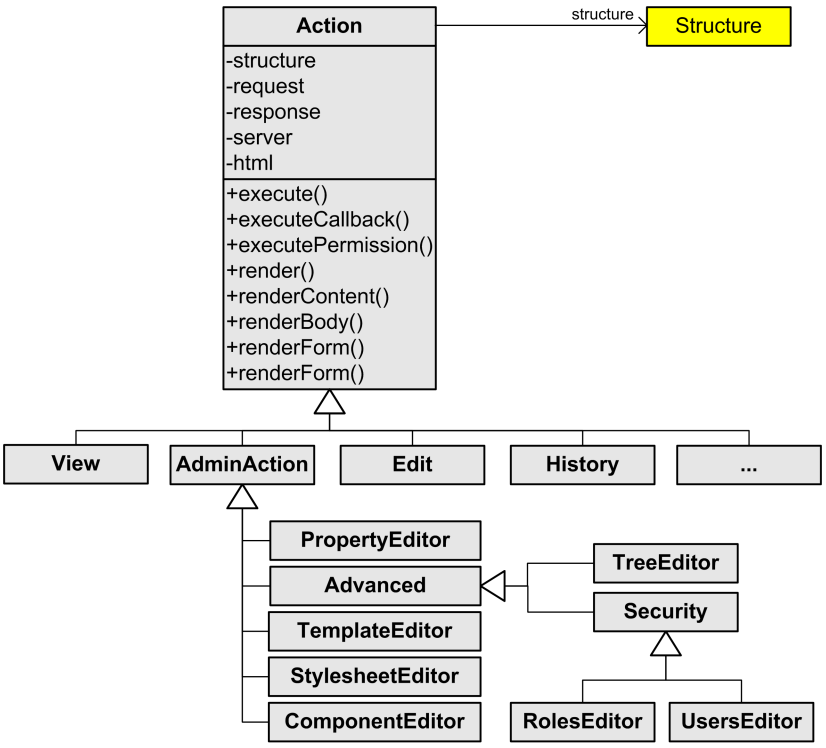


Figure 3.8: The Action Hierarchy

The *Action* and the *Rendering* protocols are described as follows:

- **Action Protocol.** This part of the action is used to handle the requests. The message `#execute` is called by the structure after initializing the required

instance variables. It checks the security permissions of the current user, evaluates the callbacks and starts the rendering by calling `#render` on itself. The running action might use the accessors to manipulate and mediate with the current environment. It is usually not necessary to override the message `#execute`, the callback mechanism described in Section 3.11 can be used instead.

- **Rendering Protocol.** The rendering process is started from the message `#render` at the end of `#execute`. The message `#render` fetches the collection of templates of the associated structure and starts generating the XHTML output. Therefore the document is parsed (see Figure 3.9). It asks each template to render the content they want to emit into the <head> ... </head> part of the output. Afterwards the body part <body>...</body> is generated and again every template might contribute its content into that part. As explained in Section 3.9, there is always an instance of the class `TemplateBodyContent` available calling the message `#renderContent` of the action: by overriding this message the user-interface is rendered by the new action. The state of any component inside the rendering protocol should not be changed, as an action is unable to know when and how many times it is actually called.

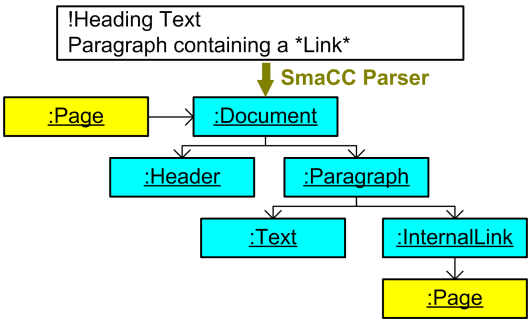


Figure 3.9: Parsing a Wiki Document

Advanced. The class `Advanced`[3] is a subclass of the `Admin` class. It provides the basic methods for the graphical user interface of the *security management* (see class description in Paragraph 3.8, Figure C.1 and Figure C.9) and of the *tree management* (see class description in Paragraph 3.8).

[3] used only in the SmallWiki Extended Security Model

Security. The class `Security`[4] is a subclass of the `Advanced` class. It is the superclass of `RolesEditor` class and `UsersEditor` class. It provides methods for rendering form buttons and methods for creating strings with information about roles details. These strings will be used to render the details of the roles via Javascript as popup or as html layer.

UsersEditor. The class `UsersEditor`[5] is a subclass of the class `Security` and is responsible for rendering the user administrating interface and rendering the overview of all users and their updated roles on a certain structure. It is also responsible to execute the corresponding actions.

The *renderForm* and the *renderInformation* protocols are described as follows:

- **renderForm.** In this protocol, there are methods that decide on which users the current *administrator* can execute admin methods, *e.g.*, an administrator is only allowed to manage the users that it has created himself, unless it is the *main administrator* that can manage any user. The method `renderForm` will generate three forms:
  1. **Policy form.** Form where one or several roles can be assigned to the users.
  2. **Creation form.** Form to create a new user or to change the password of a user.
  3. **Deletion form.** Form to delete a user by selecting the corresponding checkbox.
- **renderInformation.** This protocol contains the methods that are responsible for rendering the overview of all users and their updated roles with the permissions on a certain structure.

RolesEditor. The class `RolesEditor`[6] is also a subclass of the class `Security` and is responsible for rendering the roles administrating interface and for executing the corresponding actions.

The *renderForm* and the *renderInformation* protocols are described as follows:

- **renderForm.** As in the class `UsersEditor`, the methods in this protocol decide on which roles the current *administrator* can execute admin methods, *e.g.*, an administrator is only allowed to manage the roles and permissions that it has created itself or owns itself, unless it is the *site administrator* that can manage any role and permission.

[4] used only in the SmallWiki Extended Security Model
[5] used only in the SmallWiki Extended Security Model
[6] used only in the SmallWiki Extended Security Model

- **renderInformation.** The methods of this protocol are responsible for rendering the updating roles according to the structure on which the roles appear. The permissions of the roles are accessible via Javascript by clicking on either the *popup icon* or on the *layer icon* that appear next to the role name.

TreeEditor. The class `TreeEditor`[7] provides methods to *cut*, *copy*, and *delete* nodes out of the Wiki tree. It still lacks the support of re-arranging the links when a structure has been moved to another place. Also the roles are still copied with the structure. In a future release, the roles should be deleted before a structure is copied. The cut, copy, and delete functions are only provided for the *site administrator* named *admin*. All other administrators can only use the `treeEditor` to browse the Wiki tree.

3.9 Template

Templates are used to render common parts of Wiki pages. They are defined within a collection held in the root of the Wiki and in combination with a selected Stylesheet (see Figure 3.10), they provide the look-and-feel of the Wiki. As the templates are held in the property manager of the structure, they are shared within all children of a folder unless there is a new definition.

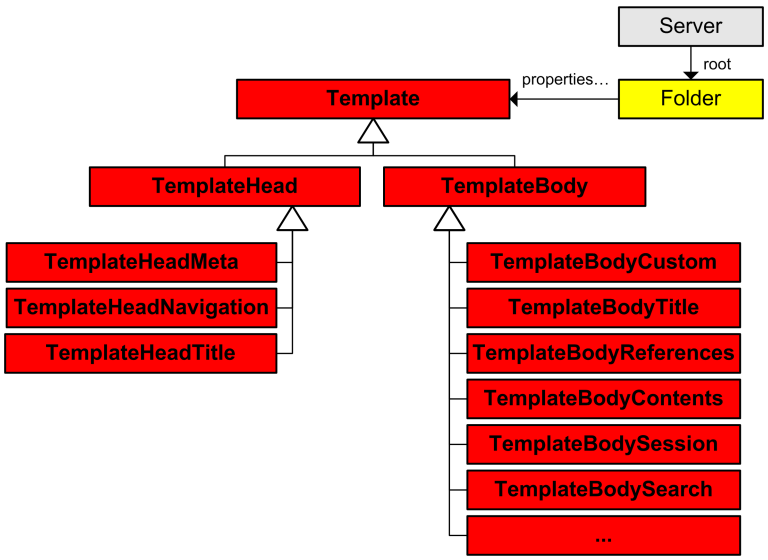


Figure 3.10: The Template Hierarchy

[7] used only in the SmallWiki Extended Security Model

TemplateHead. The class `TemplateHead` is a subclass of `Template`. It should be used for templates rendering the header of the output-file. If someone wants to render to the head and to the body, he should use `TemplateBody` as a superclass instead.

TemplateBody. The class `TemplateBody` is a subclass of `Template`. It should be subclassed in most of the cases to create a new template component. The message `#title` on the class-side should be implemented in order to return a string describing this subclass. The title is also used by default to identify the associated CSS-id and to render it into the body part. The messages `#defaultId` and `#defaultTitle` might be used to change this behavior. The user is always able to edit the id and title from within the template-editor (see Figure 3.11) in the web-browser to customize the template to his needs and to the applied stylesheet. Other adjustments can be made via the property-editor (see Figure 3.12).

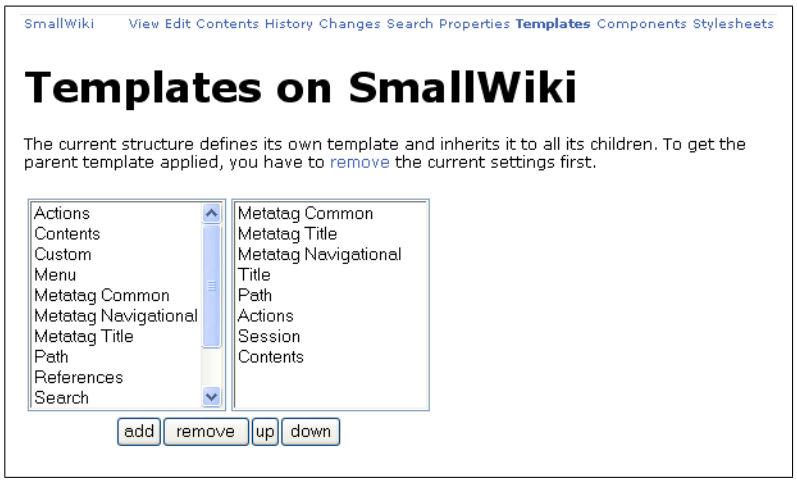


Figure 3.11: The Template Editor

Figure 3.12: The Property Editor

3.10 Storage

The abstract storage class provides a protocol to all kinds of storage mechanism implementing persistence in a Wiki. It takes care of the notification of changes. Subclasses should implement either the message `#changed` or `#changed:` to make the given structure persistent.

SnapshotStorage. The class `SnapshotStorage` provides an interface to make snapshots of Wikis on a regular bases. With the implementation of the `ImageStorage` as a concrete implementation this is the most secure and most widely used storage mechanism (see Figure 3.13).

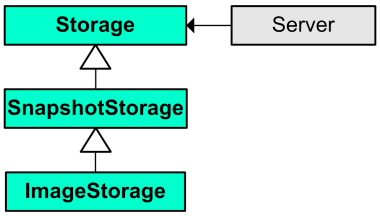


Figure 3.13: The Snapshot Hierarchy

3.11 HTML and Callbacks

Creating valid XHTML is an error prone task when using string concatenation. SmallWiki follows the design of Seaside [19] and implements the class `HtmlWriteStream`. This class subclasses `WriteStream` and provides a lot of additional messages to append text and XHTML elements to the document being rendered.

The following example could be part of the message `#renderContent` within the Action hierarchy to render a simple user-interface:

```
html heading:'Title' level:1.
html paragraph:[
  html text:'Click '.
  html
    anchor:'here'
    to:self url
    callback:[ :action | action doSomething ] ]
```

The code produces the following output:

```
<h1>Title<h1>
 <p>
  Click
  <a href="/?action=ActionClass&callback=31415">
   here
  </a>
 </p>
```

The first message `#heading:level:` produces a simple section heading of level 1. The message `#paragraph:` is similar to the one of the heading, but in this example instead of passing a string we pass a block: everything done within that block will be put inside the paragraph tags. This mechanism assures that all tags are closed properly and that always valid XHTML is generated. The message `#text:` escapes the given string to make sure the code can be displayed correctly within the web-browser.

The message `#anchor:to:callback:` is used to generate an anchor with an assigned callback block. The `anchor:` argument obviously renders the things that should be rendered as the content of the link. The `to:` argument specifies the place where the callback should be handled: usually this is within the same action, but occasionally someone might need to specify something else. The `callback:` argument is evaluated when clicking the link. As an argument the block receives the action that is executing the callbacks, note that this is not necessarily the same action that rendered the link and that is referenced using the keyword `self`.

`HtmlWriteStream` does not emit any unnecessary spaces into the output stream, which makes investigation in the HTML code somehow difficult. For this purpose, there is the possibility to enable the included pretty-printer with the method

```
HtmlWriteStream prettyPrint:true
```

The pretty-printer slows down the rendering process and might have unwanted effects on the output in the web-browser.

More advanced examples about html-rendering and callbacks can be seen in the Action-class `CallbackDemo`, that is part of the examples-bundle (see Section B.1). The user interface of this class is accessible by the url `http://localhost:8080/?action=CallbackDemo`.

Chapter 4

Security in Open Collaborations

In this chapter we discuss risk, threat, and vulnerability of a web collaboration. User authentication and authorization are explained, and it is shown how security is guaranteed in *SmallWiki*. Therefore we present the **SmallWiki Default Security Model** - an easy and powerful system. There is the lack of reliability for sites with several administrators that partially manage common parts of the Wiki site (see Section 4.4.3).

Web-based collaboration security refers to efforts to create a secure collaboration platform, designed so that users are not able to perform actions that they are not allowed to perform, but can perform the actions that they are allowed to. This involves specifying and implementing a *security policy*.

4.1 Risk, Threat and Vulnerability

This section presents a general overview of the risk and the threat of a web collaboration. This is the basis for analysing the collaboration vulnerability and for improving the security of the web-based business environment [20]. We point out that a security system is only as strong as its weakest link.

Risk. The *risk* is that an intruder may be successful in attempting to access the resources of a collaboration that should not be accessible. There are many possible effects of such an occurrence. In general, the possibility exists for someone to:

- **Read access.** Read or copy information.
- **Write access.** Write, attack, or destroy data.
- **Denial of service.** Deny normal use of a web resources by consuming all of its bandwidth, CPU power, or memory.

Threat. The *threat* is anyone with the motivation to attempt to gain unauthorized access to the resources or anyone with authorized access. Therefore it is possible that the threat can be anyone. The *vulnerability* to the threat depends on several factors such as:

- **Motivation.** How useful access to or destruction of resources might be to someone.
- **Trust.** How well we can trust the authorized users and/or how well trained are these users to understand what is an acceptable use of the resources.

Vulnerability. *Vulnerability* is essentially a definition of how well protected the web-collaboration is from someone outside of the collaboration that attempts to gain access to it, and how well protected the resources are from a user within the collaboration intentionally or accidentally giving away access or otherwise damaging the resources. Most of the Wiki web-collaborations give *free access* to their resources. Mostly it is relatively easy to fix the damage made by an intruder when some sort of history/backup tools are provided by the web-based system. However this takes a lot of time and it is very annoying. Therefore it is useful to guarantee restricted access to certain resources.

4.2 User Authentication and Authorization

When a web-server gets a request, it first has to find out which user is interacting with it. After that, the server is able to decide if the user is allowed to perform a certain action. This is called *user authentication* and *user authorization*.

Authentication. *Authentication* answers the question: *who is that user*? It ensures that a valid user is logged in, based on an ID and password provided by the user. This requires some sort of user database on the server site, where at least the user names and their corresponding passwords are stored.

There is no correspondence between user names and passwords of specific operating systems (*e.g.*, unix systems with /etc/passwd file) and user names and passwords in the authentication schemes we are discussing for use in the Wikis. Web-based authentication can use similar password files, but a user never needs to have an actual account on a given operating system in order to be validated for access to the resources being served from that system and protected with HTTP-based authentication.

Authorization. *Authorization* answers the question: *is that user allowed to do this or that*? It is the process of determining whether a user is allowed to perform

a requested action.

After username and password are authenticated, the authorization is based on the user's privileges. Many systems assign one or several roles to a user. A role is a collection of privileges that dictate the type of system access the user has. If the user is not authorized for a certain action, the related item or button (*e.g.*, a button in order to *edit* content) on the user interface should be hidden or disabled at optimal behavior.

To recapitulate we can say that *authentication* opens the front door, *authorization* then tells the user what rooms they can go into once they get inside.

4.3 SmallWiki Security Model

The security system of SmallWiki is based on *users*, *roles* and *permissions*. The server is responsible for *authenticating* the user and for deciding whether the user is allowed to perform a certain action. To understand this authorization process, we explain the default permissions and roles and show how the server gets the relevant set of permissions of the user who sends the request to the Wiki server. For this the server has to update the roles of the user, since they might change through the Wiki site.

Secure Transmission. The authentication process of SmallWiki passes login information over the wire in an easily decryptable way. The user names and passwords are stored in plaintext in *cookies* that are always sent with the requests of the users. To prevent that someone is *snooping* the username/password combinations or to manage the site more securely, there is the possibility to use a Secured Sockets Layer (*SSL*) [22] connection. We can use *SSL* just to manage the site when logging in as the *site administrator* that always owns all permissions, or use *SSL* for the complete traffic. The easiest way to put this into effect is to use *Apache* [21] or another webserver which comes with SSL support and put it in between the user and the SmallWiki server. Another way to improve the security is by implementing an encryption mechanism for the username/password pair being stored in the cookie.

Permissions. A SmallWiki structure is either a *Folder*, a *Page* or a *Resource*. A structure has permissions which describe what can be done with it such as *view*, *edit* or *add*. These permissions are named according to the kind of structure the permissions are applied to. Therefore they are called *Folder View*, *Folder Edit* or *Folder Add*. The permissions *Folder Admin*, *Page Admin* and *Resource Admin* are *admin permissions*. Permissions can be also assigned to users via roles. These permissions are the individual actions a user is allowed to perform in the system. As default, there are SmallWiki permissions, as described in Table 4.1.

Privilege Type	Description
Add	Create and add a child to this item. (child=[Page|Folder|Resource])
Admin	Administrate the current structure. (administrate the properties, templates, components, stylesheets, and security of the current structure)
Code	Write and evaluate smalltalk code on current structure. (very critical, *e.g.*, can theoretically add all permissions!)
Copy	Copy any child of the current structure.
Edit	Edit the current structure.
History	View the history of current structure.
Move	Move the children of current structure.
Remove	Delete any child of the current structure.
Template	Modify the template.
View	Read. Can view the current structure.

Table 4.1: The set of SmallWiki default permissions.

Roles. A role is a container of the permissions. One or several roles can be assigned to a Wiki user. That is how a user gets a set of permissions. Roles can also be attached to a Wiki structure. The roles on the structures are important in order to change/update the roles of a user that is visiting this structure. At the first occurrence in the Wiki tree - starting at the root - a role is called *top role*. A *child role* is a role that is already defined at a higher level in the Wiki tree. A role that owns an admin permission is called an *admin role*. As default, there are two roles defined in SmallWiki, as described in Table 4.2.

Default Roles	Description
administrator	Can do anything (admin, view, edit, delete etc.).
anonymous	Can view, edit and add pages (view, edit, add).

Table 4.2: The set of SmallWiki default roles.

SmallWiki Authentication. When a user first sends a request to the Wiki server, the Wiki server considers him as the default *anonymous* user. The *anonymous* user additionally possesses the *anonymous* role. If it tries to access a *protected* resource - a resource where the anonymous user lacks the permission to perform the requested action - the Wiki server denies the request and shows an *access denied page* (see Figure 4.1). On the next screen the server asks the user to log in by presenting an *authentication form* (see Figure 4.2). Once this form has been filled out and submitted, the Wiki server looks for the user account in order to perform the user *authentication*.

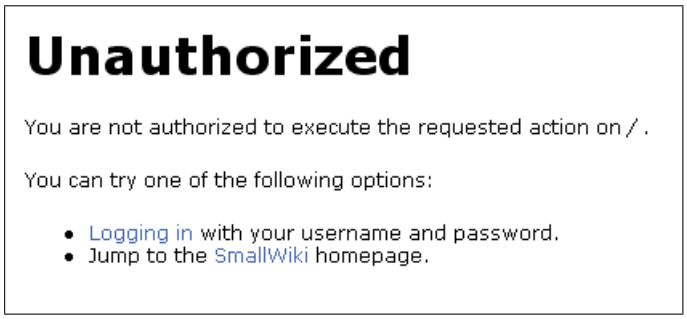


Figure 4.1: The access denied page.

Figure 4.2: The login screen.

4.4 Authorization in the SmallWiki Default Security Model

After the server has *authenticated* the username and password, the *authorization* is based on the privileges that a user has. These permissions are stored in roles. A Wiki user usually owns one or several roles. If he does not own any role he does not have any permissions at all. A role does not have to be static, since its set of permissions may change through the Wiki site, *e.g.*, a role has the permission set *p1* on folder *a*, and has a different permission set *p2* on folder *b*.

We explain in context of the SmallWiki Default Security Model how the server gets the actual set of permissions that a user owns at a specific structure in the Wiki site.

To illustrate this mechanism we use three simple examples as follows:

1. **Inheritance. A simple mechanism for updating roles.** We show in this example how the roles are inherited through the Wiki tree.

2. **Redefinition. Modification of roles by redefinition.** We explain how the mechanism of inheritance is combined with redefinition of roles.

3. **Problems. Redefining the roles of administrators.** Problems that appear in context with several administrators for the same Wiki part are illustrated.

These examples are structured as follows:

- **Intention.** We explain the purpose of the example.
- **Setup.** We illustrate the setup of the Wiki tree, the roles, and of the users.
- **Activities after the setup**[1]**.** We describe the actions of an administrator.
- **Result.** The consequences of the scenario are described. We might also focus on the problems.

The users, permissions, folders, and roles of these examples are always created by the site administrator. The labeling is done as follows:

- Users are labeled *ducasse* and *sally*.
- Permissions are labeled with *Folder Add*, *Folder View*, *Page View*, *etc.*.
- Folders are labeled with *a* and *b*.
- Roles are labeled *r1*, *teacher*, and *secretary*.

The blue colored permissions in the Figures used in these examples indicate roles that are defined locally, the yellow color marks roles that receive their permissions by inheritance. These color settings depend on the included style-sheet file (see Section B.6.1). That is why there are also icons used to illustrate this characteristic:

- Icon '+'. This icon indicate roles that are defined locally.
- Icon '||'. This icon marks roles that receive their permissions by inheritance.

[1] used only for the last example.

4.4.1 Inheritance. A Simple Mechanism for Updating Roles.

Intention. Security policies are set up by the site administrator in the root of the Wiki tree, and they are valid through the entire Wiki site since the policies are inherited to the children of the root structure. We illustrate in this example the inheritance mechanism for roles in its simplest way.

Setup. Wiki site with two folders and one role (see Figure 4.3):

1. Wiki site with folders *a* and *b*. *b* is sub-folder of *a*.
2. Role *r1* is attached on folder *a* with permissions *Folder Add* and *Folder View*.
3. The role *r1* is assigned to a user.

Figure 4.3: **Setup to illustrate inheritance of roles.** Wiki tree with two folders *a* and *b*. The role *r1* is attached on folder *a* with permissions *Folder Add* and *Folder View*.

Result. SmallWiki structures - and their attached roles - *inherit* their security policies from their containers. That is why it makes a given security policy easier to maintain.

- On the folder *a* the user with the role *r1* owns the permissions *Folder Add* and *Folder View* of the role *r1*.
- On the folder *b* the user owns the same permissions as on folder *a*, since its role *r1* gets these permissions by *inheritance* (see Figure 4.4).

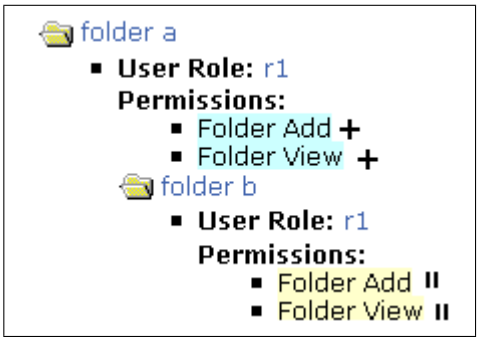


Figure 4.4: **Result to show inheritance of roles.** The roles of subfolder *b* are inherited from folder *a*. The blue colored permissions with the icon '+' indicate roles that are defined locally, the yellow color and the icon '||' mark roles that receive their permissions by inheritance.

4.4.2 Redefinition. Modification of Roles by Redefinition.

Intention. The roles defined in the root folder can be modified by the administrator on a certain subfolder in order to adjust the security settings. The administrator is able to redefine every role on a folder, and these settings are valid in its subfolders as long as they are not modified again. To illustrate this, the previous example (see Section 4.4.1) is extended.

Setup. The folder *c* is added as a sub-folder of folder *b*. On the folder *b* the role *r1* is redefined with the permissions *Page Add* and *Page View*. These new permissions are colored blue and marked with the '+' icon.

This scenario is set up as follows (see Figure 4.5):

1. Wiki site with three folders *a*, *b* and *c*. *c* is sub-folder of *b*, *b* is sub-folder of *a*.
2. Role *r1* is attached on folder *a* with permissions *Folder Add* and *Folder View*.
3. Role *r1* is attached on folder *b* with the permissions *Page Add* and *Page View*.
4. User with the role *r1*.

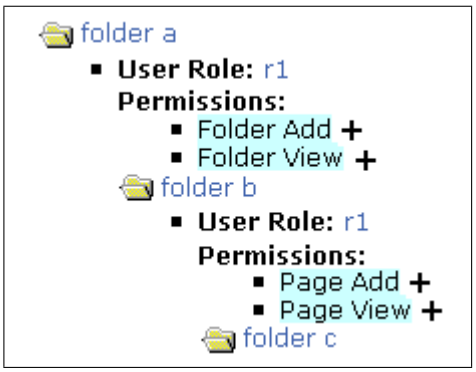


Figure 4.5: **Setup to illustrate redefinition of role *r1*.** The role *r1* is redefined on folder *b*.

Result. As illustrated in Figure 4.6, the user with the role *r1* visiting folder *a* owns the two permissions *Folder Add* and *Folder View*. On folder *b* it owns the permissions *Page Add* and *Page View*, since its role *r1* has been redefined. Finally when the user is visiting folder *c* it still owns the permissions *Page Add* and *Page View*, because the role *r1* on folder *c* inherit its permissions from the role *r1* of folder *b*. This mechanism makes it possible for the administrator to adjust the security settings for any subtree of the Wiki.

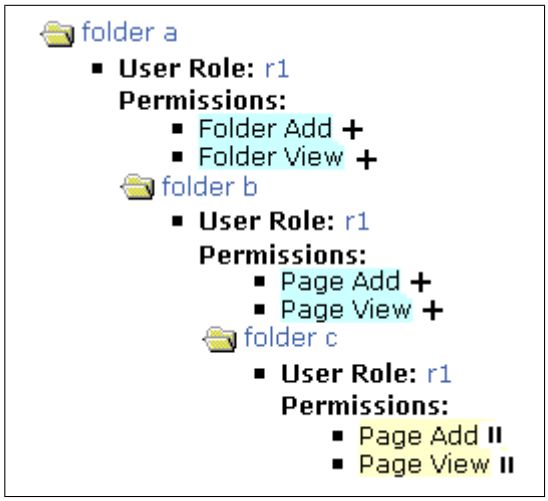


Figure 4.6: **Result to show the consequences of redefinition of the role *r1*.** The role *r1* is redefined on folder *b*. Therefore the role *r1* of folder *c* owns the permissions form role *r1* that is redefined on folder *b*.

4.4.3 Problems. Redefining the Roles of Administrators.

Intention. A Wiki site can contain more than one user with an admin role (see Section 4.3). The updating mechanism by inheritance applies to all roles, including the roles that own an *admin permission* (see Section 4.3), and every administrator can redefine all roles. This might have unwanted consequences for the administrators, *e.g.*, they could be able to remove permissions from each other or could even assign more permissions to themselves than they already own. This weakness of the SmallWiki Default Security Model is illustrated in the following example.

Setup. The initial configuration is shown in Figure 4.7.

- Wiki site with two folders *a* and *b*. *b* is sub-folder of *a*.
- The role *teacher* with the admin permissions *Page Admin*, *Folder Admin* and *Resource Admin* is added to the folder *a*.
- The role *secretary* with the admin permissions *Folder Admin* and *Page Admin* is added to folder *a*.
- The users *sally* and *ducasse* are created.
- The role *teacher* is assigned to the user *ducasse*.
- The role *secretary* is assigned to the user *sally*.

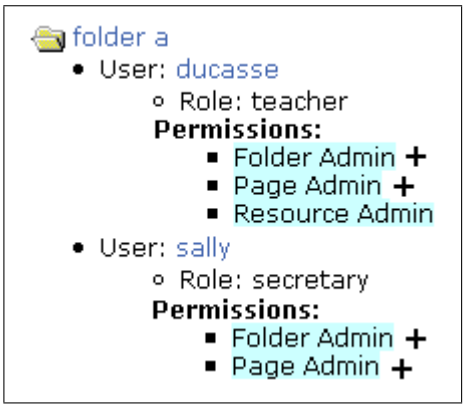


Figure 4.7: **The setup of roles and users to illustrate unwanted side-effects.** The users *sally* and *ducasse* own admin permissions on folder *a*.

Activities after the setup. Redefinitions by user sally: since the user named *sally* owns the admin permission *Folder Admin* also on folder *b*, it is able to remove the permissions *Resource Admin*, *Folder Admin* and *Page Admin* from the role *teacher*. It can even add new permissions such as *Folder History* and *Folder Code* to the role *secretary*. This is done by *redefinition* of these roles on folder *b*. The consequences of the activities of *sally* are displayed in Figure 4.8.

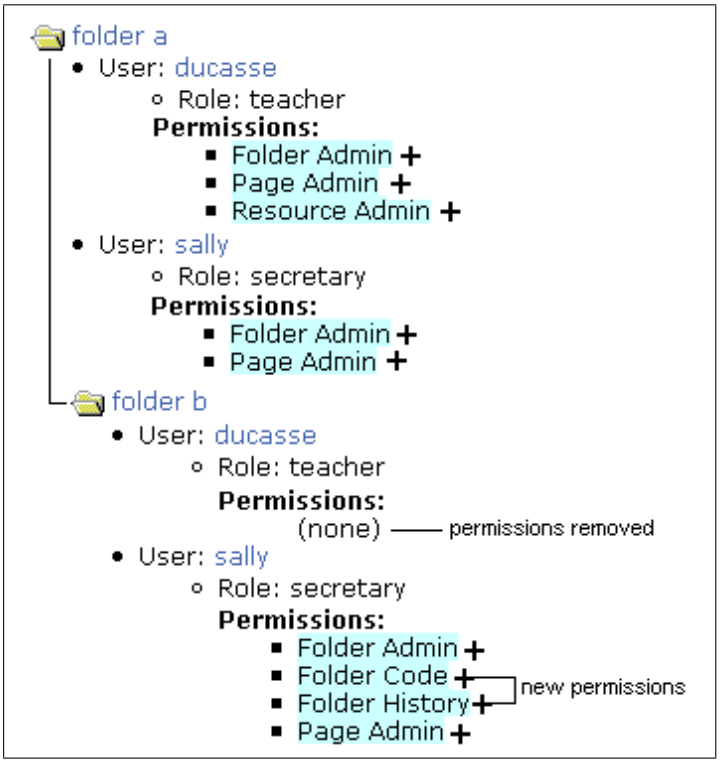


Figure 4.8: **The consequences of the *malicious* acts of the administrator *sally*.** On the folder *b*, the administrator *sally* removed all permissions from user *ducasse*, and gave itself additionally permissions.

Result and problem. When we analyse the permissions of the two users currently visiting folder *b*, we face the facts that the user *ducasse* with the role *teacher* really lost all its privileges, whereas the user *sally* with the role *secretary* gained the permissions *Folder History* and *Folder Code*. This scenario displays a weak point of the SmallWiki Default Security Model: on the one hand the administrators are too powerful, since they are able to extend their own privileges and curtail roles of another administrator. On the other hand their roles are not guarded from such malicious acts by other administrators. Additionally an administrator is even allowed to delete a user. The SmallWiki Extended Security Model introduced in Section 5.1 focuses on these drawbacks and provides a solution.

Chapter 5

SmallWiki Extended Security Model

The SmallWiki Default Security Model described in Chapter 4 is powerful enough for Wiki sites that are run by only one administrator. If there is more than one administrator, they are able to abuse their privileges as seen in Section 4.4.3. Therefore we introduce a model that fulfils the pattern of safe delegation (see Section 5.1.4) with several sector administrators. This model is called SmallWiki Extended Security Model.

The principles of the SmallWiki Extended Security Model are explained and illustrated in this chapter.

5.1 Authorization in the SmallWiki Extended Security Model

The default permissions, roles, and users are the same as in the SmallWiki Default Security Model. There are alterations in the SmallWiki Extended Security Model:

- The security policies are distributed on the Wiki tree by acquisition (see Section 5.1.1) instead by inheritance.
- As described in Section 5.1.4, the administrators are not able to:
  - change their own roles.
  - delete any user and any role they want to.
  - manage permissions that they do not own.

We explain how the server gets the actual set of permissions that a user owns at a specific structure in the Wiki site.

To illustrate this mechanism we use four examples as follows:

1. **Acquisition. A mechanism for updating roles by acquisition.** We show how the security policies are distributed by acquisition on the Wiki site.
2. **Acquisition. Curtailing roles with the Barrier role.** This example illustrates how the Barrier role is used to stop *acquisition* for an indicated set of permissions at a certain point on the Wiki site.
3. **Acquisition. Updating the roles of an administrator.** An important rule concerning the Barrier role and the administrators is introduced here.
4. **Delegation. Providing safe delegation through *security rules*.** We show in this example, that an sector administrator can not modify or delete any roles and any users it wants to. Therefore are introduced rules that restrict the power of the sector administrators.

The examples of this chapter are structured as follows:

1. **Intention.** We explain the purpose of the example.
2. **Setup.** We illustrate the setup of the Wiki tree, the roles, and of the users.
3. **Result.** Finally the consequences of the scenario are described.
4. **Explanation.** We give further helpful commentaries.

The blue colored permissions in the Figures used in these examples indicate permissions that are added by locally defined roles. The yellow color marks permissions that are acquired from parent roles. The permissions of the Barrier role are highlighted with red color. These color settings depend on the included style-sheet file (see Section B.6.1). That is why there are also icons used to illustrate this characteristic:

- Icon '+'. This icon indicates permissions that are added by a locally defined role.
- Icon '||'. This icon marks permissions that are acquired from parent roles.
- Icon '−'. This icon highlights permissions of the Barrier role.

5.1.1 Acquisition. A Mechanism for Updating Roles by Acquisition.

Intention. We show how the security settings are distributed by acquisition on the subtrees of a Wiki site. Therefore three folders and a role on the root folder are created.

Setup. Wiki three with three folders and one role (see Figure 5.1). Everything is created by the administrator.

1. Wiki site with three folders *a*, *b* and *c*. *c* is sub-folder of *b*, *b* is sub-folder of *a*.
2. Role *r1* is attached on folder *a* with permissions *Folder Add* and *Folder View*.
3. Role *r1* is also attached on folder *b* with the permissions *Page Add* and *Page View*.
4. The role *r1* is assigned to a user.

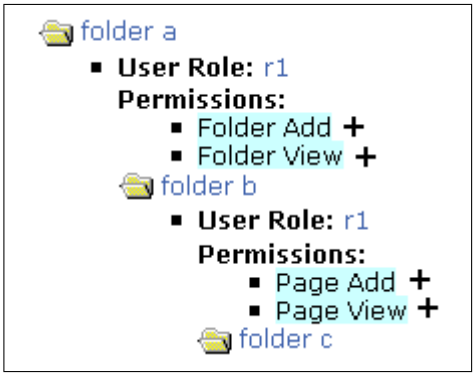


Figure 5.1: **Setup to illustrate the mechanism for acquisition.** Wiki tree with three folders *a*, *b*, and *c*. The role *r1* is defined on folder *a*, and also on subfolder *b*.

Result. Instead of overriding the role *r1*, its permissions are *added* to the parent role (see Figure 5.2).

- On the folder *a* the user owns the permissions *Folder Add* and *Folder View* of the role *r1*.
- On the folder *b*, the user owns the permissions of role *r1* from folder *a* **and** the permissions of role *r1* from folder *b*. So it owns the permissions *Folder Add*, *Folder View*, *Page Add* and *Page View*.

- On the folder *c* the user owns the same permissions as on folder *b*.

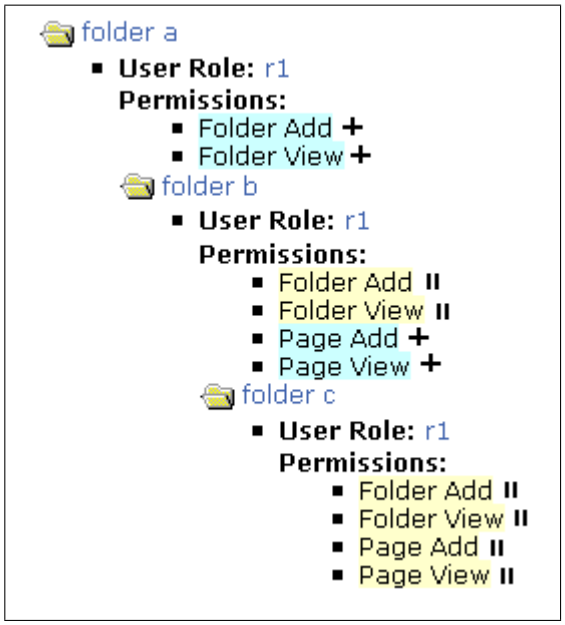


Figure 5.2: **Result of acquisition.** Instead of overriding the role, its permissions are *added* to the previously defined one.

Explanation of acquisition. The role *r1* on folder *b* does not just own the permissions that are defined there. It owns the permissions from the role *r1* of folder *b* and also the permissions of role *r1* of folder *a*.

If we apply the SmallWiki Default Security Model to this Wiki setup, the role *r1* on folder *b* would only own the permissions that are re-defined on the role *r1* of the folder *b*. This is the major difference in updating roles between the two security models. This new mechanism for updating roles is called *acquisition*. It is also a kind of *inheritance*, but instead of overriding the top role, its properties are added to the child role.

SmallWiki structures - and their attached roles - *acquire* their security policies from their containers. That is why it makes a given security policy easier to maintain.

5.1.2 Acquisition. Curtailing Roles with the Barrier role.

Intention. SmallWiki also uses a special kind of role, the *Barrier role* . This role is used to stop *acquisition* for an indicated set of permissions at a certain point on the Wiki site. If there would not be a mechanism for stopping acquisition, the set of permissions of a role would only grow or stay unmodified and never scale down.

Setup. We extend the example seen in Section 5.1.1 (see Figure 5.3):

- Setup as on previous example.
- The folder *d* is added as a sub-folder of folder *c*.
- The *Barrier role* is used on the folder *d* in order to stop acquisition for the permissions *Folder Add* and *Page Add*.

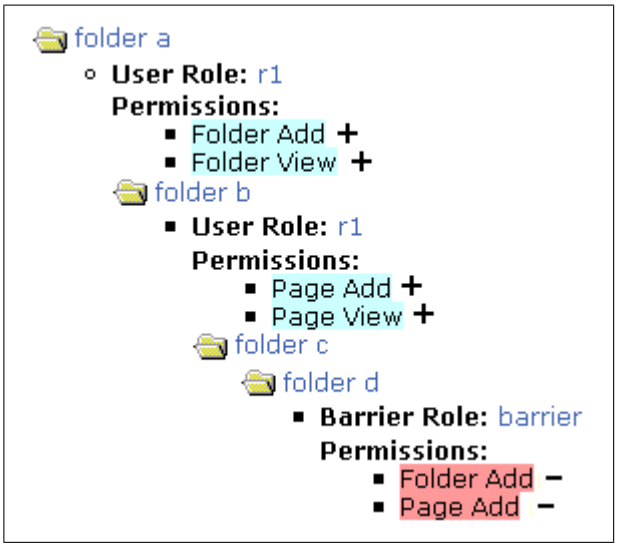


Figure 5.3: **Setup to illustrate the usage of the Barrier role.** Wiki tree with four folders *a*, *b*, *c*, and *d*. The role *r1* is defined on folder *a*, and also on subfolder *b*. The Barrier role is defined on folder *d*.

Result. The Barrier role has blocked a set of permissions on folder *d* (see Figure 5.4).

- The security settings on the folder *a*, *b* and *c* are the same as on the example seen before (see Section 5.1.1)

- The role *r1* on folder *d* owns the permissions *Folder View* and *Page View*, and has actually lost the two permissions *Folder Add* and *Page Add*. These are the two permissions that the Barrier role owns.

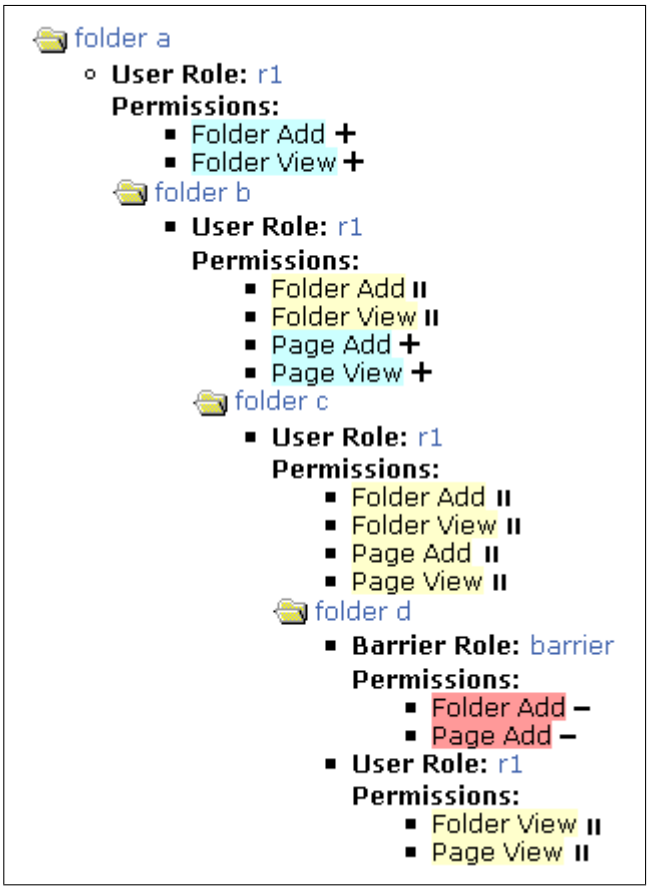


Figure 5.4: **Result of the usage of the Barrier role.** The *Barrier role* defined on folder *d* stops the acquisition mechanism for the roles on folder *d* by the set of permissions that it owns.

Explanation. All the roles are updated as expected from the root of the Wiki tree until the folder *c*. Without the Barrier role defined on folder *d*, all the acquired permissions of role *r1* on folder *c* would also be acquired to the role *r1* on folder *d*. The Barrier role will stop the acquisition mechanism for a certain set of permissions. More precisely, the acquisition is stopped for that set of permissions that is owned by the Barrier role, namely *Folder Add* and *Page Add*.

5.1.3 Acquisition. Updating the Roles of an Administrator.

Intention. An important rule concerning the Barrier role and the administrators is introduced here. The Barrier role has an effect on all roles, except for the roles that are assigned to an *administrator* that is currently visiting the relevant structure. In other words, when a user visits a resource, the acquisition mechanism only removes the *Barrier permissions* from the set of privileges of the roles of the visitor, if that user does not own any *admin permissions*.

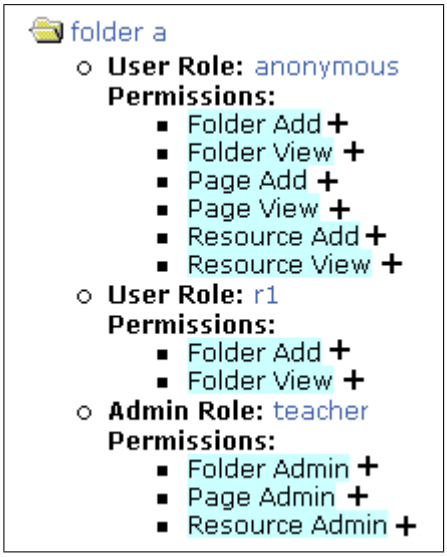


Figure 5.5: **The setup of roles and users to illustrate acquisition for an administrator.** The roles *anonymous* and *teacher* are added to folder *a*. User *ducasse* with roles *teacher* and *anonymous* is created. User *david* with roles *r1* and *anonymous* is created.

Setup. To illustrate this, we extend the Example seen before in Section 5.1.2 with new roles and users (see setup in Figure 5.5). This is done by the site administrator.

- The role *teacher* with the admin permissions *Page Admin*, *Folder Admin* and *Resource Admin* is added to the folder *a*.
- The role *anonymous* with the admin permissions *Page Add*, *Page View*, *Folder Add*, *Folder View*, *Resource Add* and *Resource View* is added to folder *a*.
- The users *david* and *ducasse* are created.
- The roles *anonymous* and *teacher* are assigned to the user *ducasse*.
- The roles *anonymous* and *r1* are assigned to the user *david*.

Result. By analysing the permissions of the two users ducasse and david currently visiting the folder *d*, it appears that the roles of *ducasse* - a user with admin permissions - did not lose any privileges at all, in contrast to the user *david* that lost the Barrier permissions *Folder Add* and *Page Add* (see results in Figure 5.6).

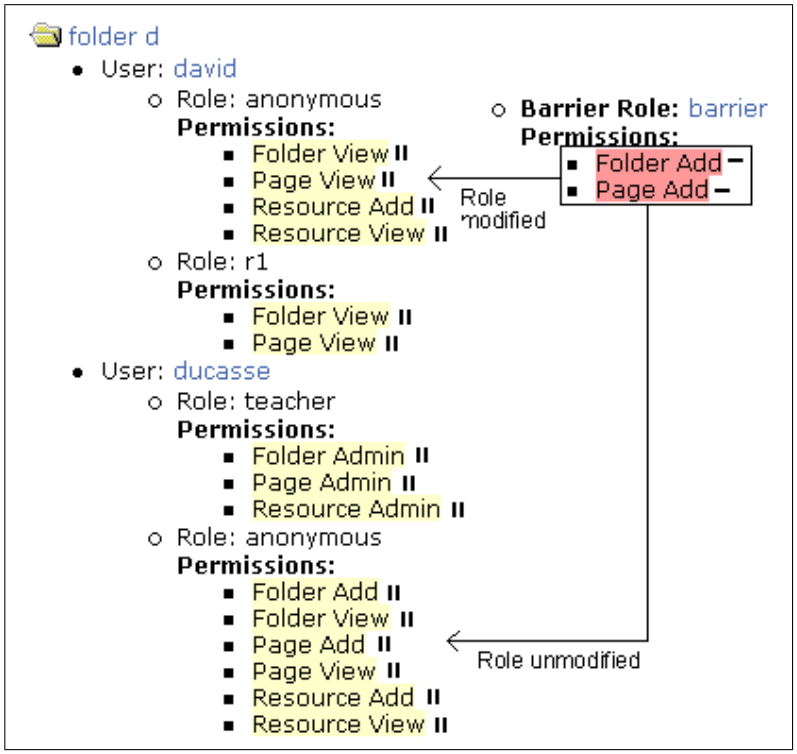


Figure 5.6: **Result. Acquisition for an administrator compared to a common user.** The administrator ducasse did not lose any permissions. The common user david lost the Barrier permissions.

Explanation. Since the Barrier role has no effect for *administrators*, no permissions at all will be blocked for them, and they get the normal set of permissions by acquisition.

This security policy guarantees that the permissions of the administrators can never be scaled down by a Barrier role. Therefore an administrator cannot remove any permissions from another one using the Barrier role. This is necessary to fulfill the needs of *save delegation* described in Section 5.1.4.

5.1.4 Delegation. Providing Safe Delegation through *Security Rules*.

Intention. Out of the box, a SmallWiki site has two different users: a *site administrator* and an *anonymous* user. The initial *site administrator* is a user with the name *admin* and the role *administrator*, which allows it to perform any duty that can be performed within the site and to use all the functions of the management interface (see Figure C.1 and Figure C.9).

The *site administrator* can create other administrators. Therefore it must assign one or several of the admin permissions *Page Admin*, *Folder Admin* and *Resource Admin* to those users. These administrators are called *sector administrators*, since they are usually responsible for a certain part of the Wiki site. A *sector administrator* is also able to create roles and users and to manage them. Since the Barrier role does not effect the permissions of an administrator, a sector administrator is not able to curtail the privileges of other administrators, even if they visit *its* part of the Wiki.

In this example a sector administrator cannot modify or delete any roles and any users it wants to. There are rules that restrict the power of the sector administrators.

Setup. This scenario illustrates a school site with a sector administrator named *ducasse*, that is able to manage its students. Therefore it creates the role *student*, creates users and assigns the role *student* to these users. There is also the need of a code expert, that is responsible to code on the whole site. Nobody but the site administrator should be able to remove the permissions *Folder Code* and *Page Code* from the code expert.

The *site administrator* has created (see Figure 5.7):

- the roles *anonymous*, *codeExpert* with the permissions *Folder Code* and *Page Code*, *secretary*, and *teacher* (teacher as an *admin role* with the permissions *Page Admin*, *Folder Admin* and *Resource Admin*).
- the users *anonymous*, *fred* with role *codeExpert*, *sally* with role *secretary* and *ducasse* with roles *teacher* and *anonymous*.

Figure 5.7: The roles and users created by the *site admin*.

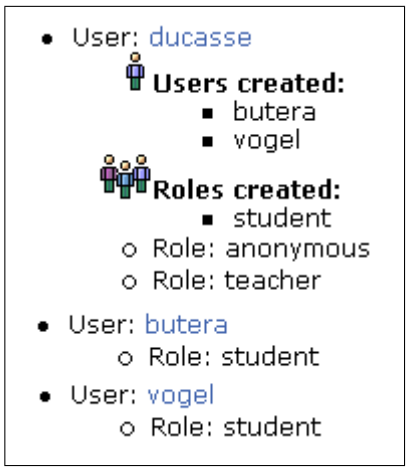


Figure 5.8: The roles and users created by the sector administrator *ducasse*.

Since the user *ducasse* owns the admin role *teacher*, it is able to create roles and users as follows (see Figure 5.8):

- the role *student* with the permissions *Folder View*, *Page View* and *Resource View*.
- the users *butera* and *vogel*, both with the role *student*.

In the next two paragraphs we explain some security rules concerning the sector administrators. These rules do not apply to the site administrator. In the first paragraph we refer to the management of roles, and in the second one we refer to the management of users. Both paragraphs are accompanied by a screenshot of the relevant user management interface. The usage of the interface is described in detail in Chapter C.

Explanation. Security Rules in Context with Management of Roles. The following rules concerning the management of *roles* apply to the sector administrators (*e.g., ducasse*). Some of these rules are displayed on the roles management user interface shown in Figure 5.9.

The *sector administrator* can:

- use the Barrier role in order to stop the acquisition mechanism (see Section 5.1.1). It can only assign the permissions that it owns himself to the Barrier role.
- create new roles. The name must not be in use before.
- only assign permissions that it owns himself to roles.
- only delete roles that it has created and any *local child role* (see Section 4.3).

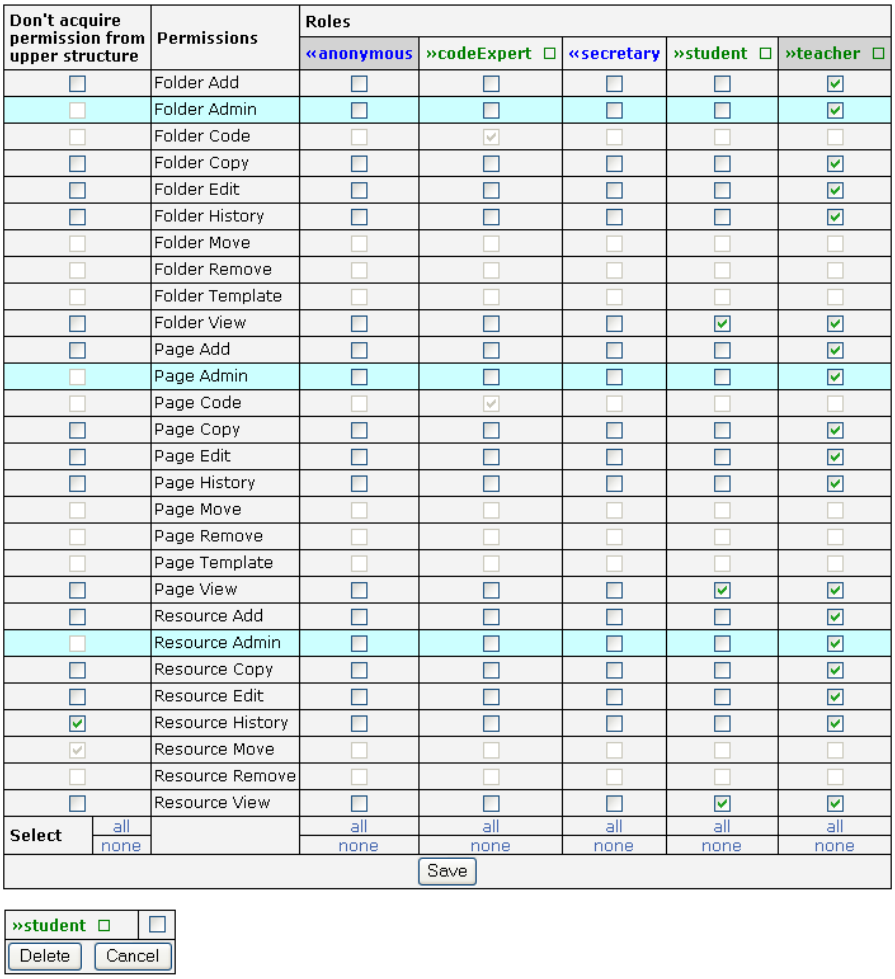


Figure 5.9: The user management interface for *roles* presented for the sector administrator *ducasse*. Some checkboxes are disabled since the sector administrators are not allowed to manage permissions that they do not own themselves.

Explanation. Security Rules in Context with Management of Users. The following rules concerning the management of *users* apply to the sector administrators. Some of these security rules are displayed in Figure 5.10.

The *sector administrator* can:

- create new users. The name must not be in use before.
- only assign roles that it has created or that it owns.
- only manage (add/remove roles form a user) users that it has created.
- only delete users that it has created.
- only change passwords of users that it has created.

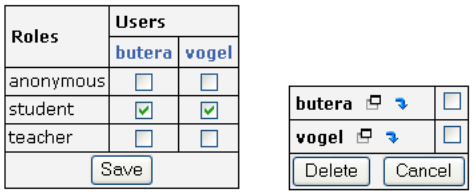


Figure 5.10: The user management interface for *users* presented for the sector administrator *ducasse*. It can only manage users that it has created, and assign roles that it owns or that it has created. The other users and roles are not listed.

If a *sector administrator* is deleted, also the *users* and the *roles* that this administrator has created will be deleted as well.

5.1.5 Summary.

Since any role on any Wiki node can be managed by administrators, they are able to create a fine-grained security policy. The special security rules seen before that are applied to a sector administrator are very important. They make sure that a sector administrator can not abuse its admin permissions, *e.g.*, remove a set permissions from another sector administrator, delete a role/user that it did not create itself, add a set of permissions to itself *etc*.

SmallWiki provides with this extended security model the ability to *safely delegate capabilities* to roles defined within different parts of a Wiki site. This is one of the important and differentiating security features of the SmallWiki Extended Security Model. It is possible to grant *sector administrators* the capability to safely administer users and their roles via the user management interface (see Figure C.1 and Figure C.9). This is called *safe delegation* because it is relatively safe to grant

users these kinds of privileges within a particular portion of a Wiki site, as it does not put at risk the operating system security nor the Wiki security in other portions of the site.

Weakness of safe delegation still refers to resource exhaustion (it is not possible yet to control a user's resource consumption), but it is possible to delegate these capabilities to semi-trusted sector administrators in order to *decentralize* control of a web site, allowing it to grow faster and require less overview from a central source.

Chapter 6

Security Scenarios for the Extended Security Model

In this chapter we illustrate the power of the SmallWiki Extended Security Model. The validation of the SmallWiki Extended Security Model is done based on security scenarios. For a detailed description on how to set up these scenarios with the graphical user interface, we refer to Section C.

The default settings of SmallWiki with the *site administrator* and the *anonymous user* may be sufficient for many simple websites and applications, especially *public-facing* sites which have no requirement for users to log in or compose explicitly their own content. We show on the scenarios how to set up security policies starting with a simple scenario and then going on to more complex ones.

We use four scenarios as follows:

- **Scenario 1: Open Editing.** A scenario where everybody is able to *view*, *add*, and *edit* content.
- **Scenario 2: Newspaper with Editors and Readers.** A site for a newspaper with two classes of users.
- **Scenario 3: Managing Common and Private Resources.** In this scenario there are resources that two classes of people need to manage in common.
- **Scenario 4: Delegating Control to Sector Administrators.** This scenario contains several sector administrators, which are responsible for their Wiki part.

These scenarios are structured as follows:

- **Intention.** We explain the purpose of the scenario.
- **Setup.** We illustrated the setup of the Wiki tree, the roles, and of the users.

- **Result.** We describe the consequences of the scenario. We might also focus on problems.

The blue colored permissions in the Figures used in these scenarios indicate permissions that are added by locally defined roles. The yellow color marks permissions that are acquired from parent roles. The permissions of the Barrier role are highlighted with red color. There are also icons used to illustrate this characteristic:

- Icon '+'. This icon indicates permissions that are added by a locally defined role.
- Icon '||'. This icon marks permissions that are acquired from parent roles.
- Icon '−'. This icon highlights permissions of the Barrier role.

6.1 Scenario 1: Open Editing.

Intention. A security policy is created for a simple *open editing* web site. Anybody should be able to *view*, *add* and *edit* content. There is no need to know who has made the changes.

Setup. The roles and users are used as they come out of the box with the default SmallWiki installation. (see Figure 6.1).

- **Roles:**
  - **administrator**. All permissions.
  - **anonymous**. With permissions *Page View*, *Page Add*, *Page Edit*, *Folder View*, *Folder Add*, *Folder Edit*, *Resource Add*, *Resource View* and *Resource Edit*.
- **Users:**
  - **site administrator**. With role *administrator*.
  - **anonymous**. With role *anonymous*.

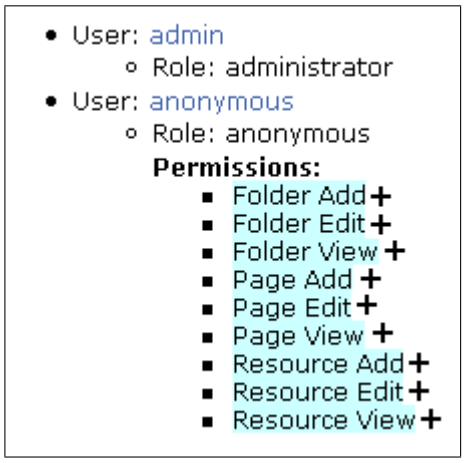


Figure 6.1: **Scenario 1. Open editing policy.** Everybody can view, edit, and add content.

Result. In this scenario, the *anonymous user* has the privileges of *viewing*, *editing*, and *adding* content to the site, while the *site administrator* is still privileged to view the history, restore older versions, delete pages, manage roles and users etc. This security policy is valid through the entire Wiki site since SmallWiki roles

get their security policies from their parent roles by acquisition (see description in Section 5.1.1).

6.2 Scenario 2: Newspaper with Editors and Readers.

Intention. A site for a newspaper is set up. Therefore two classes of users are needed:

- Readers. These users are only allowed to view the content of the site.
- Editors. They are responsible for the content of the site.

Setup. The setup is illustrated on Figure 6.2.

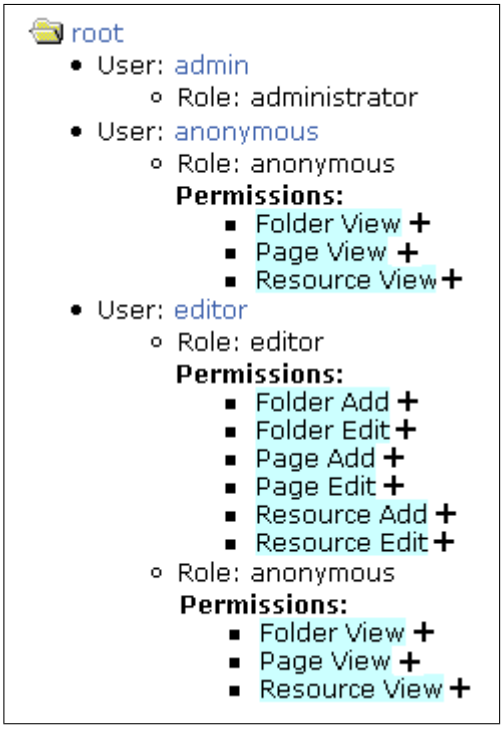


Figure 6.2: **Scenario 2. Site with two classes of users.** Readers - the anonymous users - can only view content, and editors are responsible to edit and add content.

There are *editors*, that are responsible for editing and adding content. The *anonymous user* - the reader - is only allowed to view the content on the Wiki site. All needed roles are created on the root folder of the Wiki tree by the site administrator.

- **Roles:**
  - **administrator**: owns all permissions.

  - **editor**: with permissions *Page View*, *Page Add*, *Page Edit*, *Folder View*, *Folder Add*, *Folder Edit*, *Resource Add*, *Resource View* and *Resource Edit*.
  - **anonymous**: with permissions *Page View*, *Folder View* and *Resource View*.

- **Users:**
  - **site administrator**: with role *administrator*.
  - **editor**: with roles *editor* and *anonymous*.
  - **anonymous**: with role *anonymous*.

Result. As long as a Wiki user only requests to *view* the content of the site, the Wiki server assigns the user *anonymous* to this visitor. When the user wants to perform the action *edit*, the server will deny to process the edit request. The server asks the visitor to log-in in order to perform *authentication*. After successfully logging in as *editor*, the server assigns the role *editor* to it, performs *user authorization* and the visitor is able to edit the content of the site. Since the roles are only defined at the root folder, the security policies are in effect on the entire Wiki tree.

6.3 Scenario 3: Managing Common and Private Resources.

Intention. A Wiki Site of a company with programmers and salespeople. This scenario contains *programmers* and *salesmen* and represents a Wiki site for a software engineering company. There are two classes of users, *programmers* and *salesmen* that are able to view the content of the entire site. In general the programmers and the salespeople **add**, **admin**, and **edit** the content of different web resources on their own. However, there might be documents that both types of people need to manage in common, such as advertisements that have to contain complex software specifications. The permissions to manage the content on both resources cannot be assigned to the two groups of users, because a programmer should not have free access to all resources of the salesmen and vice versa. Therefore a new folder for the shared resources is created. Both groups store the common resources in this folder, and full access to this folder is given to both groups. To make it easier to illustrate, we use only the permissions concerning the *Folder* (*e.g.*, *Folder View*, *Folder Admin etc.*) in this scenario.

Setup. The setup is illustrated on Figure 6.3.

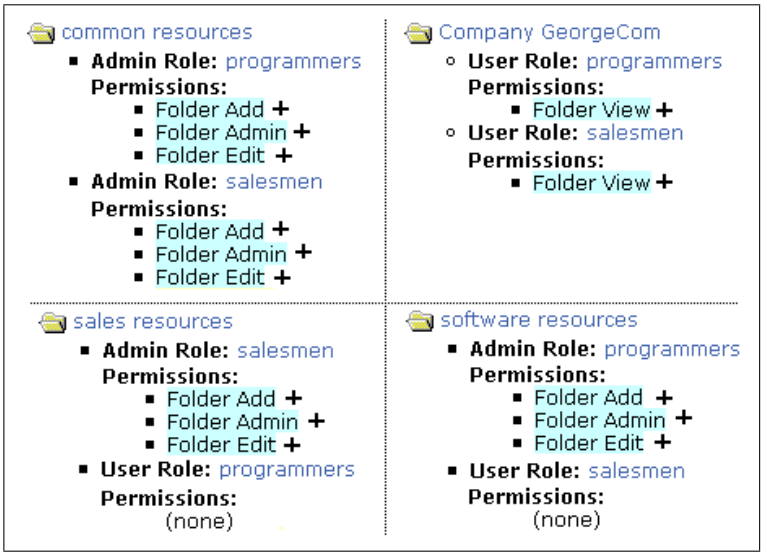


Figure 6.3: **Scenario 3. Setup of a Wiki site, where resources have to be managed in common by two classes of people.** The permissions to *add*, *admin*, and *edit* are given on specific folders to the roles.

All roles and users are created by the site administrator. The root folder named *Company GeorgeCom* is created. The three folders *sales resources*, *software resources* and *common resources* are appended as subfolders to the root folder. There are several roles created on different folders:

- **Roles on folder *Company GeorgeCom***
  - **programmers**. With permission *Folder View*.
  - **salesmen**. With permission *Folder View*.
- **Roles on folder *sales resources***
  - **salesmen**. With permission *Folder Admin* and *Folder Edit*.
- **Roles on folder *software resources***
  - **programmers**. With permission *Folder Admin* and *Folder Edit*.
- **Roles on folder *common resources***
  - **programmers**. With permission *Folder Admin* and *Folder Edit*.
  - **salesmen**. With permission *Folder Admin* and *Folder Edit*.

Result. Both groups of users are responsible for their private resources on their own: the salesmen for the folder *sales* and the programmers for the folder *software*). They are additionally able to manage the common resources on folder *common*. Both groups of users are able to view the content, since they received the permission *Folder View* from their role defined in the root folder (see Figure 6.4).

Problem. There might still be problems when resources are being edited by different people at the same time: it is all too easy to overwrite each others' changes unless the editors are careful. Some editors, like GNU Emacs [23], try to make sure that the same file is never modified by two people at the same time. Unfortunately, the web browsers (*e.g.*, Mozilla) and the SmallWiki server do not support such a safeguard. The changes are always accessible by the history tool of SmallWiki, so that an administrator is able to *merge* the changes if necessary.

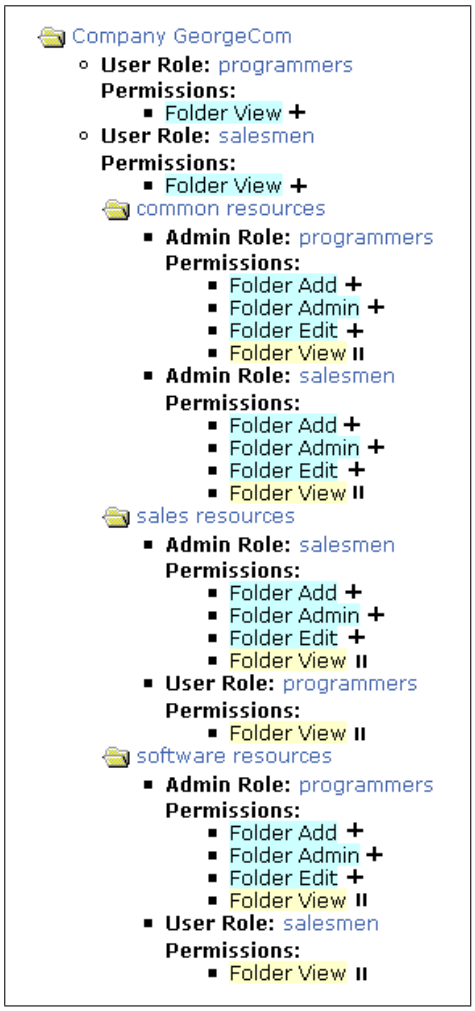


Figure 6.4: **Scenario 3. Result. Wiki site where two groups of users are able to manage common resources.** The *salesmen* and the *programmers* are responsible for their private resources on their own. Additionally both groups of users are able to manage the shared resources in common.

6.4 Scenario 4: Delegating Control to Sector Administrators.

Intention. The *pattern of delegation* is central to the SmallWiki Extended Security Model. We can collect resources in folders, and create roles in these folders in order to manage their contents (see description of delegation on Section 5.1.4). A scenario with a Wiki site for a University is used to illustrate this. The *site administrator* creates a *sector administrator* named *ese administrator* for the ESE[1] lectures and delegates the management of those resources to it. This *ese administrator* creates a sector administrator for every subfolder (group01, group02, group03) and delegates the responsibility for those resources to them.

Setup of the Wiki tree. The folder *University of Bern* is created as the root of the Wiki site. The *Lectures* folder is added as a subfolder of it. *ESE* is a subfolder of *Lectures* and contains the three subfolders *group01*, *group02* and *group03* (see Figure 6.5).

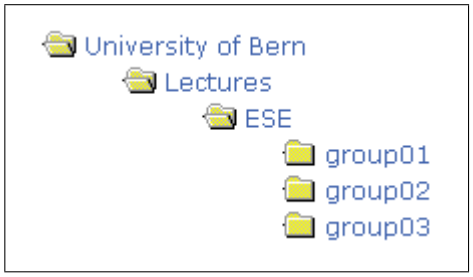


Figure 6.5: **Scenario 4. Setup of the folders of a school site.** Wiki tree with six folders.

Setup of the roles and users. The management of the *ESE* folder in this Wiki site is delegated over to an *ese administrator*. This *ese administrator* should not be able to change any structures which live outside the *ESE* folder, that is why the *ese admin role* is not defined outside of the *ESE* folder.

The *site administrator* creates:

- the role *ese administrator* with the admin permissions *Folder Admin*, *Page Admin* and *Resource Admin* and other permissions on the folder *ESE*.
- a new user named *michele* and assigns the roles *ese administrator* and *anonymous* to the user *michele*.

[1] Evolution of Software Engineering.

The user *michele* is able to create *sector administrators* for the different *ese* group folders. Activities of the user *michele*:

- It defines the role *group01 admin* with the admin permissions on folder *group01*.
- It creates a user *admin01* and assigns the role *group01 admin* and *anonymous* to this new user.
- It does the equivalent on the folders *group02* and *group03*.

The users *admin01*, *admin02* and *admin03* are now able to create and manage group members such as users with the role *student01*, *student02*, and *student03* on their folder:

- User *admin01* creates role *student01* on folder *group01* and a user named *harry* with the new role *student01* and the role *anonymous*.
- The equivalent happens on folders *group02* and *group03*:
  - *sally* with role *student02* on folder *group02*.
  - *kirk* with role *student03* on folder *group03*.

These roles settings are shown in Figure 6.6 and the users properties are illustrated in Figure 6.7.

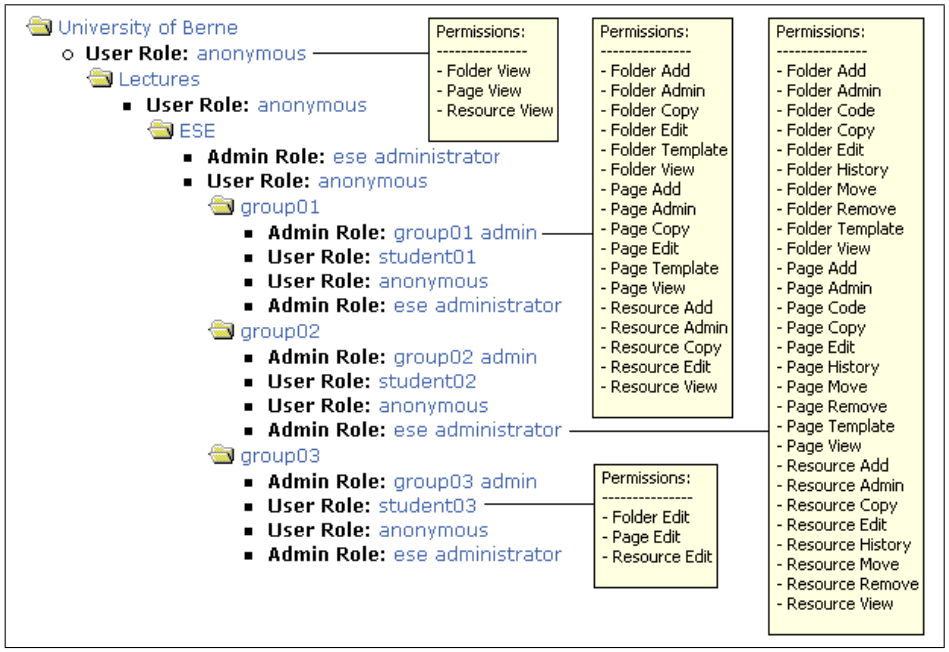


Figure 6.6: **Scenario 4. The setup of roles on a school site.** The roles are created in order that responsibilities are safely delegated to *sector administrators*.

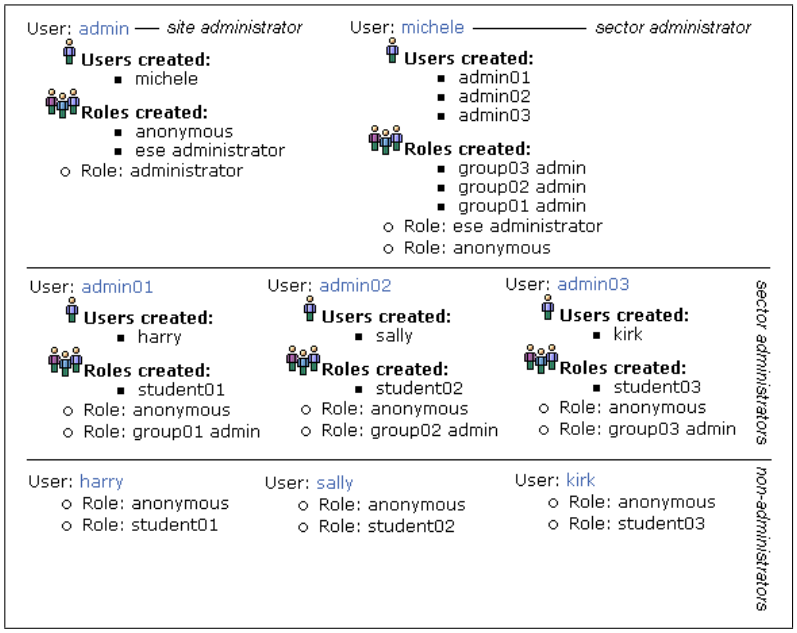


Figure 6.7: **Scenario 4. Overview of the users and their properties in this school site.** The *site administrator*. The non-administrators *harry*, *sally*, and *kirk*. The *sector administrators michele*, *admin01*, *admin02*, and *admin03*.

Result. To illustrate the effects of these security settings, some of the folders and the users with their updated roles for the current folders are analyzed. The site administrator named *admin* is listed just once, since it always owns all permissions.

Result on folder *University of Berne* (see Figure 6.8)

- The site administrator named *admin* always keeps its set of all possible permissions.
- Since the roles *ese admin*, *admin01*, *admin02*, *admin03*, *student01*, *student02* and *student03* are not defined on this folder yet, the users with these roles only own the permissions which they get from the *anonymous* role (*Folder View*, *Page View* and *Resource View*).

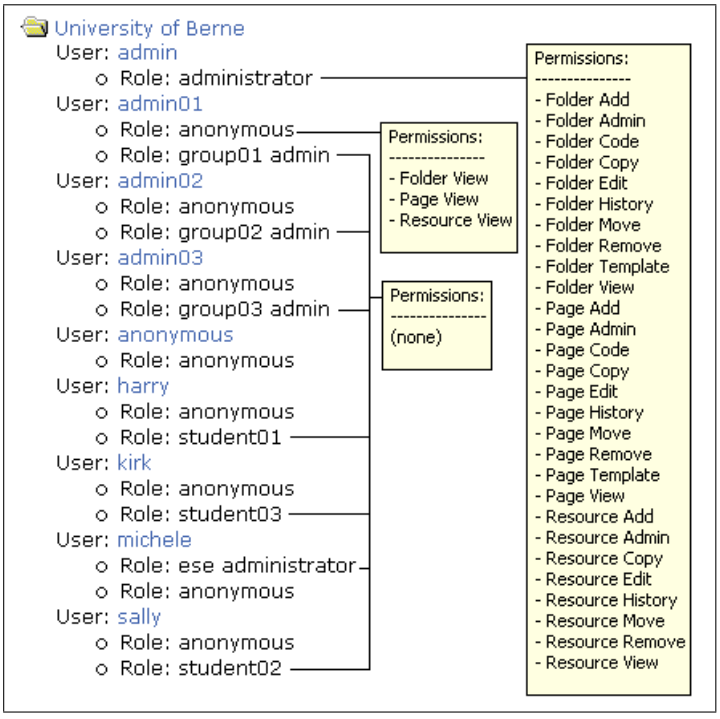


Figure 6.8: Scenario 4. Users with their updated roles on the root folder *University of Bern*

Result on folder *Lectures*. Since no roles were added or modified on this folder, the security settings for all users stay the same like on the folder *University of Bern*.

Result on folder *ESE* (see Figure 6.9) On this folder, the role *ese admin* was added. That is why only the permissions for the user *michele* change.

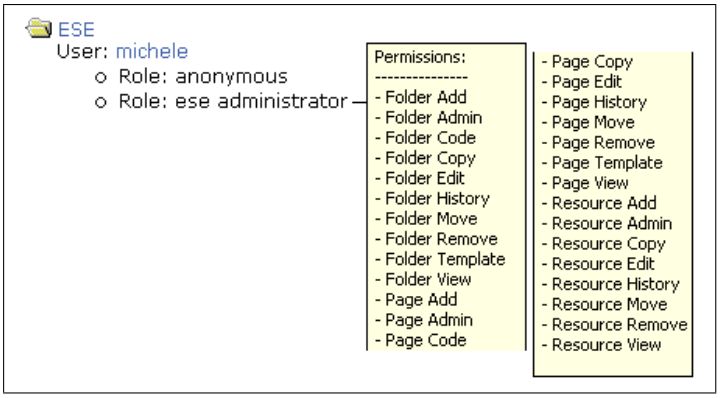


Figure 6.9: Scenario 4. User *michele* with its updated roles on folder *ESE*. The role *ese admin* that is assigned to the user michele has been created on this folder. All other users keep their security settings from the folder *Lectures*.

Result on folder *group01* (see Figure 6.10) The users *admin01* and *harry* gain a set of permissions, because their roles *group01 admin* and *student01* were added on the current folder. The permissions of the other users stay the same as on the parent folder *ESE*.

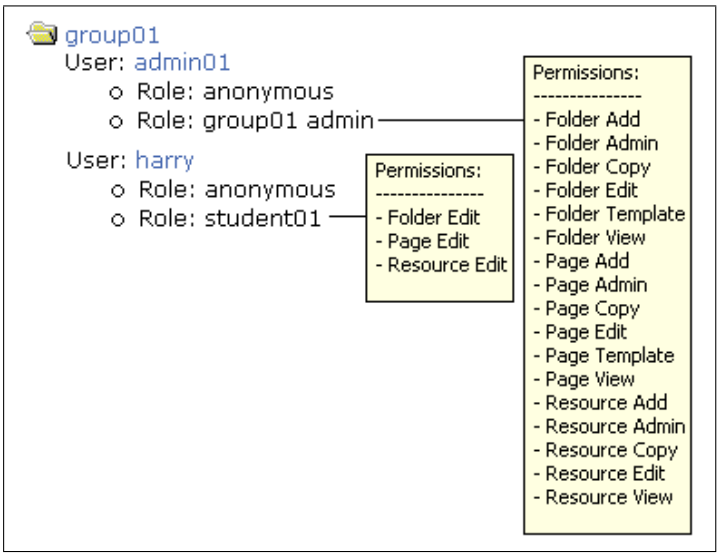


Figure 6.10: Scenario 4. Users *admin01* and *harry* with their updated roles on folder *Group01*

Anybody with the role *anonymous* is still allowed to view the content on the folders *group01*, *group02* and *group03*. If the user *admin01* decides to restrict the view-access for other users on his wiki-part, it has to make use of the Barrier role. Therefore it adds a Barrier role with the permissions *Folder View*, *Page View* and *Resource View* to the folder *group01* (see Figure 6.11).

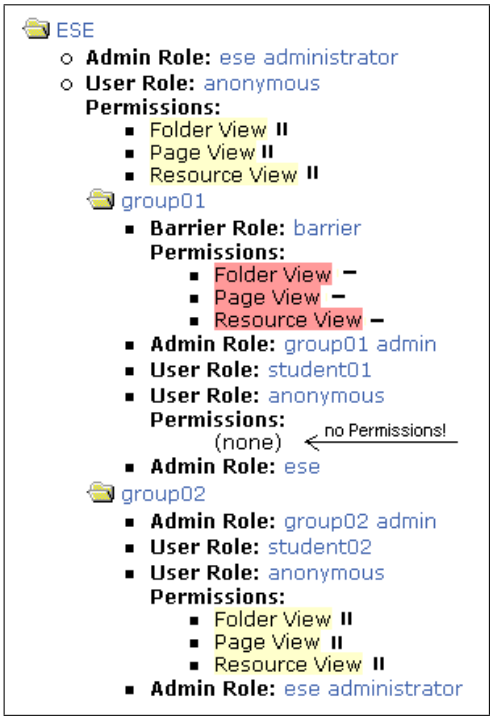


Figure 6.11: Scenario 4. User admin01 restricts the view access for foreign users. Therefore it adds the Barrier role with the view permissions on folder group01. Thereby the view permissions of role anonymous are removed on folder group01.

Summary. This scenario with several sector administrators defined on different folders illustrates how save delegation is achieved for a Wiki site. Each sector administrator is responsible for its part of the Wiki tree. This *pattern of delegation* is central to the SmallWiki Extended Security Model. We can collect resources in folders and in their sub-folders, and design sector administrators that are responsible to manage the contents and users of their Wiki part. It is the assigned responsibility of these administrators to manage the user privileges on their part of the Wiki site.

Chapter 7

Formal Description of SmallWiki Extended Security Model

This chapter provides a formal description of the SmallWiki Extended Security Model. The relevant elements of the model such as structure, permission, Meta role, user *etc.* are introduced, and it is shown how the roles of a user are computed in context of a certain node of the Wiki tree.

The sections of this chapter are structured as follows:

1. **Intro and definition.** A short description is given, and the formulas are defined.
2. **Characteristics.** The attributes of the element are explained.
3. **In the context of the implementation.** The element is described, as it appears in the SmallWiki implementation. This description repeats some of the characteristics.
4. **Computing.** Formulas for computing are defined and examples of how to use these formulas are given.

7.1 Context C

First a context C is set up. It contains the basic elements of the formal description.

$$C = (S, P, U, M, R, B)$$

- **S:** Structure. The structure is the basic entity of SmallWiki. It is either a *Folder*, a *Page*, or a *Resource* (see Section 7.2).

- **P:** Permissions. These are the individual actions a user is allowed to perform in the system (see Section 7.3).
- **M:** Meta Roles. Meta roles are containers of a set of permissions. These roles are attached to a structure (see Section 7.4).
- **U:** Users. The users are visiting the Wiki tree (see Section 7.5).
- **R:** User Roles. User roles are containers of a set of permissions. These roles are assigned to a user (see Section 7.5).
- **B:** Barrier Roles. Barrier roles are containers of a set of permissions. These roles are attached to a structure. These roles are used in order to remove a set of permissions from Meta roles (see Section 7.6).

7.2 Structures S_{x^n}

Wiki Structures are nodes of the Wiki tree. The Wiki tree is a data structure accessed beginning at the root node. Each node is either a leaf or an internal node. An internal node has one or more child nodes, and is called the parent of its child nodes. All the children of the same node are siblings. Contrary to a physical tree, the root is depicted at the top of the tree, and the leaves are depicted at the bottom (see Figure 7.1).

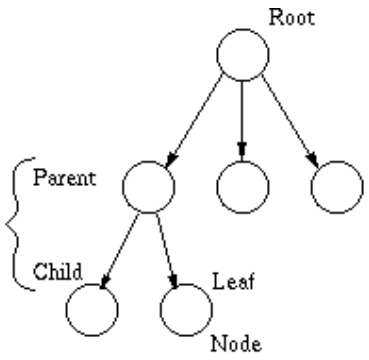


Figure 7.1: A Wiki tree with root, parent, children and leafs.

General. Nomenclature of the nodes with numbers.

- The root is labeled S_0.
- Its sub-nodes are labeled S_{00}, S_{01}, S_{02}, S_{03} etc.
- The subnodes of S_{00} are labeled S_{000}, S_{001}, S_{002}, S_{003} etc.

- The subnodes of S_{01} are labeled S_{010}, S_{011}, S_{012}, S_{013} etc.
- Abstract. The subnodes of S_{xxxx} are labeled S_{xxxxi} ; $x, i \in \{0..9\}$.

The enumeration is illustrated in Figure 7.2

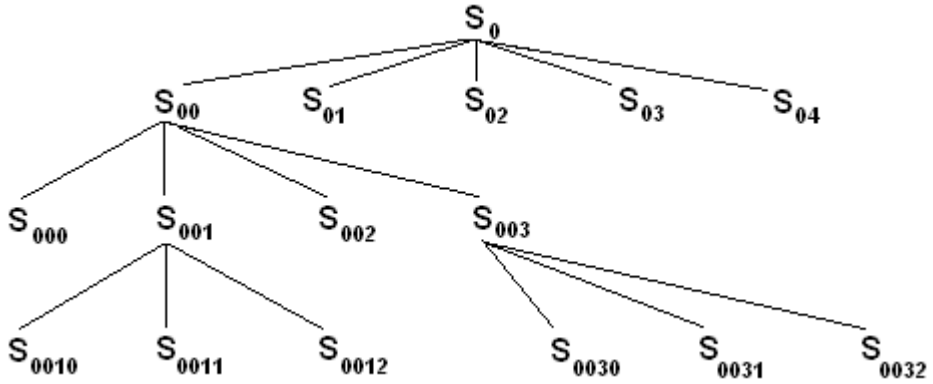


Figure 7.2: Wiki tree made out of structure-nodes to illustrate the nomenclature.

Abstract. Nomenclature of the nodes with the set of unique one-digit symbols. The previous example used only the set of one-digit numbers in order to differ one sibling from another. But this limits the number of children to ten. To improve the formal description we are using one-digit unique symbols. Therefore we introduce the set $\mathbb{A}$:

$$\mathbb{A} = \text{the set of one-digit unique symbols}$$

and therefore

$$[\text{a-z, A-Z, 0-9, } \alpha - \zeta] \subset \mathbb{A}$$

In order to describe any node in the Wiki tree, we say that

$$S_{\underbrace{x \cdots x}_{n}} = S_{x^n}, n \in \mathbb{N}, x \in \mathbb{A}$$

E.g., to refer to the node $Sxxxxu$ we write

$$S_{xxxxu} = S_{x^4u};\ x, u \in \mathbb{A}$$

and the children of S_{x^4} are labeled S_{x^4i} ; $x, i \in \mathbb{A}$.

7.2.1 Characteristics

- A structure has a parent (exception: the root of the Wiki tree has no parent).
- A structure S_{x^n} can have children $S_{x^n i}$ for $n \in \mathbb{N}$.
- A structure S_{x^n} can have siblings $S_{x^{n-1}i}$ for $n \geq 2$.
- A structure can have Meta Role(s) M_i (see Section 7.4).

7.2.2 In the context of the implementation

- The wiki site is set up as a Wiki tree. Its nodes are structures.
- A structure is either a folder, a page or a resource (see Section 3.6).
- Only a folder can have children.
- A structure has a set of permissions, which describe what actions can be performed on it.

7.3 Permissions P

The permissions are the individual actions a user is allowed to perform in the system.

First we define the total set of the permissions TP as they occur in the implementation of SmallWiki. Then this set is pulled together with our formal description and the total set of permissions P used in this context is defined.

- **Folder Permission FP.** There are permissions which are applied in context of a *Folder* as follows:

  $$FP = \{Folder\ Add, Folder\ Admin, Folder\ Code, Folder\ Copy, \\ Folder\ Edit, Folder\ History, Folder\ Move, \\ Folder\ Remove, Folder\ Template, Folder\ View\}$$

- **Page Permission PP.** There are permissions which are applied in context of a *Page* as follows:

  $$PP = \{Page\ Add, Page\ Admin, Page\ Code, Page\ Copy, \\ Page\ Edit, Page\ History, Page\ Move, Page\ Remove, \\ Page\ Template, Page\ View\}$$

- **Resource Permission RP.** There are permissions which are applied in context of a *Resource* as follows:

$$RP = \{Resource\ Add, Resource\ Admin, Resource\ Copy, \\ Resource\ Edit, Resource\ History, Resource\ Move, \\ Resource\ Remove, Resource\ View\}$$

- **Total set of permissions TP.**

$$TP = FP \cup PP \cup RP$$

Permissions P in the formal system.

The set of permissions P in this formal system is defined as follows:

$$P = \{p_x | \forall x \in TP : p_x \in \overbrace{[0,1]}^{\mathbb{B}}\}$$

I.e., the indexing of the elements of P is done via the names of the permissions.

The value of a permission is either *0* or *1*. *0* means *false*, and *1* means *true*. In this formal description we are using the values *0* and *1*, because it is more common to calculate with these values in context with vectors.

7.3.1 Characteristics and example

A set of permissions is used to create Meta Roles (see Section 7.4) and user roles (see Section 7.5.1). A permission is a boolean which describes if a certain privilege exists.

E.g., the permission

$$p_{(Folder\ View)} = 1$$

means, that the permission with name *Folder View* exists, whereas the permission

$$p_{(Folder\ History)} = 0$$

does not exist.

7.3.2 In the context of the implementation

A permission represents a privilege in the system and is the basic entity for the security management. The permissions are the individual actions a user is allowed to perform in the system. The names of the permissions defined in the set TP are also used in the SmallWiki implementation.

7.3.3 Admin Permissions AP

There are some admin permissions AP that allow one to perform admin actions on a Structure. The set of admin permissions AP is defined as follows:

$$AP = \{p_x | x \in [Folder\ Admin, Page\ Admin, Resource\ Admin]\}$$

This set of admin permissions AP will classify the users in the systems into two groups:

1. **Admin users.** Users in the system that own an admin permission. They are also called *administrators*, and are described in Section 7.5.3.

2. **Ordinary users.** All other users.

7.4 Meta Roles M

Meta Roles are containers of permissions. Meta Roles are attached on the nodes of the Wiki tree. They are used in order to compute the final set of roles of a user as described in Section 7.7. A Meta Role M_i is a vector[1] and is described as follows:

$$M_i = |mi| \text{ with elements } mi_x \in P \text{ and } x \in TP$$

To describe that a Meta Role M_i is attached to a structure S_{x^n} we write

$$M_i \bowtie S_{x^n}$$

Now we can describe the Meta Roles dependent on their container. *I.e.*, if we want to access the Meta Role M_i of structure S_{x^n}:

$$M_i(S_{x^n}) = |mi| \text{ with elements } mi_x \in P,\ x \in TP \text{ and } M_i \bowtie S_{x^n} \tag{7.1}$$

7.4.1 Characteristics

- Meta Roles are defined on a element S_{x^n} of the Wiki tree.

- A Meta Role is a vector. Its components are boolean values that define a specific abstract permission.

[1] all vectors used in this chapter have the same length, and their elements are ordered in the same way via index i $\in TP$.

7.4.2 In the context of the implementation

- A Meta Role is static through the Wiki tree until it is redefined by the administrator (see Section 7.5.3), or modified by the Barrier role (see Section 7.6).
- The Meta Roles are used in order to compute the roles of a user, that is visiting a certain structure of the Wiki tree.

7.4.3 Computing. Creation and modification of Meta Roles

- Meta Role on structure S_{x^n} can be created and modified.
- The creation and modification rules of Meta roles are the same as for the Barrier role described in Section 7.6.2.

7.5 User U_x and its set of Roles $R(U_x)$

A user can visit the nodes of a Wiki tree and perform actions on these nodes. Permissions can also be assigned to users via roles. These permissions are the individual actions a user is allowed to take in the system. The set of Users U is described as follows:

$$U = \{u_x | x \in [0, \cdots, m], m \in \mathbb{N}\}$$

7.5.1 Roles of a User $R(U_x)$

A role is - similar to a Meta Role - a vector made out of permissions. The components of a role vector are boolean values. One or several roles are assigned to a user. Thereby the user gets a set of permissions.

A single role R_i is described as follows:

$$R_i = |ri| \text{ with elements } ri_x \in P \text{ and } x \in TP$$

A role R_i can be assigned to a user U_x. To describe this, we write:

$$R_i \bowtie U_x$$

A User U_x owns one or several roles $R(U_x)$:

$$R(U_x) = \{R_i | i \in [0, \cdots, m], m \in \mathbb{N}, R_i \bowtie U_x\} \tag{7.2}$$

7.5.2 Permissions of a User $P(U_x)$

The set of permissions $P(U_x)$ that a User U_x owns are the permissions $p_i \in P$ which boolean values are set to 1. This set $P(U_x)$ is defined as follows:

$$P(U_x) = \{p_u | \forall\ |ur| \in R(U_x) \text{ with elements } p_u \in P : u \in TP \text{ and } p_u = 1\} \tag{7.3}$$

7.5.3 Admin Users AU

The admin users

$$AU = \{u_x | \exists i \in TP : p_i \in P(u_x) \text{ and } p_i \in AP\}$$

are allowed to perform admin actions on the system.

7.5.4 Characteristics

- A user can visit a node in the Wiki tree.
- A user has one or several role(s). The permissions of these roles may change as the user travels through the nodes of the Wiki tree, since the roles are dependent on the Meta roles of the Wiki tree (see Section 7.7).

7.5.5 In the context of the implementation

- A user has one or several role(s). The set of the permissions contained in these roles describes what actions this user can perform in the system.
- A role can be assigned to one or several users. That is how a set of permissions is assigned to a user.

7.6 Barrier Role B

In this section we describe the Barrier role, how it is applied to Meta Roles, the way an admin user can modify it, and how its value is set in context of a certain user. The Barrier Role B is used to set a certain set of permissions of Meta Roles to zero.

A Barrier Role B is a vector

$$B = |b| \text{ with elements } b_x \in P \text{ and } x \in TP$$

with default value

$$B = |b| \text{ with elements } b_x = 1 \text{ and } x \in TP$$

7.6.1 Characteristics

- An admin user can modify it.
- A Barrier Role is attached on structures S_{x^n} just like a Meta Role.
- There is exactly one Barrier Role on a structure S_{x^n}.
- The Barrier Role is used to set up a barrier for certain permissions of Meta Roles. The barrier is set for those permissions of the Meta Roles, wherever the values of the Barrier Role are set to zero. *I.e.*, if we want to set up a barrier for the permission with index *Folder View*, we set the element with that index on the Barrier role to zero.

7.6.2 Computing

Barrier role in context of the Wiki tree. A Barrier role is attached on a node of the Wiki tree just like Meta Role. In order to access the Barrier role vector $|b|$ of structure S_{x^n} we write:

$$B(S_{x^n}) = |b|, \text{ with } |b| \bowtie S_{x^n}$$

Barrier role in context of the Meta Roles. We describe how the Barrier role can remove some permissions from a Meta Role. When we talk about *removing a permission* p_i from the Meta Role M_x, we mean that the element p_i of the Meta Role vector is set to zero. The vector elements bp of the Barrier role with value zero are the ones that will be removed from the Meta Roles.

The function $f(\mathbf{b}, \mathbf{M_i})$ applies the Barrier role $|b|$ to one Meta Role M_i

$$f(\mathbf{b}, \mathbf{M_i}) = |b| \wedge M_i \tag{7.4}$$

Example.

- *Barrier role with vector* $|b| = \langle 1, 1, 0 \rangle$
- *Meta role with vector* $|v| = \langle 1, 0, 1 \rangle$

$$f(\mathbf{b}, \mathbf{M_i}) = |b| \wedge |v| = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \wedge \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0^2 \end{pmatrix}$$

The function $g(\mathbf{b}, \mathbf{M_i})$ applies the Barrier role $|b|$ to a set of Meta Roles M

$$g(\mathbf{b}, \mathbf{M}) = |b| \wedge \left(\bigvee_{i=1}^{m} (M_i | M_i \in M,\ m \in \mathbb{N}) \right)$$

Barrier role in context of the User. Barrier Roles are not applied to an admin user, *i.e.,* a Barrier Role can not remove permissions of a admin user. Therefore the default Barrier role will be used for the admin user.

$$B\left(U\right)\left(S_{x^n}\right) = \begin{cases} \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}, U \in AU \\ B\left(S_{x^n}\right), U \notin AU \end{cases} \tag{7.5}$$

Modification of a Barrier role using a modifier vector. An administrator is not allowed to set up a Barrier role in the way it wants to. It can only modify those permissions of the Barrier roles, that it owns itself via its roles. We define a rule that describes how an administrator can edit the Barrier role B. Therefore we introduce the modifier vector. The administrator can only change a Barrier role via this modifier vector.

- $\vec{u}$: Permission vector of an administrator.
- $\vec{v_i}$: The administrator can set up a modifier vector v_i such that:

$$|u| = \bigvee_{i=1}^{m} v_i,\ m \in \mathbb{N}$$

- $\vec{b}$: Permission vector of Barrier.

- **Activate (set vector elements in Barrier role to 0) a set of permissions in Barrier role.** If the admin user wants to set up a barrier for a certain permissions p_x, it sets the element in the modifier vector $|v_i|$ with index x to one, and the formula

$$B = \neg\vec{v_i} \wedge \vec{b}$$

  is applied.

[2] set to zero by Barrier role

- **Deactivate (set vector elements in Barrier role to 1) a set of permissions in Barrier role.** If the admin user wants to remove a barrier for a certain permissions p_x, it sets the element in the modifier vector $|v_s|$ with index x to one, and the formula

$$B = \vec{v_s} \vee \vec{b} \text{ with } |u| = \bigvee_{i=1}^{m} v_s,\ m \in \mathbb{N}$$

  is applied.

- **Result.** This rule describes how an admin user can edit the Barrier Role

$$B = (\neg\vec{v_i} \wedge \vec{b}) \vee (\vec{v_s} \vee \vec{b})$$

Example of a modification of Barrier role by an admin user.

- *Permission vector of admin user on Structure S_{x^n} is* $\begin{pmatrix}1\\0\\1\\0\end{pmatrix}$
- *Permission vector of Barrier role on Structure S_{x^n} is* $\begin{pmatrix}1\\1\\0\\0\end{pmatrix}$ *, the 0-flags have been set by some other admin user.*

1. ***Intention 1.*** *Activate (set vector elements in Barrier role to 0) as many flags in the Barrier role as possible. Therefore the admin user sets as many elements of the modifier vector v_i to one.*

$$\Rightarrow v_i = \begin{pmatrix}1\\0\\1\\0\end{pmatrix}$$

$$\neg\vec{v_i} \wedge \vec{b} = \neg \begin{pmatrix}1\\0\\1\\0\end{pmatrix} \wedge \begin{pmatrix}1\\1\\0\\0\end{pmatrix} = \begin{pmatrix}0\\1\\0\\1\end{pmatrix} \wedge \begin{pmatrix}1\\1\\0\\0\end{pmatrix} = \begin{pmatrix}0\\1^{3}\\0\\0\end{pmatrix}$$

2. ***Intention 2.*** *Deactivate (set vector elements in Barrier role to 1) as many flags in the Barrier role as possible. Therefore the admin user sets as many elements of the modifier vector v_s to one.*

$$\Rightarrow v_i = \begin{pmatrix}1\\0\\1\\0\end{pmatrix}$$

$$\vec{v_s} \vee \vec{b} = \neg \begin{pmatrix}1\\0\\1\\0\end{pmatrix} \vee \begin{pmatrix}1\\1\\0\\0\end{pmatrix} = \begin{pmatrix}1\\1\\1\\0^{4}\end{pmatrix}$$

[3] admin user could not activate this flag, because it does not own that permission
[4] admin user could not deactivate this flag, because it does not own that permission

7.7 Computing Roles of a User

Summary.

- Structure S_{x^n} has a set of Meta Roles $M_1(S_{x^n}), \cdots, M_n(S_{x^n})$, see (7.1).
- User U_i has a set of roles $R_1, \cdots, R_m$. This set of roles is described with $R(U_i)$, see (7.2).
- Barrier Role on structure S_{x^n} in context of the user U_i is $B(U_i)(S_{x^n})$, see (7.5).
- Function $f(\mathbf{b}, \mathbf{M_i})$ applies the Barrier role $|b|$ to one Meta Role M_i, see (7.4).

7.7.1 Computing the Roles of a User

Computing without Barrier role. The Roles R of a user U_i are computed as follows:

$R_1(U_i)(S_{x^n}) = \bigvee_{i=1}^{n} M_1(S_{x^i})$
$R_2(U_i)(S_{x^n}) = \bigvee_{i=1}^{n} M_2(S_{x^i})$
$\vdots$

and therefore we get

$$R_m(U_i)(S_{x^n}) = \bigvee_{i=1}^{n} M_m(S_{x^i}) \text{ with } n \in \mathbb{N} \tag{7.6}$$

The roles of the user U_i are: $R(U_i)$ (see (7.2)) and its set of permissions is $P(U_i)$ (see (7.3)).

Computing with Barrier role. The Barrier role will stop the acquisition mechanism for some permissions. We are using a recursion to calculate the role R_a of the user U_i on structure S_{x^n}.

$R_a(U_i)(S_x) = M_a(S_x)$
$R_a(U_i)(S_{x^2}) = M_a(S_{x^2}) \vee f(\mathbf{B(U_i)(S_{x^2})}, \mathbf{R_a(U_i)(S_x)})$
$R_a(U_i)(S_{x^3}) = M_a(S_{x^3}) \vee f(\mathbf{B(U_i)(S_{x^3})}, \mathbf{R_a(U_i)(S_{x^2})})$
$\vdots$
$R_a(U_i)(S_{x^n}) = M_a(S_{x^n}) \vee f(\mathbf{B(U_i)(S_{x^n})}, \mathbf{R_a(U_i)(S_{x^{n-1}})})$

The roles of the user U_i are: $R(U_i)$ (see (7.2)) and its set of permissions is $P(U_i)$ (see (7.3)).

7.7.2 Example for Computing the Roles of a User

A simple Wiki tree is set up that contains some Meta roles. It is shown how the roles of a user are computed on the Wiki node S_{000}. The vectors used in this example have a length of four and are ordered by the index sequence: *Folder View*, *Folder Edit*, *Folder History* and *Folder Remove*.

Setup. Wiki tree

- Wiki tree with root S_0.
- Subfolders of S_0 are S_{00} and S_{01}.
- Subfolders of S_{00} are labeled S_{000} and S_{001} .

Setup. Meta roles and Barrier role.

- $M_1(S_0) = \langle 1, 0, 0, 0 \rangle$
- $M_1(S_{00}) = \langle 0, 1, 1, 0 \rangle$
- $M_1(S_{000}) = \langle 1, 0, 0, 0 \rangle$
- $M_2(S_0) = \langle 0, 1, 0, 1 \rangle$
- $M_2(S_{00}) = \langle 1, 0, 0, 0 \rangle$
- $M_2(S_{000}) = \langle 0, 0, 1, 0 \rangle$
- $B(S_{000}) = \langle 1, 0, 1, 1 \rangle$

Setup. User and its roles R.

- User U_i with Roles R_1 R_2.

Result. Computing the roles without the Barrier role. When calculating without the Barrier role, the vectors are simply added as described on (7.6).

$$R_1(U_i)(S_{x^3}) = \bigvee_{i=1}^{3} M_1(S_{x^i})$$
$$= \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \vee \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \vee \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

$$R_2(U_i)(S_{x^3}) = \bigvee_{i=1}^{3} M_2(S_{x^i})$$
$$= \begin{pmatrix}0\\1\\0\\1\end{pmatrix} \vee \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \vee \begin{pmatrix}0\\0\\1\\0\end{pmatrix} = \begin{pmatrix}1\\1\\1\\1\end{pmatrix}$$

and therefore the permissions that are set to one are:

$P(U_i)(S_{x^3}) = \{p_x | x \in \{$Folder View, Folder Edit, Folder History, Folder Remove$\}\}$

Result. Computing the roles with the Barrier role. The Barrier role is always considered in the computation. The roles are computed using the formula seen on Paragraph 7.7.1. The setup and the result is illustrated in Figure 7.3.

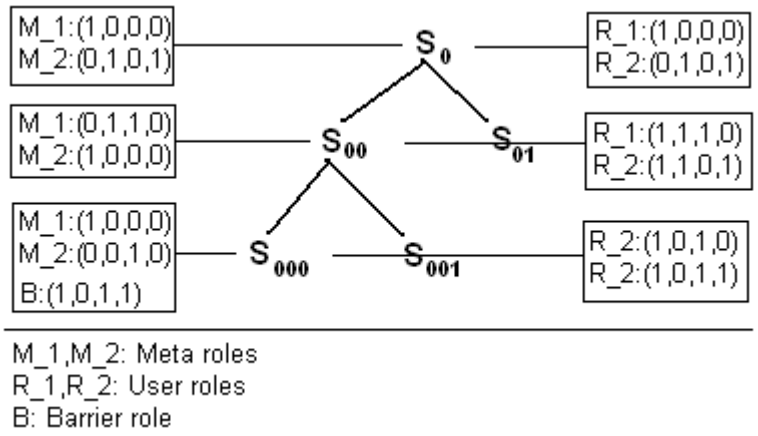


Figure 7.3: Wiki tree with Meta roles, Barrier role and the computed user roles.

Computation for Role $R_1(U_i)(S_{0^3})$:

$$
\begin{aligned}
R_1(U_i)(S_0) &= M_1(S_0) = \langle 1,0,0,0\rangle \\
R_1(U_i)(S_{0^2}) &= M_1(S_{0^2}) \vee f(\mathbf{B(U_i)(S_{0^2})}, \mathbf{R_1(U_i)(S_0)}) \\
&= \langle 0,1,1,0\rangle \vee \underbrace{f(\langle \mathbf{1},\mathbf{1},\mathbf{1},\mathbf{1}\rangle, \langle \mathbf{1},\mathbf{0},\mathbf{0},\mathbf{0}\rangle)}_{\langle 1,0,0,0\rangle} \\
&= \langle 0,1,1,0\rangle \vee \langle 1,0,0,0\rangle = \langle 1,1,1,0\rangle \\
R_1(U_i)(S_{0^3}) &= M_1(S_{0^3}) \vee f(\mathbf{B(U_i)(S_{0^3})}, \mathbf{R_1(U_i)(S_{0^2})}) \\
&= \langle 1,0,0,0\rangle \vee \underbrace{f(\langle \mathbf{1},\mathbf{0},\mathbf{1},\mathbf{1}\rangle, \langle \mathbf{1},\mathbf{1},\mathbf{1},\mathbf{0}\rangle)}_{\langle 1,0,1,0\rangle} \\
&= \langle 1,0,0,0\rangle \vee \langle 1,0,1,0\rangle = \langle 1,0,1,0\rangle
\end{aligned}
$$

Computation for Role $R_2(U_i)(S_{0^3})$:

$$
\begin{aligned}
R_2(U_i)(S_0) &= M_2(S_0) = \langle 0,1,0,1\rangle \\
R_2(U_i)(S_{0^2}) &= M_2(S_{0^2}) \vee f(\mathbf{B(U_i)(S_{0^2})}, \mathbf{R_2(U_i)(S_0)}) \\
&= \langle 1,0,0,0\rangle \vee \underbrace{f(\langle \mathbf{1},\mathbf{1},\mathbf{1},\mathbf{1}\rangle, \langle \mathbf{0},\mathbf{1},\mathbf{0},\mathbf{1}\rangle)}_{\langle 0,1,0,1\rangle} \\
&= \langle 1,0,0,0\rangle \vee \langle 0,1,0,1\rangle = \langle 1,1,0,1\rangle \\
R_2(U_i)(S_{0^3}) &= M_2(S_{0^3}) \vee f(\mathbf{B(U_i)(S_{0^3})}, \mathbf{R_2(U_i)(S_{0^2})}) \\
&= \langle 0,0,1,0\rangle \vee \underbrace{f(\langle \mathbf{1},\mathbf{0},\mathbf{1},\mathbf{1}\rangle, \langle \mathbf{1},\mathbf{1},\mathbf{0},\mathbf{1}\rangle)}_{\langle 1,0,0,1\rangle} \\
&= \langle 0,0,1,0\rangle \vee \langle 1,0,0,1\rangle = \langle 1,0,1,1\rangle
\end{aligned}
$$

and therefore the permissions that are set to one are:

$$P(U_i)(S_{x^3}) = \{p_x | x \in \{\text{Folder View, Folder History, Folder Remove}\}\}$$

Summary. This chapter provided a formal description of the SmallWiki Extended Security Model. We explained the most important elements of the Wiki tree and its security components in a technical way. The rules of updating and modifying roles were shown in a compact way.

The formulas being introduced in this chapter represent the model of SmallWiki Extended Security Model. These formulas are not used in the Smallwiki implementation. *E.g.*, in order to handle a set of permissions in the implementation we use *collections* instead of vectors with a constant size.

Chapter 8

Conclusion

8.1 Summary

Security is an important issue for collaborative web environments. We introduced Wiki with its *open editing concept* and listed the problems such as vandalism. The SmallWiki Default Security Model provided an easy and powerful system in order to manage the users and their privileges. If there is more than one administrator, they are likely to abuse their privileges as seen in Section 4.4.3. Therefore we introduced the SmallWiki Extended Security Model that fulfils the pattern of safe delegation (see Section 5.1.4) with several sector administrators. This approach was validated by security scenarios.

Additionally we provided a formal description of the SmallWiki Extended Security Model, a description of the user interface, relevant design aspects accompanied by uml diagrams, and installation and configuration instructions for SmallWiki.

8.2 Future Work

In this Section we present some additional ideas and where possible extensions of SmallWiki could lie.

IP banning to avoid Vandalism. Some Wiki engines allow banning individual users from editing, which can be accomplished by banning their particular IP address or their username, if they are using one. But many Internet Service Providers assign a new IP address for each login, so that IP bans can often be circumvented relatively easily.

To deal with the problem of changing IP addresses, we can use and extended *self-expiring bans* to all IP addresses in a particular range, thereby ensuring that the

vandal cannot edit pages within a given timeframe, the underlying assumption being that this is often sufficient as a deterrent.

Accounting for Web Services. Most Wiki engines do not supply any logging mechanism. Therefore it is not possible to provide the billing of web services. To implement web services with cost, we need some sort of user *authentication*, *authorization*, and *accounting* (*AAA*) [25]. This *AAA* technology fulfils the needs for providing restricted access to the protected resources, and for billing the offered services.

User authentication and authorization is already realised in the SmallWiki server. But there is still the lack of supporting *user accounting*.

Topic Maps. Topic maps [24] are a new ISO standard for describing knowledge structures and associating them with information resources. Topic maps are also destined to provide powerful new ways of navigating large and interconnected corpora. They enable multiple, concurrent views of sets of information objects. The structural nature of these views is unconstrained; they may reflect an object oriented approach, or they may be relational, hierarchical, ordered, unordered, or any combination of the foregoing. Moreover, an unlimited number of topic maps may be overlaid on a given set of information resources.

Topic maps can be used:

- To qualify the content and/or data contained in information objects as topics to enable navigational tools such as indexes, cross-references, citation systems, or glossaries.
- To link topics together in such a way as to enable navigation between them. This capability can be used for virtual document assembly, and for creating thesaurus-like interfaces to corpora, knowledge bases, etc.
- To filter an information set to create views adapted to specific users or purposes. For example, such filtering can aid in the management of multilingual documents, management of access modes depending on security criteria, delivery of partial views depending on user profiles and/ or knowledge domains, etc.
- To structure unstructured information objects, or to facilitate the creation of topic-oriented user interfaces that provide the effect of merging unstructured information bases with structured ones. The overlay mechanism of topic maps can be considered as a kind of external markup mechanism, in the sense that an arbitrary structure is imposed on the information without altering its original form.

Appendix A

Glossary

The list of the terms used in this diploma is given as follows:

- **AAA.** Authentication, authorization and accounting. A system in IP-based networking to control what computer resources users have access to and to keep track of the activity of users over a network.
- **Accounting.** The process of keeping track of a user's activity while accessing the network resources, including the amount of time spent in the network, the services accessed while there and the amount of data transferred during the session. Accounting data is used for trend analysis, capacity planning, billing, auditing and cost allocation.
- **Admin permission.** The permissions *Folder Admin*, *Page Admin* and *Resource Admin*.
- **Admin role.** A role with an admin permission.
- **Administrator.** A user with an admin role.
- **Anonymous role.** A default role of SmallWiki with name *anonymous* and permissions to *view*, *edit* and to *add* content.
- **Anonymous user.** A Wiki user with the role *anonymous*.
- **Apache.** Apache is a freely available Web server that is distributed under an *open source* license. Apache has been the most popular web server on the Internet since April of 1996.
- **Aquisition.** Combination of inheritance and redefinition: the permissions from the parent role are added to the child role. This mechanism is used in the SmallWiki Extended Security Model.

- **Authentication.** The process of identifying an individual, usually based on a username and password. Authentication merely ensures that the individual is who he or she claims to be, but says nothing about the access rights of the individual.

- **Authorization.** The process of granting or denying a user access to network resources once the user has been authenticated through the username and password. The amount of information and the amount of services the user has access to depend on the user's authorization level.

- **CMS.** Content Management System. Software that enables one to add and/or manipulate content on a Web site.

- **Collaboration.** The act of combining the efforts of several parties. These may include systems used for programming, writing among each others.

- **Delegation.** The act of delegating, or investing with authority to act for another.

- **Folder.** A Folder is a node of the Wiki tree. The folder groups a number of children, *i.e.*, it can contain other structures.

- **FTP.** File Transfer Protocol. The protocol used on the Internet for exchanging files. FTP works in the same way as HTTP for transferring Web pages from a server to a user's browser and SMTP for transferring electronic mail across the Internet in that, like these technologies, FTP uses the Internet's TCP/IP protocols to enable data transfer.

- **HTML.** HyperText Markup Language. The authoring language used to create documents on the World Wide Web. HTML is similar to SGML, although it is not a strict subset.

- **HTTP.** HyperText Transfer Protocol. The underlying protocol used by the World Wide Web. HTTP defines how messages are formatted and transmitted, and what actions Web servers and browsers should take in response to various commands. For example, when you enter a URL in your browser, this actually sends an HTTP command to the Web server directing it to fetch and transmit the requested Web page.

- **Inheritance of a role.** A role gets its set of permissions from a top role by some sort of inheritance, *i.e.*, a child role gets its permissions from the top role.

- **IP.** Internet Protocol. IP specifies the format of packets, also called datagrams, and the addressing scheme. Most networks combine IP with a higher-level protocol called Transmission Control Protocol (TCP), which establishes a virtual connection between a destination and a source.

- **ISO.** International Organization for Standardization. A voluntary, nontreaty organisation founded in 1946, responsible for creating international standards in many areas, including computers and communications. Its members are the national standards organisations of 89 countries, including the American National Standards Institute.
- **Mime-type.** Multipurpose Internet Mail Extensions. A specification for formatting non-ASCII messages so that they can be sent over the Internet. Many e-mail clients now support MIME, which enables them to send and receive graphics, audio, and video files via the Internet mail system. In addition, MIME supports messages in character sets other than ASCII.
- **MVC.** Model View Controller. A way of partitioning the design of interactive software. The *model* is the internal workings of the program (the algorithms), the *view* is how the user sees the state of the model and the *controller* is how the user changes the state or provides input.
- **Newsgroup.** An on-line discussion group. On the Internet, there are literally thousands of newsgroups covering every conceivable interest. To view and post messages to a newsgroup, you need a news reader, a program that runs on your computer and connects you to a news server on the Internet.
- **P2P.** Peer-to-peer architecture. Often referred to simply as peer-to-peer, or abbreviated P2P, a type of network in which each workstation has equivalent capabilities and responsibilities. This differs from client/server architectures, in which some computers are dedicated to serving the others. Peer-to-peer networks are generally simpler, but they usually do not offer the same performance under heavy loads
- **Page.** A sole entity that contains a composite of documents modeling the contents of the page that the user entered using the Wiki syntax. A Page is a node of the Wiki tree.
- **Permission.** The individual action a user is allowed to perform in the system.
- **Pull technology.** The World Wide Web is based on a pull technology where the client browser must request a Web page before it is sent.
- **Push technology.** In client/server applications, to send data to a client without the client requesting it.
- **Redefinition of a role.** A role - that is already created at some higher level in the Wiki tree - is created again.
- **Resource.** A Resource is a node of the Wiki tree. A resource might contain any data, like images, videos, sounds, pdf or zip files. In fact it can be anything that someone wants to include with the pages, or wants to provide as a possibility to download.

- **Role.** Container of a set of permissions. A Role can be assigned to a user, and can be attached on a structure of the Wiki tree.
- **RSS.** RDF Site Summary or Rich Site Summary. An XML format for syndicating Web content. A Web site that wants to allow other sites to publish some of its content creates an RSS document and registers the document with an RSS publisher. A user that can read RSS-distributed content can use the content on a different site. Syndicated content includes such data as news feeds, events listings, news stories, headlines, project updates, excerpts from discussion forums or even corporate information.
- **Site administrator.** The user with the name *admin* and the role *administrator*. It always owns all permissions in the Wiki site.
- **SmallWiki.** A Wiki implementation written in VisualWorks Smalltalk.
- **SSH.** Secure Shell. Developed by SSH Communications Security Ltd., Secure Shell is a program to log into another computer over a network, to execute commands in a remote machine, and to move files from one machine to another. It provides strong authentication and secure communications over insecure channels. It is a replacement for rlogin, rsh, rcp, and rdist.
- **SSL.** Secure Sockets Layer. A protocol developed by Netscape for transmitting private documents via the Internet. SSL works by using a private key to encrypt data that's transferred over the SSL connection. Both Netscape Navigator and Internet Explorer support SSL, and many Web sites use the protocol to obtain confidential user information, such as credit card numbers. By convention, URLs that require an SSL connection start with https: instead of http:.
- **Structure.** The structure is the basic entity of SmallWiki. It is either a *Folder*, a *Page*, or a *Resource*.
- **Updating a role.** The mechanism of computing the role on a certain structure of the Wiki tree. This involves inheritance (on the SmallWiki Default Security Model) or aquisition (on the SmallWiki Extended Security Model).
- **Wiki tree.** The Wiki tree is a data structure accessed beginning at the root node. Each node is either a leaf or an internal node. An internal node has one or more child nodes, and is called the parent of its child nodes. All the children of the same node are siblings. Contrary to a physical tree, the root is depicted at the top of the tree, and the leaves are depicted at the bottom.
- **Wiki.** A collaborative Web site comprised of the perpetual collective work of many authors. A wiki allows anyone to edit, delete or modify content that has been placed on the Web site using a browser interface, including the work of previous authors. The term wiki refers to either the Web site or the software used to create the site.

- **WYSIWYG.** What You See Is What You Get. relating to or being a word processing system that prints the text exactly as it appears on the computer screen
- **XML.** Extensible Markup Language. A specification developed by the W3C. XML is a pared-down version of SGML, designed especially for Web documents. It allows designers to create their own customized tags, enabling the definition, transmission, validation, and interpretation of data between applications and between organizations.
- **Zope.** Z Object Publishing Environment. A free, open source Web application platform used for building high-performance, dynamic Web sites.

Appendix B

SmallWiki in a Nutshell

SmallWiki has been implemented using VisualWorks 7, which can be downloaded for free from the Cincom home-page [2]. To download the latest implementation of SmallWiki itself use Cincom Public StORE or you can have a look at the goodies directory within a current VisualWorks distribution.

B.1 Loading Into the Image

1. Load the bundle named *SmallWiki* from Cincom Public StORE or from the goodies directory into your image. You will be asked to load additional bundles like the Swazoo web-server [26] and the SmaCC parser generator [16].

2. Load the package named *Smallwiki.Admin* from Cincom Public StORE into your image. This package is required if you want to use the *security extension* with its *security user interface* described in section C and the features of *safe delegation* (see Section 5.1.4). Run the postload class method *SmallWiki.Structure migrateAction:SmallWiki.Advanced* in order to register the *Advanced* menu item into your Template. This postload method should be executed automatically when loading the package, but if the *Advanced* menu item does not show up in the *Admin* menu, re-run the method in the workspace.

3. If you like to have example extensions available, also load the bundle called *SmallWiki Examples*.

B.2 Running the Tests

1. Make sure you have the package *RBSUnitExtensions* installed.

2. Locate the package *SmallWiki Tests*, that contains all the SUnit tests of SmallWiki.
3. Select all the test-cases and click on run.
4. You should see a green bar, if all tests pass.

B.3 Starting a Server

1. While loading SmallWiki a workspace has opened automatically with useful commands to run the web-server. Read through the whole text to see more possibilities for configuration.
2. If you are in a hurry, just evaluate the first expression

   ```
   server := SmallWiki.SwazooServer startOn:  8080
   ```

   which starts the server on the port 8080.
3. Switch to your favorite web-browser and point it to

   ```
   http://localhost:8080
   ```

4. Have a look at the *Information folder* and read through those pages for important news published there.

B.4 Accessing the Admin Account

There are commands like changing the design, manage roles and users, removing pages or modifying the history that require you to log-in. A default administrator has been created during installation with the user-name *admin* and the password *smallwiki*. These default settings can be changed by using the provided workspace. There is more advanced user-interface underway to manage all the security aspects of SmallWiki.

B.5 Accessing the Admin Advanced Interface

When using the package *SmallWiki.Admin*, the *anonymous role* does not have any permissions and therefore the anonymous user is not allowed to access any resources of the wiki site. To change this, you have to log-in as *admin* and access the roles management interface by clicking on the links: *Admin* → *Advanced* → *Security* → *Roles Management* (see Figure C.1). On this screen you can set the permissions for the anonymous role (see section C.1).

B.6 Stylesheets, Images and Javascript

If you run SmallWiki on your own server, it is recommended that you download some resources and customize your local SmallWiki settings. These adjustment can be done by using the standard user interface and by using the VisualWorks class browser. These resources should be downloaded to operate at full capacity, since there is no guarantee that the development server is accessible at all times. The *style sheets* and the *javascript functions* are also stored as class comment of *SmallWiki.Memo*.

B.6.1 Adding the Stylesheets

By default, SmallWiki is using a stylesheet named *style.css*. So far, there are four *css* templates available:

- standard_black/style.css
- standard_blue/style.css
- standard_green/style.css
- standard_red/style.css

The package *SmallWiki.Admin* with its *Advanced* user interface is using additionally a separate *style sheet* file named *style_admin.css* also stored in these directories.

You can access and download all these needed files from the development server *www.iam.unibe.ch* by using a http browser. Therefore you can use the urls:

```
http://www.iam.unibe.ch/~scg/smallwiki/standard_blue/style.css
```

and

```
http://www.iam.unibe.ch/~scg/smallwiki/standard_blue/style_admin.css
```

After downloading and storing, you have to make these files accessible for your web browser. For this you can use another web server like *apache*. Another way is to use the *SmallWiki* site. Therefore you have to add the style sheet files as attachment to a structure in your site via the web interface (*Contents* → *add resource*).

The *paths* to the style sheets can be set on the user interface by following the links: *Admin* → *Stylesheets*. To access this interface you have to log-in as user *admin*.

At the end, the style sheet settings could look like this:

```
@import "http://myDomain/css/standard_blue/style.css";
@import "http://myDomain/css/standard_blue/style_admin.css";
```

B.6.2 Adding the Images

The Advanced Interface is using a set of *icons*. There is a *zip* file with all the necessary images available that you can download with your web browser. Therefore use the url:

```
http://www.iam.unibe.ch/~scg/smallwiki/images/images.zip
```

Unzip and store these images on your web server and make them accessible in the same way as the style sheets. Additionally you have to set the path to the *image directory* manually with your Visualworks class browser.

e.g.,

```
SmallWiki.Advanced class >> pathToImageDirectory
  ^'http://myDomain/images/'
```

B.6.3 Adding the Javascript

The Advanced user interface is using javascript for creating a collapsible Wiki tree (see Figure 5.2) and for showing the details of roles and users as pop window (see Figure C.7) or in a separate html layer (see Figure C.8). To make use of these features download and store the javascript file named *popup.js* in the same way you have done it with the style sheets and the images.

Download url:

```
http://www.iam.unibe.ch/~scg/smallwiki/javascript/popup.js
```

Set the path to the *javascript file* manually with your Visualworks class browser.

```
SmallWiki.Security class >> popupScriptPath
^'http://myDomain/javascript/popup.js'
```

B.7 Editing a Page

The syntax used to edit a SmallWiki page is simple and easy to remember.

Paragraphs. As carriage returns are preserved, simply add a newline to begin a new paragraph.

Headers. A line starting with !s becomes a header line.

Horizontal Line. A line starting with (underscore) becomes a horizontal line. This is often used to separate topics.

Lists. Using lines starting with #s and -s, creates a list: A block of lines, where each line starts with - is transformed into a bulleted list, where each line is an entry. A block of lines, where each line starts with # is transformed into an ordered list, where each line is an entry.

Tables. To create a table, start off the lines with | and separate the elements with |s. Each new line represents a new row of the table.

Pre-formatted. To create a pre-formatted section, begin each line with =. A pre-formatted section uses equally spaced text so that spacing is preserved.

Links. To create a link, put it between two *. There are three different types of links:

Internal Link. If the item exists in the SmallWiki (*e.g.*, `*Title of Item*`), a link to that item shows up when the page is saved. In case the item does not already exist, the link shows up with a create-button next to it; click on it to create the new item.

External Link. If the link is a valid url (*e.g.*, `*http://www.google.ch*`), a link to that external page shows up.

Mail Link. If the link is an e-mail address (*e.g.*, `*self@mail.me.com*`), a link to mail that person shows up, but it is obfuscated to prevent robots from collecting.

You can also alias all these links using >. So, you can create a link like this: `*Alias>Reference*`. The link will show up as Alias, but link to Reference. For images, the alias text will become the alternate text for the image.

HTML. Use any HTML anywhere you want. Useful HTML tags are:

To make a string bold, surround it by and .

To make a string italic, surround it by and .

To underline a text, surround it by <u> and </u>.

The Table B.1 lists all the mark-up tags and compares them to the syntax of *Squeak-Wiki* [17] and *WikiWorks* [18].

	SmallWiki	SqueakWiki	WikiWorks
Heading	!, !!, ...	!, !!, ...	!, !!, ...
Horizontal Rule	-	-	-
Numbered List	#	#	#
Bullet List	-	-	*
Table	|	|	{, |, }
Pre-formatted	=	=	<pre>
Link	*Reference*	*Reference*	[Reference]
Link Alias	*Alias>Reference*	*Alias>Reference*	[Alias>Reference]
Smalltalk Code	[Code]	n/a	n/a

Table B.1: SmallWiki Syntax compared to SqueakWiki and WikiWorks.

Appendix C

SmallWiki Management User Interface

SmallWiki provides a user interface in order to set up the security policies. The interface being described here refers to the SmallWiki Extended Security Model (see Section 5.1).

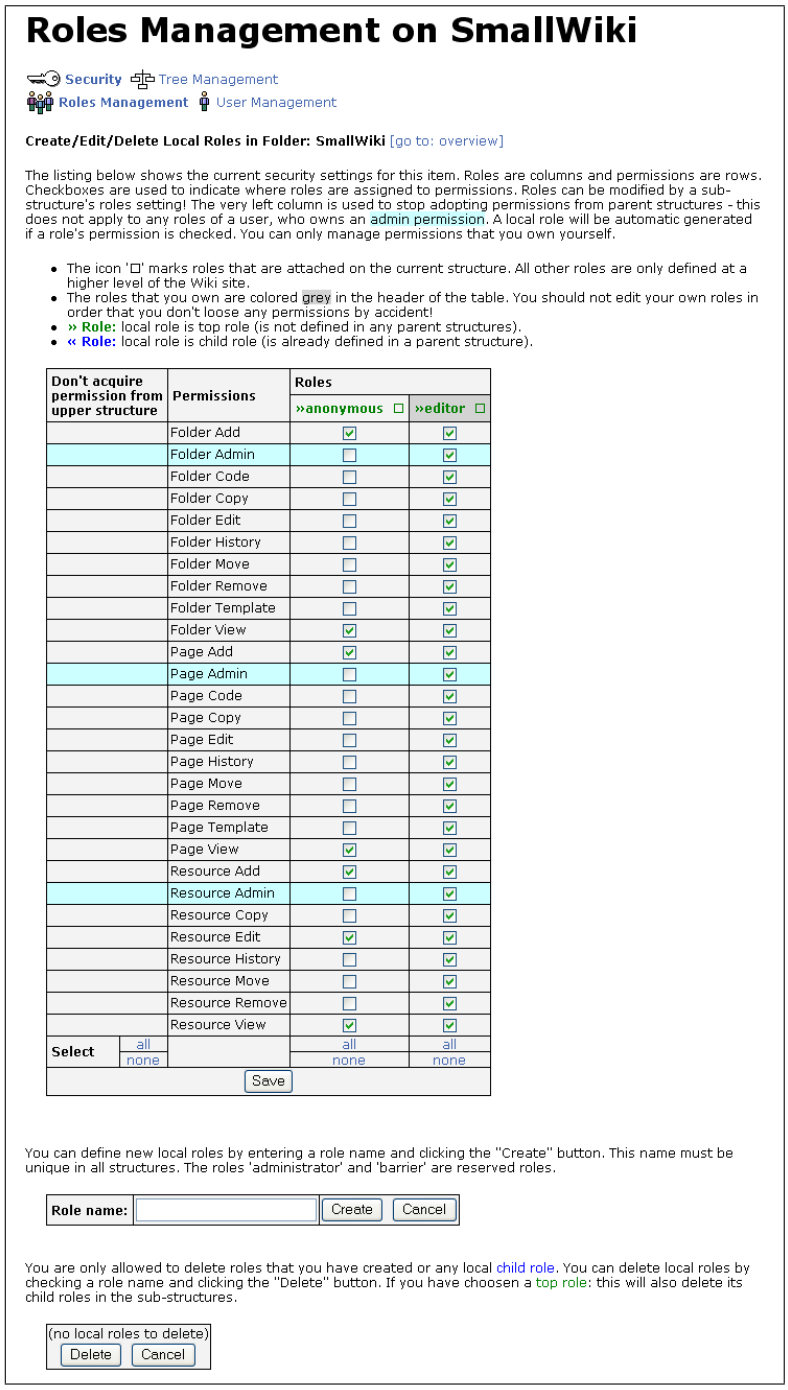
The security management is disposed in two major parts:

- **Management of Roles**. Roles can be created and deleted. Permissions can be assigned to roles (see Section C.1).
- **Management of Users**. This form is used to create new users and to change passwords. Users can be deleted and roles can be assigned to users (see Section C.2).

From both of these management forms, an *overview screen* can be visited by clicking on the link *go to: overview*. This overview provides a collapsible Wiki tree with all updated roles that are attached on the nodes of the Wiki tree (see Figure C.8), respectively lists all users with updated roles and their properties - a list of users and roles that have been created by the current user (see Figure C.15).

C.1 Management of Roles

Roles can be attached to one or several structures. In order to create and manage a role on the relevant structure, we have to make sure to visit the correct structure. From there, we get the *roles management interface* (see Figure C.1) by following the links: *Admin* → *Advanced* → *Security* → *Roles Management*. Here the following features are provided:



Roles Management on SmallWiki

Security | Tree Management
Roles Management | User Management

Create/Edit/Delete Local Roles in Folder: SmallWiki [go to: overview]

The listing below shows the current security settings for this item. Roles are columns and permissions are rows. Checkboxes are used to indicate where roles are assigned to permissions. Roles can be modified by a sub-structure's roles setting! The very left column is used to stop adopting permissions from parent structures - this does not apply to any roles of a user, who owns an admin permission. A local role will be automatic generated if a role's permission is checked. You can only manage permissions that you own yourself.

- The icon '□' marks roles that are attached on the current structure. All other roles are only defined at a higher level of the Wiki site.
- The roles that you own are colored grey in the header of the table. You should not edit your own roles in order that you don't loose any permissions by accident!
- **» Role:** local role is top role (is not defined in any parent structures).
- **« Role:** local role is child role (is already defined in a parent structure).

Don't acquire permission from upper structure	Permissions	Roles	
		»anonymous □	»editor □
	Folder Add	☑	☑
	Folder Admin	☐	☑
	Folder Code	☐	☑
	Folder Copy	☐	☑
	Folder Edit	☐	☑
	Folder History	☐	☑
	Folder Move	☐	☑
	Folder Remove	☐	☑
	Folder Template	☐	☑
	Folder View	☑	☑
	Page Add	☑	☑
	Page Admin	☐	☑
	Page Code	☐	☑
	Page Copy	☐	☑
	Page Edit	☐	☑
	Page History	☐	☑
	Page Move	☐	☑
	Page Remove	☐	☑
	Page Template	☐	☑
	Page View	☑	☑
	Resource Add	☑	☑
	Resource Admin	☐	☑
	Resource Copy	☐	☑
	Resource Edit	☑	☑
	Resource History	☐	☑
	Resource Move	☐	☑
	Resource Remove	☐	☑
	Resource View	☑	☑
Select all none		all none	all none
Save			

You can define new local roles by entering a role name and clicking the "Create" button. This name must be unique in all structures. The roles 'administrator' and 'barrier' are reserved roles.

Role name: [ ] Create Cancel

You are only allowed to delete roles that you have created or any local child role. You can delete local roles by checking a role name and clicking the "Delete" button. If you have choosen a top role: this will also delete its child roles in the sub-structures.

(no local roles to delete)
Delete Cancel

Figure C.1: Roles management interface.

Adding a Role (see Figure C.2).

- We enter the name of the role in the input field. The roles *administrator* and *barrier* are reserved role names.
- To create the new role we click the *create button*.

Figure C.2: Adding role *anonymous* to a structure.

The new role will appear on the policy overview table on the next screen (see Figure C.3). This new role has no permissions assigned yet.

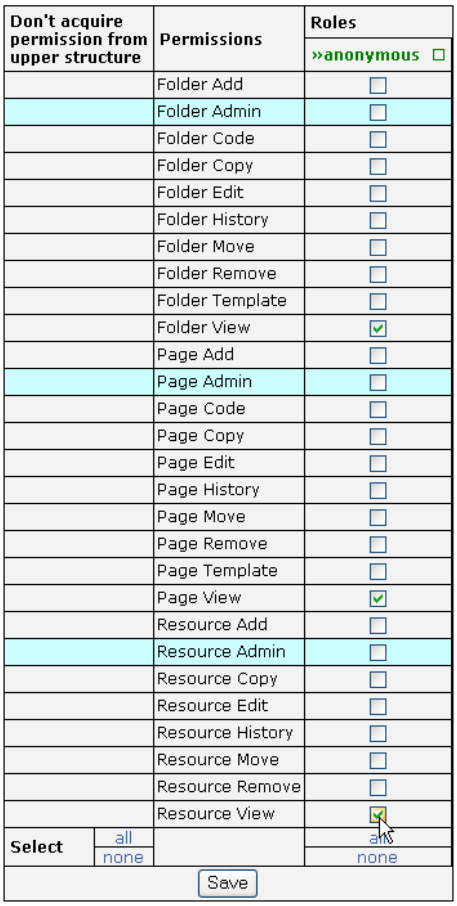


Figure C.3: Changing role properties.

Changing Role Properties (see Figure C.3).

- We select or de-select the check boxes in the policy table in order to add or remove certain permissions from a role. Roles are columns and permissions are rows. Check boxes are used to indicate where roles are assigned to permissions.
- We click the *save button* to save the new settings.

The new settings will be visible on the next screen. We are only allowed to manage permissions that we own ourselves.

Using a Barrier role (see Figure C.4).

- We select or de-select the check boxes on the very left in the policy table in order to disable or enable acquisition (see description of delegation on Section 5.1.4) to all roles for the selected permissions. This column will not show up in the root structure, since acquisition at the root top level does not make sense.
- We click the *save button* to save the settings.

Viewing Role Properties (see Figure C.3).

- By analysing the settings on the policy table we can figure out which privileges are assigned to the local roles. These settings are local and might not match with the settings that the roles get by the dynamic mechanism of acquisition (see Section 5.1).
- To view the dynamically updated roles we click on the link *go to: overview*. On the next screen we can see a collapsible Wiki tree with all the updated roles for the corresponding nodes of the tree (see Figure C.8). The actual set of permissions of a role might depend on the Barrier role and on the parent role. SmallWiki has to update the role for a given node in the tree.

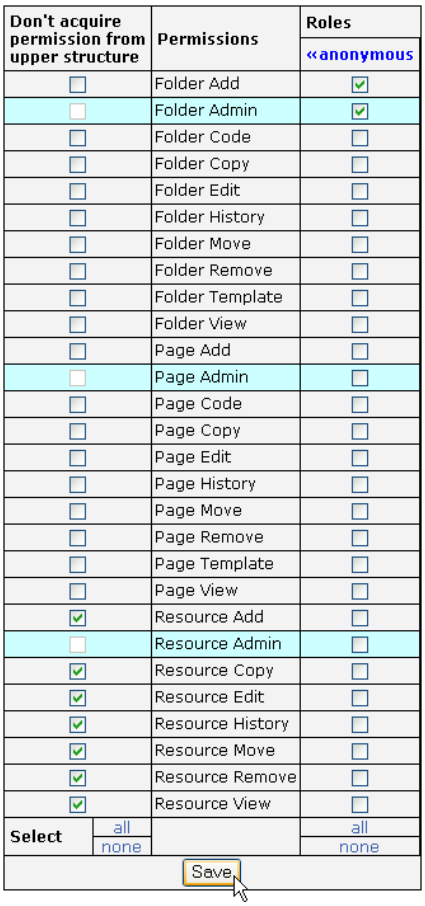


Figure C.4: Using a Barrier role to enable and disable acquisition to all roles for the selected permissions.

Deleting Roles (see Figure C.5).

- We click the check box of the roles to be deleted to select them.
- To delete the selected roles we click the *delete button.*

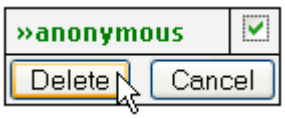


Figure C.5: Deleting roles.

We are only allowed to delete roles that we have created or any local child role - a role that also exists in an upper node of the Wiki tree (see Section 4.3). If we have chosen a top role: this will also delete its child roles in the sub-structures.

Overview of All Updated Roles from the Current Structure on.

- We click on the link *go to: overview*.
- On the next screen, a collapsible Wiki tree with the current structure as closed root folder appears. This closed structure folder is visible as clickable icon (folder/page/resource).
- We click on the icon of the structure in order to list the updated roles of that specific structure and to open the folder (see Figure C.6). By clicking on the popup-icon next to the a role, a popup window with the permissions of the role will appear (see Figure C.7). By clicking on the arrow-icon, the permissions will be shown as a html layer (see Figure C.8). We have to make sure to enable Javascript in our browser settings.

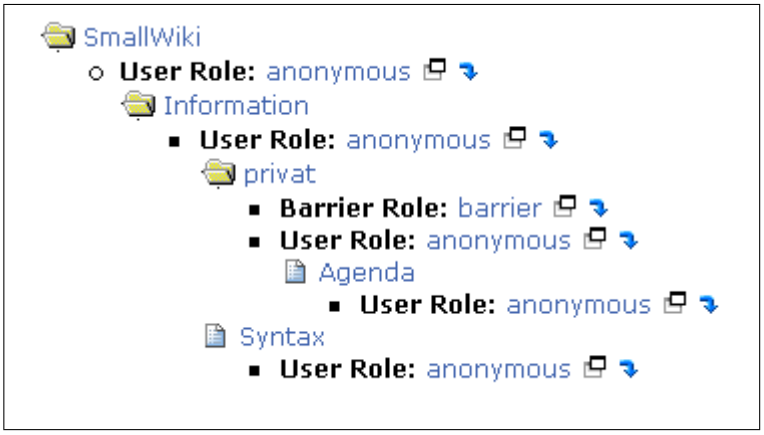


Figure C.6: Collapsible Wiki tree with nodes and updated appended roles.

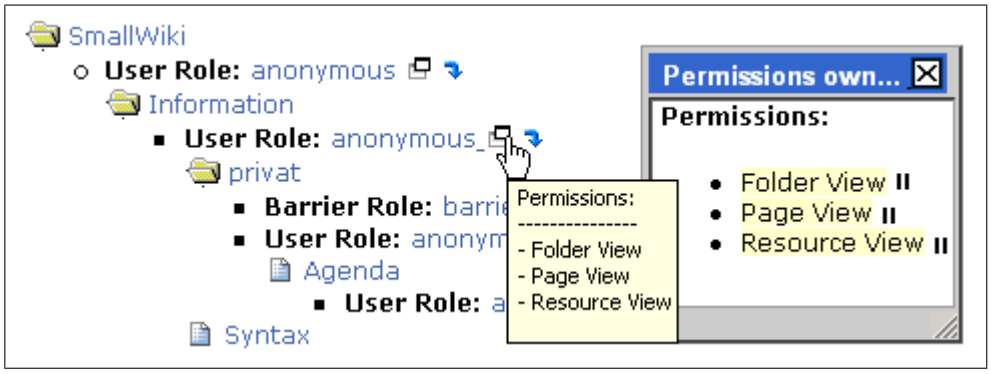


Figure C.7: View appended roles details as popup.

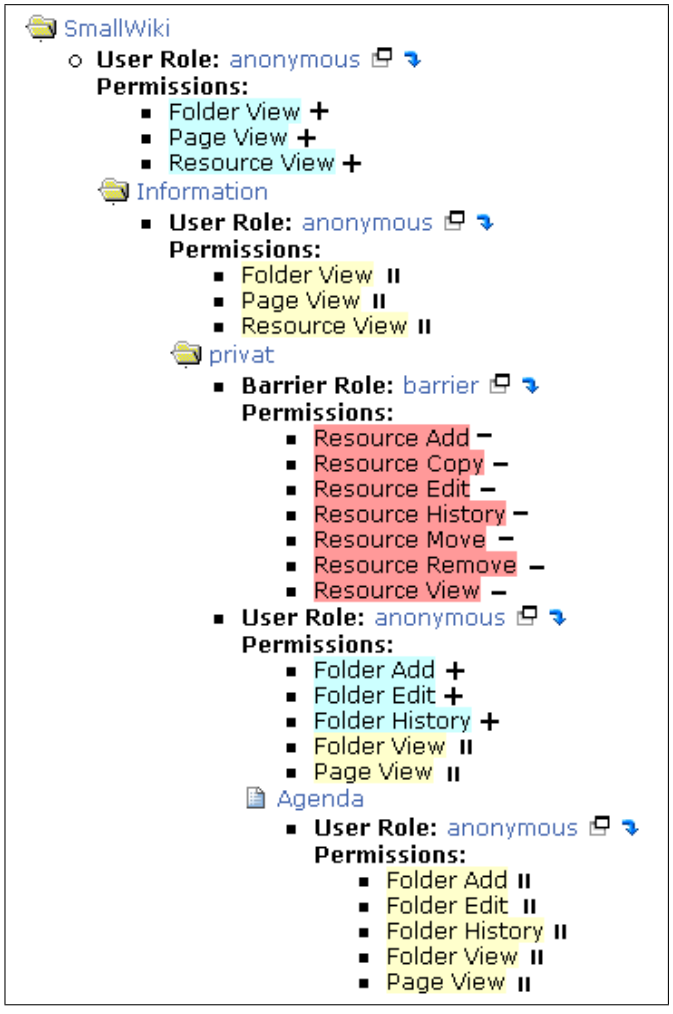


Figure C.8: View appended roles details as html layer.

C.2 Management of Users

We are also allowed to create users. Users are valid through the entire WikiWeb. In order to create a new user, we visit the relevant *users management interface* (see Figure C.9) by following the links: *Admin* → *Advanced* → *Security* → *Users Management*. Here the following features are provided:

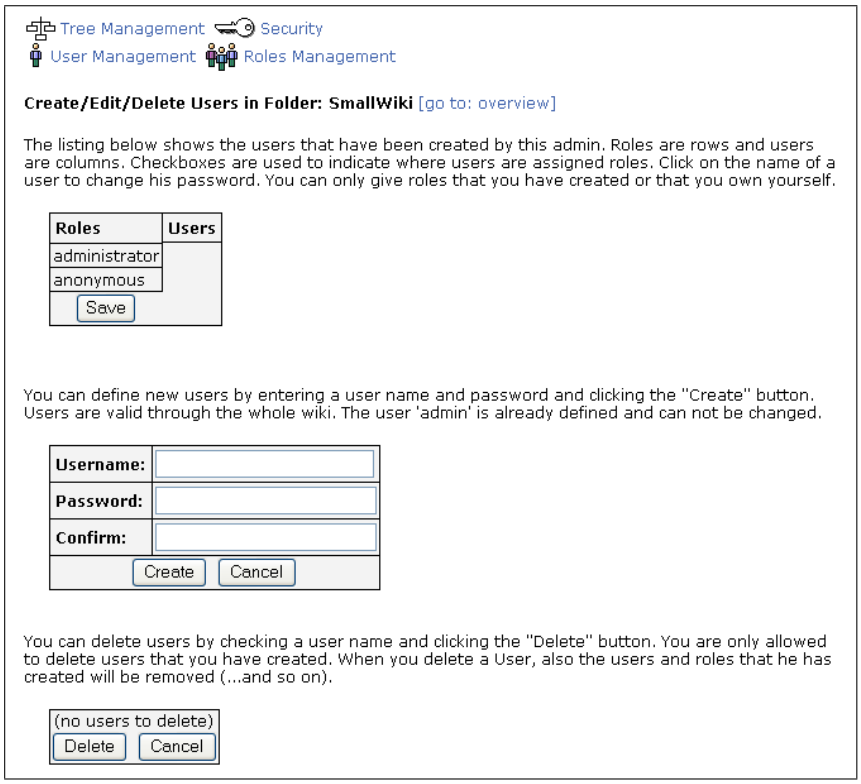


Figure C.9: Users management interface.

Adding a User (see Figure C.10).

- We enter a username in the name field. The user *admin* is already defined and can not be changed. The username can contain letters, spaces, and numbers and it is case sensitive. We choose a password for the new user and we enter it in the password and confirm fields.
- To create the new user we click the *create button*.

The new user will appear on the policy table (see Figure C.11) of the next screen and has no roles assigned yet.

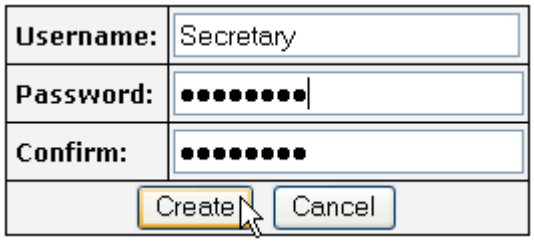


Figure C.10: Adding user *Secretary.*

Changing the Password of a User (see Figure C.10).

- We write the name and the password of the user into the corresponding form fields.
- We click the *create button*.

We can only change the password of users that we have created ourselves, unless we are the *site administrator* that has no restrictions.

Assigning Roles to Users (see Figure C.11).

- We select or de-select the check boxes in the policy table in order to add or remove certain roles from a user.
- We click the *Save button* to save the new settings.

We can only assign roles that we have created or that we own ourselves. We can give these roles only to users that we have created ourselves. We should also give the role *anonymous* to every user to make sure, that the users have at least the permissions of a user that does not have an account at all.

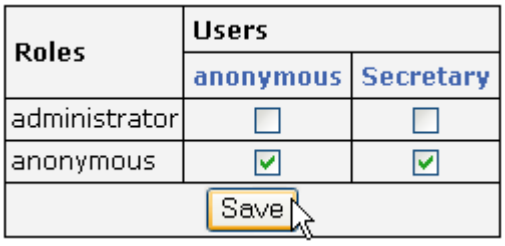


Figure C.11: Assigning roles to users.

Deleting Users (see Figure C.12).

- We click the check box to the right of the users to be deleted to select them.
- We click the *delete button* to delete the selected users.

We are only allowed to delete users that we have created. When we delete a user, also the users and the roles that we have created will be removed.

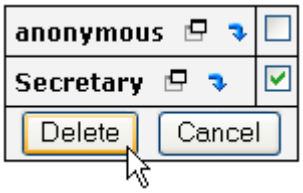


Figure C.12: Delete user *Secretary*.

Viewing User Properties and Updated Roles.

- We can view the roles that a user owns on the policy table (see Figure C.11).
- In order to view the updated roles of the users, we click on the link *go to: overview*. On the next screen we can see the user details:
  - Users and roles: a list of all users and their roles (see Figure C.13).
  - We click on a role to view its updated permissions (see Figure C.14).
  - We click on a user in order to view the details: (see Figure C.15).
    - Roles created: a list with the roles that this *administrator* has created.
    - Users created: a list with the users that this *administrator* has created so far.

Figure C.13: Overview over all users and their roles.

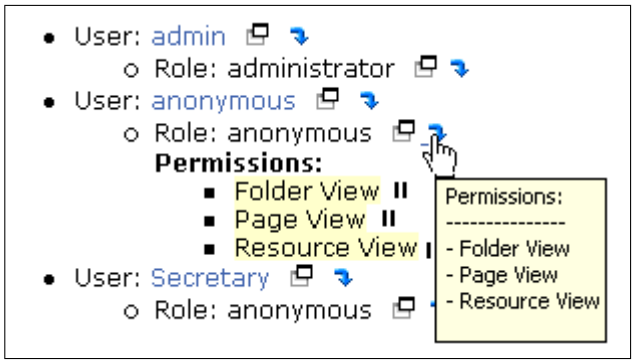


Figure C.14: View updated permissions of the role *anonymous*.

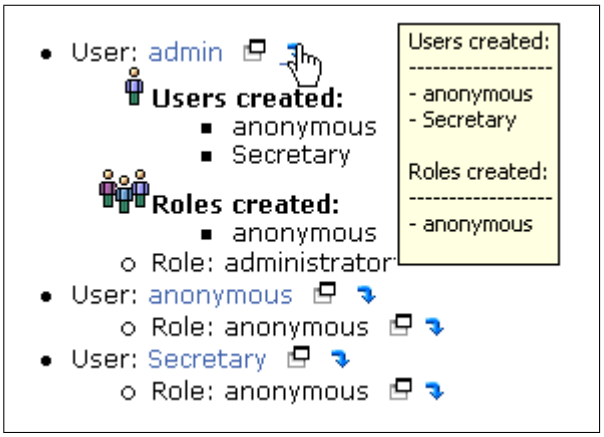


Figure C.15: View users and roles that the site administrator named *admin* has created.

Appendix D

SmallWiki Relevant Design Aspect in Detail

This chapter is an extension of Chapter 3, *i.e.*, we list the methods and provide each with a short description.

The most widely used design patterns used in SmallWiki are the Composite [15] and the Visitor [15] patterns.

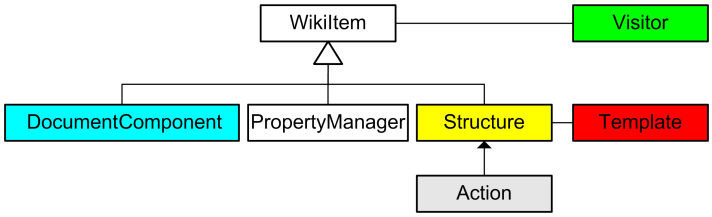


Figure D.1: Core Design

All the classes seen in Figure D.1 are abstract. Their concrete subclasses will be discussed in the following subsections. The subclasses of `WikiItem` represent the model in the MVC (*Model View Controller*) paradigm and might be visited using subclasses of the `Visitor` hierarchy. As all the rendering is done within different visitors, this part can be seen as the *view*. At last we have the controller, represented by the hierarchy below the `Action` class. Actions are used to do modifications on the model and to start the different visitors to generate the appropriate views.

D.1 Server

The basic serving is done with the chain-of-responsibilities design pattern [15] in the serving protocol. Incoming requests are passed to the first possible candidate

that is able to handle it. The request is analysed and processed within this structure and if necessary processed or passed to one of its children (see Figure D.4).

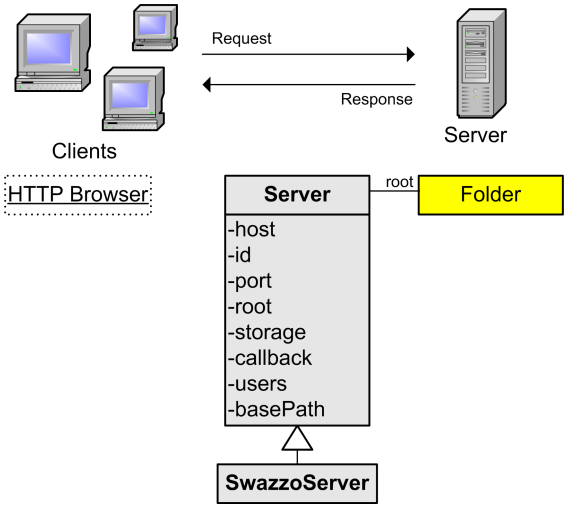


Figure D.2: Server Setup

The server class has been designed to be subclassed and to provide a common interface to different server implementations (see Figure D.2). A server might get started using the messages `#start, #startOn:, #startOn:host:ip:` or by simply instantiating using the message `#new`, configuring and starting manually. The server is not a singleton, so there might be multiple instances running within the same image.

```
server := SwazooServer startOn:8080.
```

The instance variable *root* represents the root-entity of the Wiki tree, which is usually a folder. When starting a new server, a default configuration will be created. The write-accessor for the root on a running Wiki should not be called accidentally, as all the subentries will be destroyed immediately without the possibility of going back. In order to have a look at the model of the Wiki, the following expression can be evaluated:

```
server root inspect.
```

The default server has no automatic storage mechanism assigned; this is basically useful when developing for SmallWiki and saving the image manually. When using the Wiki in a production environment, a working storage-strategy should be assigned and tests should be done extensively [1].

[1] If someone develops other storage strategies, he should let us know as we are interested to integrate them into the main-distribution.

- `server storage:ImageStorage new`
  fast and secure persistence.
- `server storage:nil`
  no persistence.

The responsibility to pass the request to the root node of the Wiki is taken by the server. The exceptions are caught and displayed as a stack-dump on the client side (see Figure D.3). The link *Open Debugger* can be used to open the debugger in VisualWorks within the context that caused the error and thus investigate the problem further.

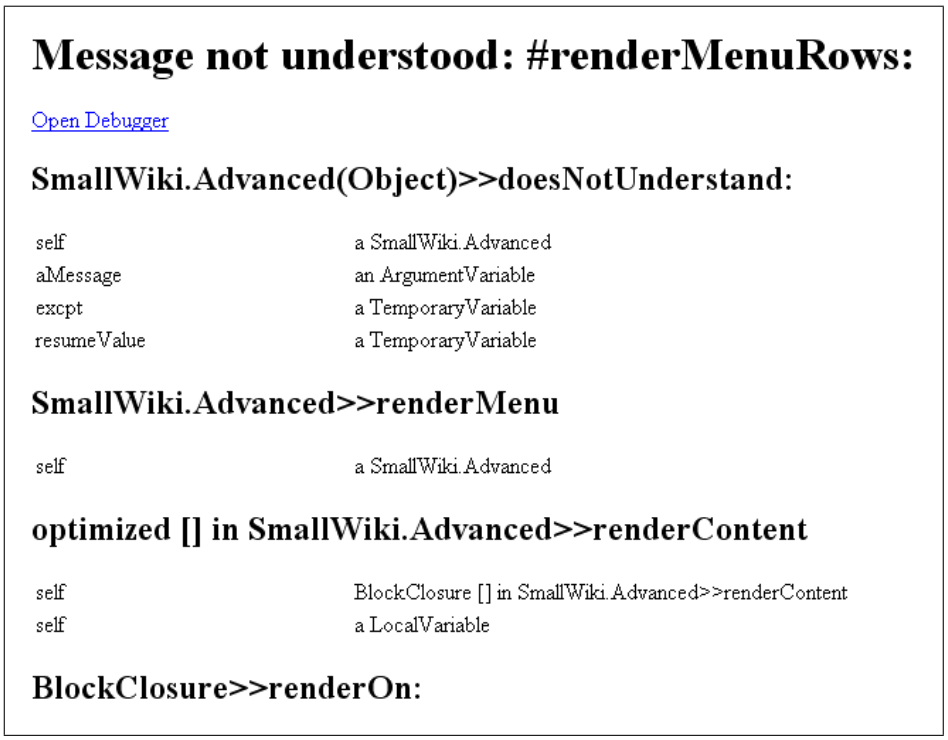


Figure D.3: Stack Dump in the Web-Browser

accessing-users

- `Server>>userAdd:anUser`
  Add a new user to the receiver. Any user with the same user name will be overridden.
- `Server>>userAddAll:aUserCollection`
  Add a collection of new users to the receiver. Any user with the same user name will be overridden.

- `Server>>userAt:aString`
  Return the user with aString as name, if there is no such user the default anonymous user is returned.

- `Server>>userAt:aString ifAbsent:anExceptionBlock`
  Return the user with aString as name, if there is no such user anException-Block is evaluated.

- `Server>>userAnonymous`
  Return the user with name anonymous.

- `Server>>userIncludes:aString`
  Return true if the receiver has got a user with the given user name.

- `Server>>userRemove:aString`
  Remove the user with the given user name from the receiver.

- `Server>>users`
  Return a collection of all users.

- `Server>>usersWithoutMainAdmin`
  Return a collection of users without the main administrator.

- `Server>>roles`
  Return a set of roles of all the users.

configuration

- `Server>>defaultRoot`
  Return the default Wiki that will be used when setting up a new server.

- `Server>>defaultBasePath`
  Return the default base path used in the html page.

- `Server>>defaultCallbackCache`
  Return the default callback cache.

serving

- `Server>>isServing`
  Return true if the receivers web-server is up and running.

- `Server>>restart`
  Restart the web-server of the receiver.

- `Server>>start`
  Start the web-server of the receiver.

- `Server>>stop`
  Stop the web-server of the receiver.

serving-private

- `Server>>process:aRequest`
  Start the chain of responsibilities in the root of the Wiki. Any unhandled exceptions thrown while processing the request will be displayed as a stack-trace within the browser of the client.

- `Server>>startUp`
  Start the server of the receiver if not already running.

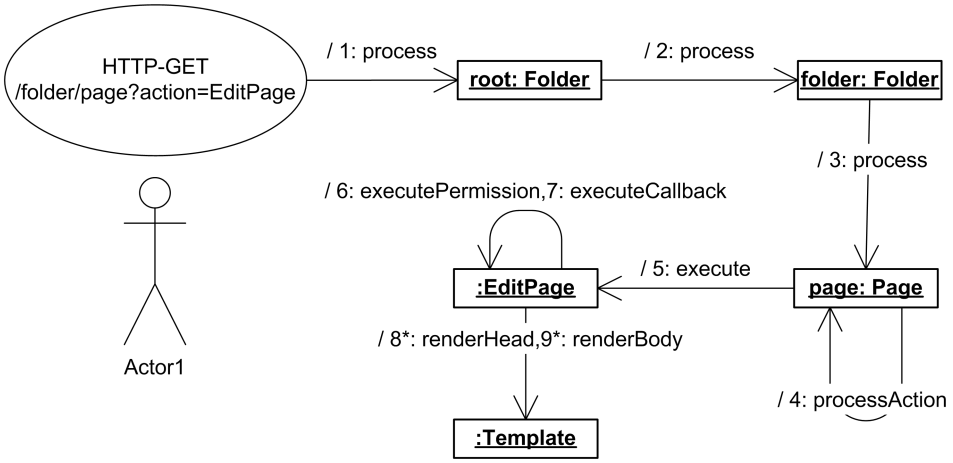


Figure D.4: Chain of Responsibility: Content Serving

In order to edit the page *page* contained in the folder *folder* a user enters an URL such as

```
http://www.smallwiki.org/folder/page?action=EditPage
```

Then the following steps, as seen in the collaboration diagram in Figure D.4, are taken:

1. The web-server gets the request emitted by the client and starts the look-up process by delegating it to the root folder.

2. Because the target is not the root chapter itself, the request is delegated to a the folder called *folder*.

3. As in the previous step, in this folder the request is delegated at the page called *page*.

4. There is no-one else that could be interested in this request, it is therefore processed by extracting the parameters and determining the action that should be executed. In this example the class `PageEdit` will be instantiated, initialised and the message `#execute` will be sent.

5. The action first checks the permissions of the user and evaluates the callbacks, see page 129 for further information.

6. Then the action asks all the template-components to emit their html-header and their html-body, see see page 144 for further information.

D.2 SecurityInformation

The abstract class `SecurityInformation` represents the security-information in the system (see Figure D.5). Its responsibility is to check if the current user has a certain permission. There is also the possibility to assert the presence of a Permission in the current session. If no such permission is present, an *UnauthorizedError* is thrown and an error page will be rendered instead of the one of the current action.

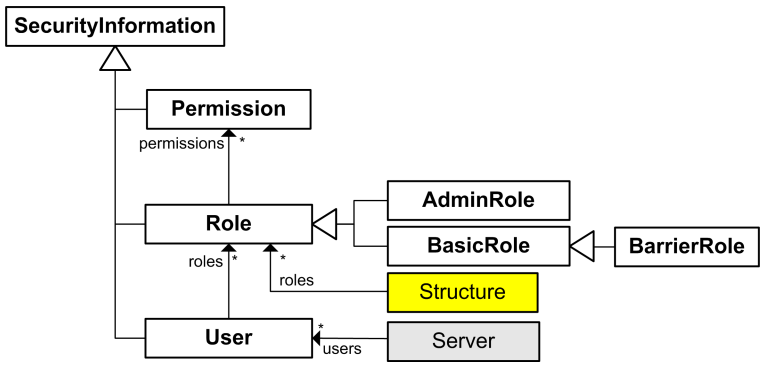


Figure D.5: The Security Hierarchy

testing

- `SecurityInformation>>assertPermission:aPermission`
  Assert that aPermission is valid in the receiver, throws a UnauthorizedError if the permission is missing.

D.3 Permission

A permission represents a privilege in the system and is the basic entity for the permission management. A permission will be granted if it is equal to the given permission. For a list of the permissions available in SmallWiki we refer to Table 4.1.

A permission will be usually used in conjunction with `User>>hasPermission:aPermission` or `User>>assertPermission`.

accessing

- `Permission>>name`
  Return the name of the receiver.
- `Permission>>permissionListItemAsHtmlTextComparedWith: oldPermissions fromBarrier:isBarrierRole`
  Return a html string with the receivers information (compare the receiver with the permission of the oldPermissions collection and highlight the difference).

copying

- `Permission>>copy`
  Return the receiver, because permissions should be unique.
- `Permission>>shallowCopy`
  Also return the receiver, because permissions should be unique.

testing

- `Permission>>hasPermission:aPermission`
  Return true if receiver equals aPermission.
- `Permission>>isAdmin`
  Return true if receiver is an admin permission.

D.4 Role

Multiple permissions might get assigned to any role. A role can be assigned to a user and can be attached to a structure. The roles on the structures are important in order to change/update the roles of a user that is visiting this structure. A role grants a certain permission if this permission is present in the set of the permission of the role. For a list of the roles available in SmallWiki we refer to Table 4.2.

accessing-permissions

- `Role>>permissionsAsAltText`
  Return the permissions of the receiver as proper alt text. unfortunately usable only for IE.
- `Role>>permissionsAsHtmlTextComparedWith:oldRole`
  Return a string with the permissions of the receiver compared with a certain updated role (compare the permissions of the receiver with the old role and highlight the difference, also add div admin, to make sure that css works in popups)

testing

- `Role>>hasPermissionAdmin`
  Return true if the receiver has an admin permission.
- `Role>>isBarrier`
  Return true if the receiver is barrier role (used for stopping inheriting permissions from parent structure). Return false.

D.4.1 AdminRole

The AdminRole is used in conjuction with the *Main Admin user* and grants any permission in the system: the message `Role>>hasPermission:aPermission` always returns true.

accessing

- `AdminRole>>permissions`
  Return all permissions found in any structure.
- `AdminRole>>name`
  Return string *administrator*.

testing

- `AdminRole>>hasPermission:aPermission`
  Always return true, since this Role is the main admin role.

D.4.2 BasicRole

This role is relevant for all users but the *Main Admin user*. Permission will be granted if any of the permission of the role grants the permission.

accessing-permissions

- `BasicRole>>add:aPermission`
  Add aPermission to the receivers permission collection.

- `BasicRole>>addAll:aCollection`
  Add a collection of permissions to the receivers permission collection.

- `BasicRole>>do:aBlock`
  Evaluate aBlock on the receivers permissions.

- `BasicRole>>remove:aPermission`
  Remove aPermission from the receivers permission collection.

- `BasicRole>>removeAll:aSet`
  Remove a set of permission from the receivers permission collection.

copying

- `BasicRole>>postCopy`
  Make sure to copy also the permissions of the receiver.

reflecting

- `BasicRole>>title`
  Return *Admin Role* if the receiver has an admin permission, otherwise return *User Role*.

D.4.3 BarrierRole

The *BarrierRole*[2] is used to stop adopting permissions from parent structures - this does not apply to any roles of a user, who owns an *admin permission*, *e.g.*, if the BarrierRole owns the *permissions x*, then none of the roles on the same structure will adopt this *permissions x* from their parents' structure roles (see Section 5.1.2).

reflecting

- `BarrierRole>>title`
  Return *Barrier Role* as title of the receiver.

[2] used only in the SmallWiki Extended Security Model

testing

- `BarrierRole>>isBarrier`
  Return true if the receiver is barrier role. Return true.

D.5 User

Multiple roles might be assigned to any user. Permission will be granted if any of the roles grants the permission.

accessing

- `User>>roles`
  Return the roles collection of the receiver.

- `User>>username`
  Return the username of the receiver.

- `User>>createdRoles`
  Return collection of createdRoles of the receiver.

- `User>>createdUsers`
  Return collection of createdUsers of the receiver.

accessing-roles

- `User>>add:aRole`
  Add aRole to the receiver. If a role with the same name is already in the collection: replace it with the new one.

- `User>>addAll:aCollection`
  Add a collection of roles to the receiver. If a role with the same name is already in the collection: replace it with the new one.

- `User>>remove:aRole`
  Remove aRole to the receiver.

- `User>>do:aBlock`
  Evaluate aBlock on the receivers roles.

- `User>>rolesDo:aBlock`
  Make sure that roles are not nil, then evaluate aBlock on the receivers roles.

accessing-createdRolesUsers

- `User>>createdRolesAdd:aRole`
  Add aRole to the createdRoles collection of the receiver.
- `User>>createdRolesRemove:aRole`
  Remove aRole from the createdRoles collection of the receiver.
- `User>>createdRolesAsAltText`
  Return a string with proper alt text with the createdRoles collection.
- `User>>createdRolesAsHtmlText`
  Return a string with proper html text with the createdRoles collection.
- `User>>createdUsersAdd:aRole`
  Add aRole to the createdUsers collection of the receiver.
- `User>>createdUsersRemove:aRole`
  Remove aRole from the createdUsers collection of the receiver.
- `User>>createdUsersAsAltText`
  Return a string with proper alt text with the createdUsers collection.
- `User>>createdUsersAsHtmlText`
  Return a string with proper html text with the createdUsers collection.

testing

- `User>>hasPermission:aPermission`
  Return true if the receiver owns aPermission.
- `User>>hasRole:aRole`
  Return true if the receiver owns aRole.
- `User>>isAnonymous`
  Return true if the receiver is the anonymous user (username=anonymous).
- `User>>validatePassword:aString`
  Return true if password of receivers equals aString.
- `User>>rolesIncludes:aRole`
  Return true if aRole is contained in the receivers roles collection.
- `User>>isAdmin`
  Return true if the receiver has an *admin permission*.
- `User>>isMainAdmin`
  Return true if the receiver has the *AdminRole*.

copying

- `User>>postCopy`
  Make sure to also copy roles, createdRoles and createdUsers of the receiver.

update

- `User>>clearRoles`
  Clear the roles of the receiver from all its permissions (add roles with the same name and no permissions; do not do anything with main admin user, since we do not have to manage any role stuff for him).

- `User>>clearRolesFromBarrierPermissions:aRolesCollection`
  Return a copy of the receiver with aCollection of roles where certain permissions have been removed.

- `User>>updateRole:currentStructureRole`
  Return a copy of the receiver with currentStructureRole updated (if the receiver owns a role with the currentStructureRole name: make a copy of the receiver and add the permissions of currentStructureRole to the receivers role with same name; keep also its permission).

- `User>>updateRoles:aCollection`
  Return a copy of the receiver with aCollection of roles updated (make sure to remove the permissions in BarrierRole from the receivers's role - except for a receiver with an admin permission; do not update anything for the main administrator).

- `User>>updateRolesFromRootUntill:aStructure`
  Return a copy of the receiver with its updated roles from root 'till current structure (first make sure to empty the roles before starting).

D.6 Structure

The structure is the basic entity of SmallWiki, representing the model of a single page. A structure is identified by exactly one URL and is usually included in a composite-tree of other structures (see Figure D.6). The three concrete subclasses of `Structure` are: `Page` and `Resource` as components and the `Folder` as composite. In fact `Structure` should not only be the subclass of `WikiItem`, but also of `Model`. As the visiting aspect, however, is far more important, the messages provided in Model have been copied from this system class.

A structure provides basic navigational accessors to its parents, children and siblings in the Wiki tree. The basic serving is done with the chain-of-responsibilities

design-pattern in the serving protocol. The resolving protocol provides messages to look up other structure items using their name.

All the structures have a title, a back-reference to their parent, and might contain user-defined properties, *i.e.*, something like a dictionary containing symbols as keys and any other objects as values. Structures are versioned automatically using a reference pointing to the previous version of the same page. The message `#postCopy` should be overridden to make it work correctly, since some subclasses of `Structure` have a dictionary with objects whereas others do not.

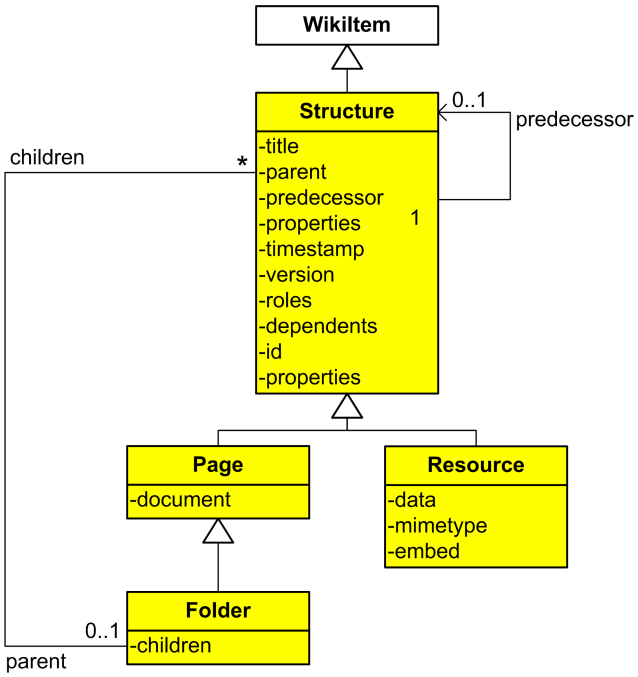


Figure D.6: Structure Composite

accessing

- `Structure>>id`
  Return the id of the receiver, that is a string build from the title of the structure. The id is used to identify a structure within its parent and therefore has to be unique.

- `Structure>>parent`
  Return the parent of the receiver. In the case the receiver is the root node nil is returned.

- `Structure>>predecessor`
  Return the previous version in the history of the receiver. If there is no history

information available nil is returned.

- `Structure>>roles`
  Return a collection of roles which are applied when the receiver processes a query. All the roles of the current user are replaced with the corresponding ones returned by this message.

- `Structure>>timestamp`
  Return a timestamp with the latest modification-time of the receiver.

- `Structure>>title:aString`
  Set the title of the receiver. If the receiver is a child of another structure the title is checked to be unique. In case of a problem, a `DuplicatedStructure` exception is thrown.

- `Structure>>version`
  Return the version-number of the receiver, the numbering starts with version 0.

accessing-calculated

- `Structure>>parents`
  Return an ordered collection with all the structures from the root up to and including the receiver of the message.

- `Structure>>root`
  Return the root node of the receiver.

- `Structure>>url`
  Return a unix-path string representing the URL of the receiver. The URL is unique within a Wiki tree and contains the id of the receiver and all its parents ids, except for the root.

- `Structure>>versions`
  Return an ordered-collection containing the receiver and all its older versions.

- `Structure>>references`
  Return a collection of all references to the receiver.

- `Structure>>referencesStartingAt: aStructure`
  Return a collection of references to the receiver starting at certain structure.

accessing-navigation

- `Structure>>first`
  Return the first node of the receiver's parent.
- `Structure>>last`
  Return the last node of the receiver's parent.
- `Structure>>next`
  Return the next node of the receiver's parent.
- `Structure>>previous`
  Return the previous node of the receiver's parent.

configuration

- `Structure>>defaultAddTarget`
  Return the target where to put new children. The default implementation returns the parent of the receiver, but structures that contain children usually want to return self.

copying

- `Structure>>postCopy`
  Copy a selection of the instance-variables and update the timestamp of the receiver. Subclasses should override this message to do a deep-copy of their data and call super. This is the key-message to make the versioning mechanism work properly.

properties-inherited

- `Structure>>properties`
  Return a property manager including the values of all inherited properties. Changes to the returned object does not change the receivers property manager.
- `Structure>>propertyAt:aKey`
  Return the value of the property of the receiver with aKey. If there is no such property defined, the look-up is retried in the parent of the receiver.
- `Structure>>propertyAt:aKey ifAbsent:anExceptionBlock`
  Return the value of the property of the receiver with aKey. If there is no such property defined, the look-up is retried in the parent of the receiver. If no such property could be found, anExceptionBlock is evaluated.

- `Structure>>propertyAt:aKey put:aValue`
  Set the property aKey to aValue in the receiver. This message does the same as `#localPropertyAt:put:` and is here simply for convenience.

properties-local

- `Structure>>localProperties`
  Return the properties of the receiver.

- `Structure>>localPropertyAt:aKey`
  Return the value of the property of the receiver with aKey. If there is no such property, nil is returned.

- `Structure>>localPropertyAt:aKey ifAbsent:anExceptionBlock`
  Return the value of the property of the receiver with aKey. If there is no such property, anExceptionBlock is evaluated.

- `Structure>>localPropertyAt:aKey put:aValue`
  Set the property aKey to aValue in the receiver.

resolving

- `Structure>>privateResolveChild:aString`
  As we usually do not have children, return nil. Subclasses with children might want to override this message.

- `Structure>>privateResolvePath:aCollection`
  Resolve a whole path starting at the receiver. If there is an error matching the path, nil is returned.

- `Structure>>resolveTo:aStringPath`
  Start the resolving-process at the receiver with the resolving-algorithm depending on the count of identifiers in aPathString:

  1. if the first character of the path is a separator the look-up is started in the root-node and processed downwards.
  2. if the first character is not a separator ...
     (a) and an empty path is given, the receiver is returned.
     (b) and a path with exactly one entry is given, a child with that name is looked for.
     (c) and a path with one or more entries is given, a look-up is started in the parent of the receiver.

The first matching structure is returned or nil if no appropriate item could be located in the tree. To see a bunch of examples about the use of this message, have a look at the tests in protocol `testing-resolving` of the class `StructureTests`.

serving

- `Structure>>process:aRequest`
  Process a basic request. First the security information contained in the request is updated, then it is decided if the request should be handled by the current component or one of its children.

- `Structure>>processAction:anAction`
  Executes the action on myself and catch basic errors.

- `Structure>>processChild:aRequest`
  The default structure has got no children, therefore we process a not-found message.

- `Structure>>processSecurity:aRequest`
  Update the roles of the current user according to the current role configuration. See `#updateRoles:` in the User class for additional information.

- `Structure>>processSelf:aRequest`
  Look for an action that might be executed on the current structure. If there is no action given, the default one is executed.

testing

- `Structure>>isComposite`
  Return true if the structure has got the possibility to hold children.

- `Structure>>isEmbedded`
  Return true if the structure should be embedded into documents referencing the receiver.

- `Structure>>isRoot`
  Answer whether the receiver is the root node. The root is a folder by default and the only structure having no parent in the Wiki tree.

- `Structure>>isSuccessor`
  Answer whether the receiver has got a previous version in the history.

- `Structure>>hasParent`
  Answer whether the receiver has a parent.

versions

- `Structure>>nextVersion`
  Copy the receiver to be used in the history and return the receiver.
- `Structure>>nextVersionBecome:aStructure`
  This message creates a copy of the receiver and puts aStructure into the history. References pointing to the receiver will be still valid, as the current version stays the same object all the time. Right now the new version must be of the same class than the receiver, else an exception is thrown.
- `Structure>>versionNumber:anInteger`
  Return the version anInteger of the receiver or nil if not present.
- `Structure>>versionRestore:anInteger`
  Restore the version anInteger. In other words: the version anInteger will become the current one, but all the other modifications are still kept in history.
- `Structure>>versionRevert:anInteger`
  Revert to the version anInteger in history. All the newer modifications will be lost.
- `Structure>>versionTruncate:anInteger`
  Truncate all the history information behind the version anInteger.

D.6.1 Resource

A resource might contain any data, like images, videos, sound, pdf or zip files. In fact it can be anything that you want to include within your pages or you want to provide as a possibility to download.

The MIME-TYPE (*Multipurpose Internet Mail Extensions Type*, a specification for formatting non-ASCII data so that it can be sent over the Internet) of the data is used to determine how the given resource should be rendered. As an example images and videos should be displayed inside the html document, whereas zip-files are only references as a link to allow the user to download the file.

testing

- `Resource>>isApplication`
  Return true if the mimetype of the receiver is application-data. This message will match types like: application/octet-stream, application/oda, application/postscript, application/zip, application/pdf, etc.
- `Resource>>isAudio`
  Return true if the mimetype of the receiver is audio-data. This message will match types like: audio/basic, audio/tone, audio/mpeg, etc.

- `Resource>>isEmbedded`
  Return true if the resource of the receiver should be embedded into the desired context. Return false if the resource should be simply linked.

- `Resource>>isImage`
  Return true if the mimetype of the receiver is image-data. This message will match types like: image/jpeg, image/gif, image/png, image/tiff, etc.

- `Resource>>isText`
  Return true if the mimetype of the receiver is text-data. This message will match types like: text/plain, text/html, text/sgml, text/css, text/xml, text/richtext, etc.

- `Resource>>isVideo`
  Return true if the mimetype of the receiver is video-data. This message will match types like: video/mpeg, video/quicktime, etc.

D.6.2 Page

A page is the most important and probably the most used class of the Structure hierarchy. As a sole entity it contains a composite of documents modeling the contents of the page that the user entered using the Wiki syntax. When initializing the instance a default document will be created to make the user aware of the newly created page.

accessing

- `Page>>document`
  Return the current document of the page.

configuration

- `Page>>defaultDocument`
  Return the default document used when a new page is created.

- `Page>>defaultVisitor`
  Return the default visitor used when rendering this document to html.

D.6.3 Folder

The folder groups a number of children. `Folder` is a subclass of `Page`, therefore it also contains a document that might be used to describe the contents.

accessing

- `Folder>>children`
  Return a collection of all the children of the receiver.
- `Folder>>sortedChildren`
  Return a sorted collection of all the children of the receiver.

children-accessing

- `Folder>>at:aString`
  Return child of the receiver with id aString or nil if absent.
- `Folder>>at:aString ifAbsent:aBlock`
  Return child of the receiver with id aString or nil if absent.
- `Folder>>includes:aString`
  Return true if the receiver contains a child with the id aString.
- `Folder>>uniqueTitle:aString`
  Proposes an unique name for a child within the receiver using aString base name.
- `Folder>>childrenCopyDo:aBlock`
  Copy the children before iterating.
- `Folder>>withAllChildren`
  Return receiver and all children and their children .... as collection.
- `Folder>>sortedChildrenDo:aBlock`
  Sort the children first before iterating.

children-structure

- `Folder>>add:aStructure`
  Add aStructure as a child to the receiver. A DuplicatedStructure exception is raised in case there is already a child with the same name.
- `Folder>>copy:aStructure`
  This message is basically the same as `#add:` but it creates a copy of the structure and makes sure that the title is unique within the receiver before adding.
- `Folder>>remove:aStructure`
  Remove aStructure from the list of children of the receiver. In case there is no such child an exception is raised.

configuration

- `Folder>>defaultChildrenCollection`
  When changing this message, you should modify the following messages: `#at:ifAbsent:`, `#add:`, `#remove:` and `#children`.
- `Folder>>defaultDocument`
  Return the default document used when a new folder is created.
- `Folder>>defaultAddTarget`
  Return receiver as default target.
- `Folder>>defaultChildrenListCode`
  Return default string.

navigation

- `Folder>>next:aStructure`
  Return the next node of the receiver's child aStructure.
- `Folder>>previous:aStructure`
  Return the previous node of the receiver's child aStructure.

D.7 Document

The document hierarchy describes the content of a Wiki page. It includes all the basic elements to represent a text such as paragraph, table, list, links, etc (see Figure D.7).

General notes.

- All the subclasses of `PageComponent` must overwrite the message `postCopy` to make a deep copy of all its content.
- All concrete subclasses of `PageComponent` should overwrite the message `accept:aVisitor` to visit this instance.

Remarks concerning Links.

- `MailToLink` holds a string with the e-mail address.
- `ExternalLink` holds a string with the URL. External links are also able to point to internal resources, but they do not update when target is renamed or moved.

- `InternalLink` holds a reference to a structure and do update automatically title and reference when target is renamed or moved. An internal link might point to a nonexistent structure and that will be created automatically when accessing.

When the user enters a text using the Wiki syntax it is parsed using SmaCC [16] and the abstract syntax tree is stored within the page.

As Table B.1 shows, the syntax of SmallWiki is similar to SqueakWiki or WikiWorks. Changing the grammar of the parser is no big deal, if you are more familiar with a different one and want to support that. However, as for all other parsers, it is difficult to write extensions that can be added and removed independently in order to parse new document entities.

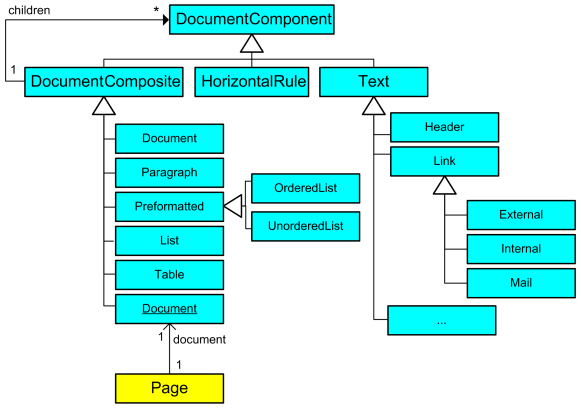


Figure D.7: The Document Hierarchy

D.8 Action

Actions are instantiated by a structure and they are initialized with that structure and the current request using the constructor method `#request:structure:`. Actions have basically two tasks: first to perform the action itself, and second to initiate or to render the GUI. Actions represent also the context in which a page is rendered as they know about their structure, the request, the response, and the security status (see Figure D.8).

Action Protocol This part of the action is used to handle the requests. The message `#execute` is called by the structure after initializing the required instance variables. It checks the security permissions of the current user, evaluates the callbacks and starts the rendering by calling `#render` on itself. The running action might use the accessors to manipulate and mediate with the current environment.

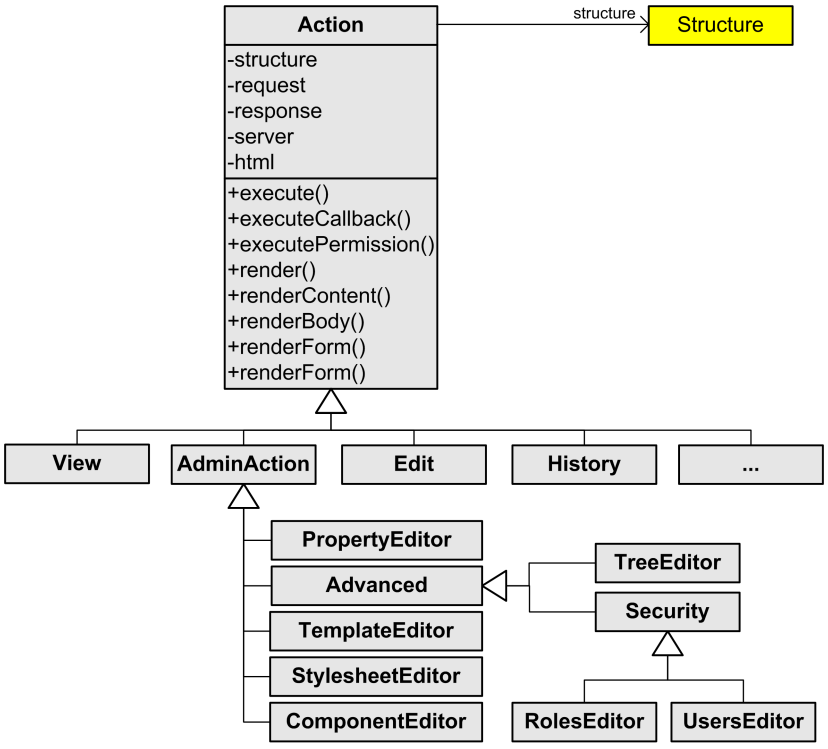


Figure D.8: The Action Hierarchy

It is usually not necessary to override the message `#execute`, the callback mechanism described in Section D.11 can be used instead.

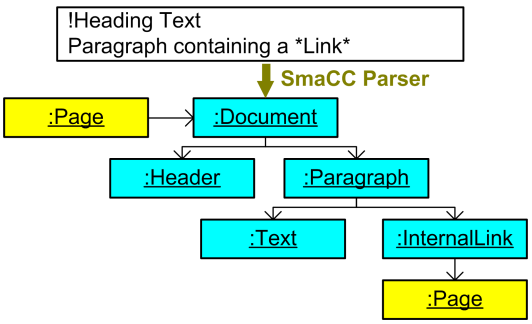


Figure D.9: Parsing a Wiki Document

Rendering Protocol The rendering process is started from the message `#render` at the end of `#execute`. The message `#render` fetches the collection of templates of the associated structure and starts generating the XHTML output. There-

fore the document is parsed (see Figure D.9). It asks each template to render the content they want to emit into the <head> ... </head> part of the output. Afterwards the body part <body>...</body> is generated and again every template might contribute its content into that part. As explained in Section D.9, there is always an instance of the class `TemplateBodyContent` available calling the message `#renderContent` of the action: by overriding this message the user-interface is rendered by the new action. The state of any component inside the rendering protocol should not be changed, as an action is unable to know when and how many times it is actually called.

accessing-heading

- `Action>>heading`
  Return the full heading of the receiver, containing the title of the receiver and the one of the structure to be handled. Do not override this message, instead have a look at `#headingAction` and `#headingStructure`.

- `Action>>headingAction`
  Return the title of the receiver to be used in the heading. Do override this message, if you want to provide something different. This might return nil, if it is not appropriate.

- `Action>>headingStructure`
  Return the title of the current structure to be used in the heading. Do override this message, if you want to provide something different. This might return nil, if it is not appropriate.

action

- `Action>>execute`
  Usually it is not necessary to override this message, instead use the provided callback mechanism. Still there are rare cases where you need to have full control over the execution process; but do not generate any output in here, use `#renderContent` instead. The rendering is called automatically, if there has not been a redirect response created while checking the permissions or while executing the callbacks.

- `Action>>executeCallback`
  Override this message to provide your own way of evaluating callbacks. This implementation only executes the anchor-callback when there are no form-callbacks being executed. This prevents from executing form and anchor callbacks at the same time by accident, what is usually not intended.

- `Action>>executePermission`
  Override this message to check permission before anything inside this action

is executed. By default an action might be used by all users, so no permissions are asserted.

rendering

- `Action>>render`
  This message starts the basic html rendering of a page. Unless you do not want the templates and the html addendum to be rendered, do not override this message.
- `Action>>renderContent`
  Override this message to customize the output of this action. Do not change the state of the component in this message.
- `Action>>renderForm`
  Renders the form. Override this message to customize the output of this action.

security

- `Action>>assertPermission:aPermission`
  Assert the presence of a Permission in the current session. If no such permission is present an `UnauthorizedError` is thrown and an error page will be rendered instead of the one of the current action.
- `Action>>hasPermission`
  Return true if the permissions is present.
- `Action>>hasRole`
  Return true if the role is present.

testing

- `Action>>isIndexable`
  Override this message and return true to tell search-engines to index the contents of this actions.

D.8.1 Advanced

The class `Advanced`[3] is a subclass of the `Admin` class. It provides the basic methods for the graphical user interface of the *security management* (see class description in Section D.8.2, Figure C.1 and Figure C.9) and of the *tree management* (see class description in Section D.8.5).

[3] used only in the SmallWiki Extended Security Model

accessing

- `Advanced>>errorMsg`
  Return the error message string. Used to give feedback on graphical user interface.

render

- `Advanced>>renderContent`
  Render the content. This will be the first menu row at this point.
- `Advanced>>renderMenu`
  Render the menu.
- `Advanced>>renderMenuRow`
  Render a menu row. This will render the menu items of the subclasses with the corresponding html icons.

renderForm

- `Advanced>>renderError`
  Render the error message. Use style sheet *error* to highlight the message.

renderInformation

- `Advanced>>renderStructureAsIconAndLinkTitle:aStructure`
  Render the structure's name with corresponding html icon and its title and its url.

utillities

- `Advanced>>handleCollapseCookieFor:aStructure`
  Deal with the cookie value that is used to open and close a node in the Wiki tree on the graphical user interface.
  Check if cookie is set:
  1. yes: remove the cookie.
  2. no: set it with key title and url, and with value open.
- `Advanced>>nodeIsOpen:aStructure`
  Return true if this node (aStructure) is open (therefore read cookie from request).

D.8.2 Security

The class `Security`[4] is a subclass of the `Advanced` class. It is the superclass of `RolesEditor` class and `UsersEditor` class. It provides methods for rendering form buttons and methods for creating strings with information about roles details. These strings will be used to render the details of the roles via Javascript as popup or as html layer.

render

- `Security>>renderContent`
  Render the menu and either the management form or an overview of the users.
- `Security>>renderMenu`
  Render the menu items of the superclass and of the subclasses.

renderButtons

- `Security>>renderTabledataWithCreateAndCancelButtons`
  Render the create and cancel form buttons with their actions.
- `Security>>renderTabledataWithCreateSaveAndCancelButtons`
  Render the create/save and cancel form buttons with their actions.
- `Security>>renderTabledataWithDeleteAndCancelButtons`
  Render the delete and cancel form buttons with their actions.
- `Security>>renderTabledataWithSaveAndCancelButtons`
  Render the save and cancel form buttons with their actions.

renderInformation

- `Security>>renderRoleHtmlInfoAsVirtualDiv:aRole comparedWith:oldRole`
  Render alt text, clickable image of role and render the details in html layer. When we click on the image, the html layer will get visible. Another click and the layer is hidden.
- `Security>>renderRoleInfoAsPopup:aRole parameterString:aString`
  Render alt text, clickable image of role and put the details in javascript popup function. When we click on image, a popup window appears with the role details inside.

[4] used only in the SmallWiki Extended Security Model

- `Security>>roleAttributesInfoStringForPopup:aRole compared:oldRole`
  Return a html string with the details of the comparison of the two roles. The difference will be marked with a css div style sheet.

utillities

- `Security>>getAdminDeleteableRoles`
  Return a collection of roles that the current user has created.
- `Security>>getAdminEditableRoles`
  Return a collection of roles that the current user has created and also his roles.
- `Security>>getRolesCollectionOfAllStructures`
  Return the roles of all structures (starting at root).
- `Security>>getRoleWithName:aRoleName`
  Returns a role with indicated name.
- `Security>>splitCollection:aCollection bySize:aSize`
  Split the collection into smaller collections and put each subcollection into returned result collection. This will be used to create a nicer html policy table. We do not want hundreds of columns next to each other.

D.8.3 UsersEditor

The class `UsersEditor`[5] is a subclass of the class `Security` and is responsible for rendering the user administrating interface and rendering the overview of all users and their updated roles on a certain structure. It is also responsible to execute the corresponding actions.

action

- `UsersEditor>>addUserToCreatedUsers:myUser`
  Add myUser to the createdUsers collection of the current user of the request.
- `UsersEditor>>changePassword`
  Change the password of the user specified in the html form if the current user is allowed to do so.
- `UsersEditor>>createAndAddUser`
  Create the new user and add it to the current user createdUsers collection.

[5] used only in the SmallWiki Extended Security Model

- `UsersEditor>>executeCreate`
  Create a user with given name and password specified in the html form.

- `UsersEditor>>executeDelete`
  Delete the specified user (has been checked on the user interface) and also its dependencies (the roles and users that he has created himself).

renderForm. In this protocol, there are methods that decide on which users the current *admin user* can execute admin methods, *e.g.*, an admin user is only allowed to manage the users that he has created himself, unless he is the *main administrator* who can manage any user.

- `UsersEditor>>renderForm`
  Render the forms. This method will generate three forms:

  1. **Policy form.** Here you can assign one or several roles to the users.
  2. **Creation form.** Create a new user or change the password of a user.
  3. **Deletion form.** Delete a user by selecting the corresponding checkbox.

- `UsersEditor>>renderFormTitle`
  Render the title of the form.

- `UsersEditor>>renderClassficationForm`
  Prepare to render the form used to assign roles to users.

- `UsersEditor>>renderClassificationTitle`
  Render some hints for the classification form.

- `UsersEditor>>renderClassficationTableUsers:users andRoles:roles`
  Render the form with the user and roles.

- `UsersEditor>>renderClassficationTableHeader:aNumberOfCols`
  Render the first header part of the classification table.

- `UsersEditor>>renderClassificationTableUsers:aUsersCollection`
  Render the second header part of the classification table containing the users.

- `UsersEditor>>renderClassificationUserCheckbox:aUser withValue:aRole`
  Render a single checkbox where aRole can be assigned to aUser.

- `UsersEditor>>renderClassficationTableCheckboxes:users with:role`
  Render all check boxes where roles can be assigned to users.

- `UsersEditor>>renderCreateForm`
  Render the form to create a new user.

- `UsersEditor>>renderCreateTitle`
  Render the title of the form.

- `UsersEditor>>renderCreateUsernameField`
  Render the form field to enter the name of the user.

- `UsersEditor>>renderCreatePasswordField`
  Render the form field to enter the password of the user.

- `UsersEditor>>renderCreateConfirmField`
  Render the form field to confirm the password of the user.

- `UsersEditor>>renderDeleteForm`
  Render the delete form.

- `UsersEditor>>renderDeleteTitle`
  Render the title of this form.

- `UsersEditor>>renderDeleteRowFor:aUser`
  Render the users with check boxes where we can select the user to be deleted.

renderInformation This protocol contains the methods that are responsible for rendering the overview of all users and their updated roles with the permissions on a certain structure.

- `UsersEditor>>renderInformation`
  Render all the user information. All the users with their updated roles and permissions will be rendered.

- `UsersEditor>>renderInformationTitle`
  Render the title of the form.

- `UsersEditor>>renderInformationText`
  Render the hint text of the form.

- `UsersEditor>>renderAllAttributesOf:aUser`
  Render all details of aUser.

- `UsersEditor>>renderUserAttributesWithPopup`
  Render the user with the details accessible in javascript popup.

- `UsersEditor>>renderUserAttributesWithVirtualDiv:aUser`
  Render the user with the details accessible via html layer.

- `UsersEditor>>renderRolesAndPermissionsAttributesOf:aUser`
  Render the roles and the permissions of aUser.
- `UsersEditor>>renderRolesWithPermissionHints:aRolesCollection`
  Render all roles with their permissions in alt text and also available as popup and as html layer.
- `UsersEditor>>userAttributesInfoString:aUser`
  Return a string with the user attributes for the virtual popup function and for virtual layer.
- `UsersEditor>>userAttributesInfoStringForPopup:aUser`
  Return a string with the user attributes and with the popup function.

utillities

- `UsersEditor>>deleteDependenciesOf:aUserCollection`
  Delete the dependencies of all users of aUserCollection.
- `UsersEditor>>deleteInAllStructures:aRolesCollection`
  Delete the roles in the whole Wiki site.
- `UsersEditor>>deleteUserWithDependenciesOf:aUser`
  Delete dependencies of aUser:
  1. Delete all roles stored in adminRoles from all structures.
  2. Delete all roles stored in adminRoles from all user's instances variables adminRoles and also from their roles collection.
  3. Delete all user stored in adminUsers.
  4. Delete all user stored in adminUsers from all user's instances variables adminUsers.
  5. Finally delete the user itself.
- `UsersEditor>>getAdminEditableUsers`
  Return collection of users that this user has created, or return all users if current user is the main administrator.
- `UsersEditor>>removeRolesCollectionDependenciesOf:aRolesCollection in:aUser`
  Remove the roles from the user and from its instance variable createdRoles.

D.8.4 RolesEditor

The class `RolesEditor` is also a subclass of the class `Security` and is responsible for rendering the roles administrating interface and for executing the corresponding actions.

accessing-roles

- `RolesEditor>>addPermission:aPermission toRole:aRole`
  If this role does not exist so far in current structure: create and append it to structure and add the given permission to it.
- `RolesEditor>>removePermission:aPermission fromRole:aRole`
  Remove a permission from a role.
- `RolesEditor>>updateRole:aRole ofStructure:aStructure`
  Update the given role and return the updated role.

action

- `RolesEditor>>addRoleToCreatedRoles:myRole`
  Add a role to the current users createdRoles collection. Make sure to update the user in both: the server and in the request. In the request, there is only a copy of the user with updated roles, but this user is used to render the form after the role is created.
- `RolesEditor>>deleteRolesFromFormOfStructure:aStructure`
  Remove a role that has been selected in the form from a structure and also from its children. Uses recursive call. If the role is a top role: also remove the role of the current user's adminRoles collection.
- `RolesEditor>>executeCreate`
  Create a new Role (self rolename) if the name is not already in use. Append it to the structure, and also append the new role to the user's adminRoles collection, make sure not to just append it to the user copy.
- `RolesEditor>>executeDelete`
  Delete some roles from a certain structure. If the role is a top role: remove it from the users createdRoles collection.
- `RolesEditor>>executeSave`
  The saving mechanism of roles with the given permissions in the form is done via callback. Just remove here the empty barrier roles and do not delete the other empty roles. The user has to do this explicit via the web-form.
- `RolesEditor>>removeRoleFromCreatedRoles:myRole`
  Make sure to remove the role from all server users instance variable adminRoles. Usually we only have to delete this role in the *current* admin adminRoles collection, but the main admin might delete a role, that somebody else has created!
- `RolesEditor>>roleValidToAppend`
  Return true if the role can be created in current structure.

renderForm As in the class `UsersEditor`, the methods in this protocol decide on which roles the current *administrator* can execute admin methods, *e.g.*, an administrator is only allowed to manage the roles and permissions that it has created itself or owns itself, unless it is the *site administrator* that can manage any role and permission.

- `RolesEditor>>renderForm`
  Render the forms. This method will generate three forms:
  1. **Policy form.** Here you can assign one or several permissions to the roles.
  2. **Creation form.** Create a new role.
  3. **Deletion form.** Delete a role by selecting the corresponding checkbox.
- `RolesEditor>>renderFormTitle`
  Render the title of the form.
- `RolesEditor>>renderPolicyForm`
  Render the policy form in order to assign permissions to roles. Make sure, that the table does not have to many columns. Use multiple tables instead.
- `RolesEditor>>renderPolicyAcquireCheckbox:permission editable:isEditable`
  Render the checkbox that allows to stop *aquisition* for a given permission.
- `RolesEditor>>renderPolicyCheckbox:aRole withValue:aPermission editable:isEditable`
  Render checkbox to add/remove a given permission to the given role. Use the id attribute for the javascript function (select all/ de-select all). Render permissions that should not be changed as disabled. Render the roles of the current user as disabled; otherwise he could remove some of his permissions by accident. Make sure to also disable the callback for disabled permissions; otherwise a user could save the form, edit it (remove the disabled tags) and send it and the callbacks would be executed.
- `RolesEditor>>renderPolicyContentWith:rolesCollection`
  Render the rows of the table, where roles meet permissions.
- `RolesEditor>>renderPolicyFormColumns:roles`
  Render the policy form for the given roles.
- `RolesEditor>>renderPolicyHeaderRows:aRolesCollection`
  Render the first two rows of the policy table.
- `RolesEditor>>renderPolicySelectAllNoneButtons: roles`
  Render two links for every role in order to select all/none permissions for a role by using javascript.

- `RolesEditor>>renderPolicySelectButtonOf:rolesCollection withMode:aMode`
  Render a link for every role in order to select permissions for a role according to the aMode argument.

- `RolesEditor>>renderPolicyTableHeader:aNumberOfCols`
  Render the header row of the policy table.

- `RolesEditor>>renderPolicyTableRoles:aRolesCollection`
  Render name of the roles into table row (color the local roles green and color the roles from a upper structure blue).

- `RolesEditor>>renderPolicyTableRow:rolesCol with:permission editable:isEditable`
  Render a row with the check boxes for a given permission and a roles collection.

- `RolesEditor>>renderRoleNameAsChildOrTop:aRole`
  Render the name of the role and consider its position in the structure.

- `RolesEditor>>renderDeleteForm`
  Render to form in order to delete the selected users.

- `RolesEditor>>renderDeleteFormTitle`
  Render the title of the delete form.

- `RolesEditor>>renderDeleteCheckboxRow:aRole`
  Render a checkbox with name of the role that could be deleted by user.

- `RolesEditor>>renderCreateForm`
  Render the form in order to create a new role.

- `RolesEditor>>renderCreateFormTitle`
  Render the title of the create form.

- `RolesEditor>>renderCreateRoleNameField`
  Render the form field to enter the name of the role.

renderInformation The methods of this protocol are responsible for rendering the updating roles according to the structure on which the roles appear. The permissions of the roles are accessible via Javascript by clicking on either the *popup icon* or on the *layer icon* that appear next to the role name.

- `RolesEditor>>renderAllRoles`
  Collect all roles in the Wiki site and render them.

- `RolesEditor>>renderAllUpdatedLocalRolesOf:aStructure`
  Update and render local roles of a structure.

- `RolesEditor>>renderAllUpdatedParentRolesOf:aStructure`
  Update and render parent roles of a structure.

- `RolesEditor>>renderAllUpdatedRolesOf:aStructure`
  Update and render parent and local roles of a structure.

- `RolesEditor>>renderInformationText`
  Render some hints for the admin.

- `RolesEditor>>renderInformationTitle`
  Render the title of the form.

- `RolesEditor>>renderLocalRolesOfSelfAndChildrenOf:aStructure`
  Render roles and permission of aStructure, then recursive call for its children.

- `RolesEditor>>renderRoles: aRolesCollection`
  Render the name of the roles of a certain roles collection.

- `RolesEditor>>renderRoleWithInfo:updatedRole compared:oldRole of:aStructure`
  Render the updated role compared with oldRole (render info as popup and also in html layer).

utillities

- `RolesEditor>>getPermissionsForPolicyTable`
  Return only permissions, that the specific admin owns himself (these are the permissions that the current admin is allowed to manage).

- `RolesEditor>>getRolesForDeleteTable`
  Return only roles from structure that the current user has created himself and also child-roles that are local added.

- `RolesEditor>>getRolesForPolicyTable`
  Return roles to render.

- `RolesEditor>>isCheckedRole:aRole with:aPermission`
  Check if aRole exists in current structure and if the role has the permission aPermission.

- `RolesEditor>>isChildRole:aRole`
  Return true, if aRole is also defined in a parent structure.

- `RolesEditor>>isTopRole:aRole`
  Return true, if aRole is not already defined in a parent structure.

D.8.5 TreeEditor

The class `TreeEditor`[6] provides methods to *cut*, *copy*, and *delete* nodes out of the Wiki tree. It still lacks the support of re-arranging the links when a structure has been moved to another place. Also the roles are still copied with the structure. In a future release, the roles should be deleted before a structure is copied. The cut, copy, and delete functions are only provided for the *site administrator* named *admin*. All other administrators can only use the `treeEditor` to browse the Wiki tree.

action

- `TreeEditor>>cutStructureFromFormOfSelfAndChildrenOf: aStructure`
  Delete the structures stored in copyCutStructures collection in the structure's tree.

- `TreeEditor>>deleteStructure:structureToDelete OfSelfAndChildrenOf:aStructure`
  Delete a certain structure out of a tree.

- `TreeEditor>>deleteStructureFromFormOfSelfAndChildrenOf: aStructure`
  Check if user is allowed to delete the selected structure. For this: update his roles according to the roles of the relevant structure. Delete one or more structures that have been selected in the form.

- `TreeEditor>>executeCutCopy`
  Check if user has permission copy. For this: update his roles according to the roles of the relevant structure. Mode is set to cut or copy. Save the collection into copyCutStructures and set selectedStructures back to nil.

- `TreeEditor>>executeDelete`
  Delete the structures selected in the form.

- `TreeEditor>>executePaste`
  If mode is set to cut, delete the structure out of the tree first. Paste structure in specified place.

- `TreeEditor>>pasteStructureOfSelfAndChildrenOf:aStructure`
  Check if the structure is selected to be pasted in. If it is selected: paste the structures from copyCutStructures collection.

[6] used only in the SmallWiki Extended Security Model

renderForm Note that the buttons to cut, paste and delete are only rendered for the main administrator.

- `TreeEditor>>highlightItem`
  Highlight the item, if it was selected in the form. Insert a style class for this item.
- `TreeEditor>>renderButtons`
  Render the buttons in order to process an action.
- `TreeEditor>>renderCheckboxOf:aStructure`
  Render the checkbox with the callback function.
- `TreeEditor>>renderForm`
  Render the form to delete, cut, copy and paste.
- `TreeEditor>>renderFormTitle`
  Render the title of the form.
- `TreeEditor>>renderHiddenModeField`
  Render a hidden input field with the mode value.
- `TreeEditor>>renderHiddenStructureField`
  Put the structure ids into hidden field.
- `TreeEditor>>renderStructureOf:aStructure`
  Render the structure with its checkbox, icon and highlight it (if it has been selected on the screen before).
- `TreeEditor>>renderStructureOfSelfAndChildrenOf:aStructure`
  Render the structure tree's items. Only open folders, that are open (the info is stored in the cookie).

utillities

- `TreeEditor>>cleanHiddenFields`
  Delete the values in the hidden field after we have pasted something, so the structure's items that have been selected before will not be highlighted anymore.

D.9 Template

Templates are used to render common parts of Wiki pages. They are defined within a collection held in the root of the Wiki and in combination with a selected Stylesheet (see Figure D.10), they provide the look-and-feel of the Wiki. As the templates are held in the property manager of the structure, they are shared within all children of a folder unless there is a new definition.

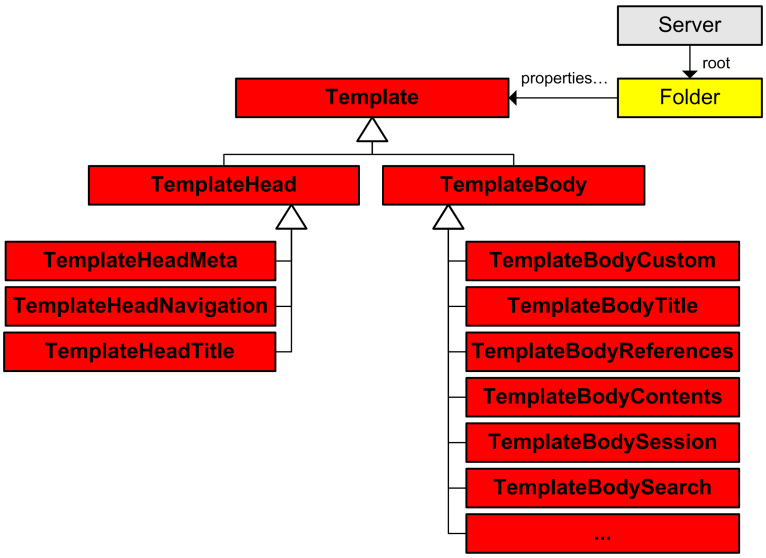


Figure D.10: The Template Hierarchy

private

- `Template>>expand:aString for:anAction`
  Expand aString within the context of anAction. This is often used to let the user specify dynamic parts within the settings of the templates. Currently the following tags are supported:
  - `%a` the title of the action
  - `%h` the host-name of the server
  - `%i` the ip-number of the server
  - `%l` the the url of the structure
  - `%m` the modification time of the structure
  - `%p` the port-number of the server
  - `%r` the title of the root structure
  - `%t` the title of the structure
  - `%u` the name of the current user
  - `%v` the version-number of the structure

rendering

- `Template>>renderBodyWith:anAction on:html`
  This message is called when the action should render its content to the body-

Figure D.11: The Template Editor

part of the resulting XHTML document. The default implementation is empty.

- `Template>>renderConfigWith:anAction on:html`
  This message is called when the configuration form of the receiver should be rendered. This message is solely called from the `TemplateEdit` action to let the user specify his settings. The default implementation is empty.

- `Template>>renderHeadWith:anAction on:html`
  This message is called when the action should render its content to the head-part of the resulting XHTML document. The default implementation is empty.

D.9.1 TemplateHead.

The class `TemplateHead` is a subclass of `Template`. It should be used for templates rendering the header of the output-file. If someone wants to render to the head and to the body, he should use `TemplateBody` as a superclass instead.

D.9.2 TemplateBody.

The class `TemplateBody` is a subclass of `Template`. It should be subclassed in most of the cases to create a new template component. The message `#title`

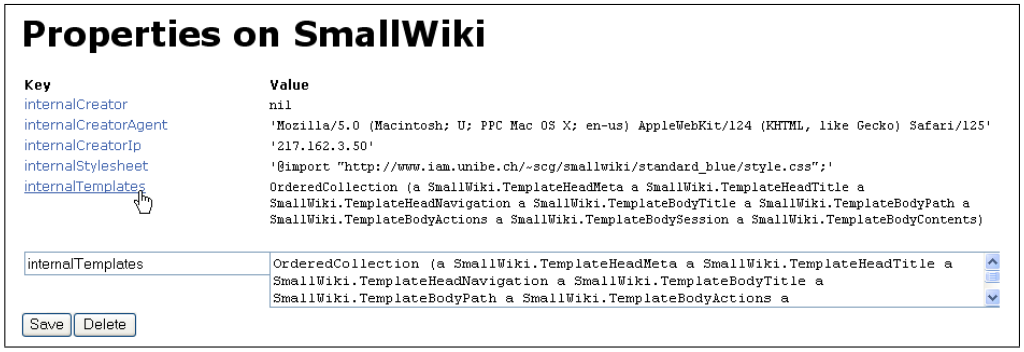


Figure D.12: The Property Editor

on the class-side should be implemented in order to return a string describing this subclass. The title is also used by default to identify the associated CSS-id and to render it into the body part. The messages `#defaultId` and `#defaultTitle` might be used to change this behavior. The user is always able to edit the id and title from within the template-editor (see Figure D.11) in the web-browser to customize the template to his needs and to the applied stylesheet. Other adjustments can be made via the property-editor (see Figure D.12) and the component editor (see Figure D.13).

rendering

- `TemplateBody>>renderBodyWith:anAction on:html`
  Override this message in all subclasses to render the body-part of the template. All the HTML rendering is done within aBlock passed to the message `#renderDivFor: on: with:` to ensure that the XHTML environment is properly set-up and that the design can be specified using css-stylesheets.

- `TemplateBody>>renderConfigWith:anAction on:html`
  If you override this message in your subclasses do not forget to call super, as there are the default properties for the title and the css-id rendered in here.

In the appendix you can find the source of the default style-sheet used to layout the components of SmallWiki. Take it as a starting point to create your own design.

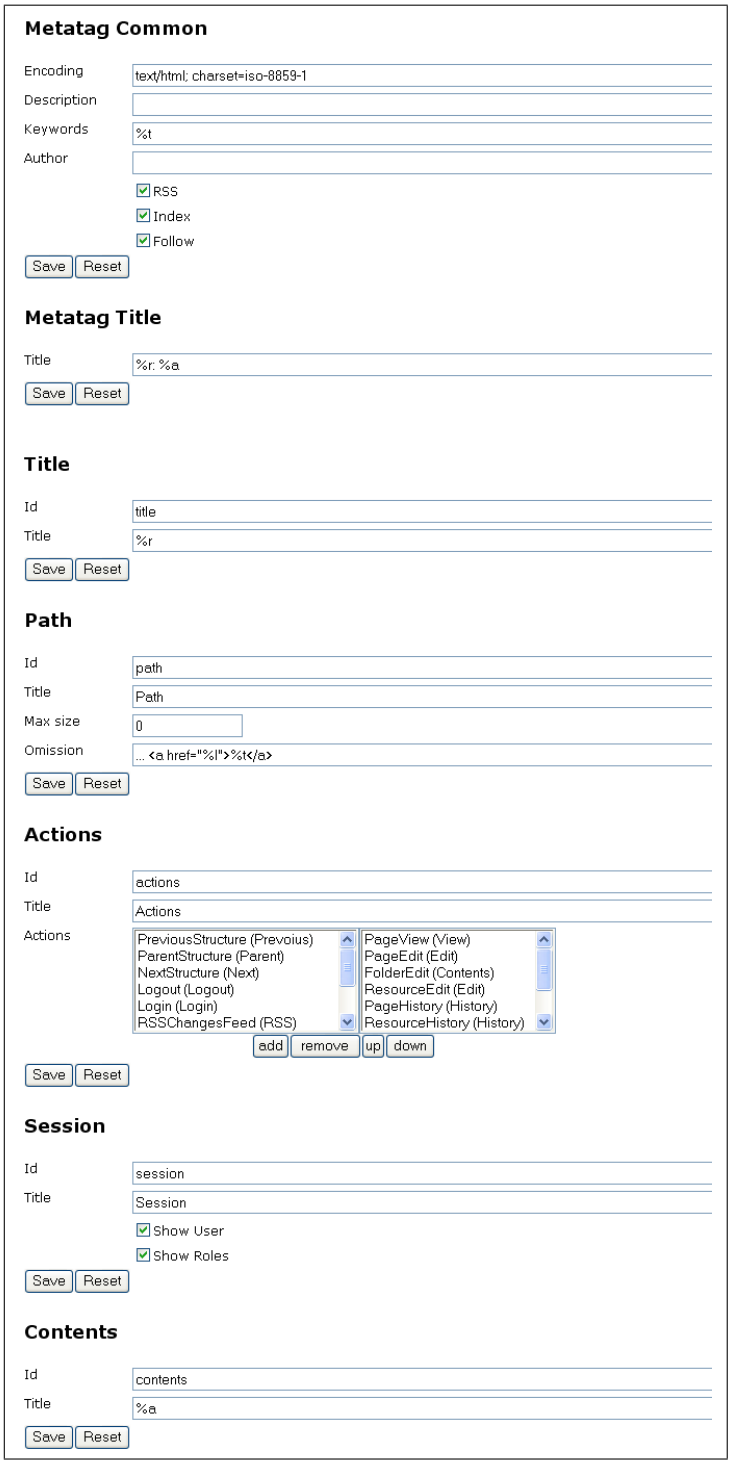


Figure D.13: The Component Editor

D.10 Storage

The abstract storage class provides a protocol to all kinds of storage mechanism implementing persistence in a Wiki. It takes care of the notification of changes. Subclasses should implement either the message `#changed` or `#changed:` to make the given structure persistent.

notification

- `Storage>>changed`
  This message is called whenever something changed inside the Wiki structure, if you need to know what exactly happened override `#changed:` instead.

- `Storage>>changed:aStructure`
  Every time a structure inside the Wiki tree changes this message is called. Override it to provide a storage mechanism.

D.10.1 Snapshot Storage

The class `SnapshotStorage` provides an interface to make snapshots of Wikis on a regular bases. With the implementation of the `ImageStorage` as a concrete implementation this is the most secure and most widely used storage mechanism (see Figure D.14).

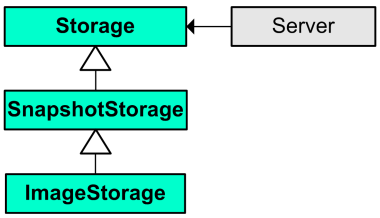


Figure D.14: The Snapshot Hierarchy

accessing

- `SnapshotStorage>>delay:aDelayInSeconds`
  Set the delay in seconds between the snapshots.

- `SnapshotStorage>>lastchange`
  Return the timestamp of the last change of the whole Wiki.

- `SnapshotStorage>>lastsnapshot`
  Return the timestamp when the last successful snapshot has been made.

snapshot

- `SnapshotStorage>>privatePostSnapshot`
  Override this message with code that should be executed after doing the actual snapshot.
- `SnapshotStorage>>privatePreSnapshot`
  Override this message with code that should be executed before doing the actual snapshot.
- `SnapshotStorage>>privateSnapshot`
  Override this message to do the actual snapshot.
- `SnapshotStorage>>snapshot`
  Do not override this message, that calls the messages `#privatePreSnapshot`, `#privateSnapshot` and `#privatePostSnapshot` in order to save the Wiki structure as a whole.

testing

- `SnapshotStorage>>isChanged`
  Return true if the Wiki has been changed since the last snapshot.
- `SnapshotStorage>>isExpired`
  Return true if the delay has been expired since the last snapshot.
- `SnapshotStorage>>isSnapshotNeeded`
  Return true if a snapshot is needed.
- `SnapshotStorage>>isThreadRunning`
  Return if the storage thread is running.

D.11 HTML and Callbacks

Creating valid XHTML is an error prone task when using string concatenation. SmallWiki follows the design of Seaside [19] and implements the class `HtmlWriteStream`. This class subclasses `WriteStream` and provides a lot of additional messages to append text and XHTML elements to the document being rendered.

The following example could be part of the message `#renderContent` within the Action hierarchy to render a simple user-interface:

```
html heading:'Title' level:1.
html paragraph:[
  html text:'Click '.
  html
    anchor:'here'
    to:self url
    callback:[ :action | action doSomething ] ]
```

The code produces something like the following output:

```
<h1>Title<h1>
 <p>
  Click
  <a href="/?action=ActionClass&callback=31415">
   here
  </a>
 </p>
```

The first message `#heading:level:` produces a simple section heading of level 1. The message `#paragraph:` is similar to the one of the heading, but in this example instead of passing a string we pass a block: everything done within that block will be put inside the paragraph tags. This mechanism assures that all tags are closed properly and that always valid XHTML is generated. The message `#text:` escapes the given string to make sure the code can be displayed correctly within the web-browser.

The message `#anchor:to:callback:` is used to generate an anchor with an assigned callback block. The `anchor:` argument obviously renders the things that should be rendered as the content of the link. The `to:` argument specifies the place where the callback should be handled: usually this is within the same action, but occasionally someone might need to specify something else. The `callback:` argument is evaluated when clicking the link. As an argument the block receives the action that is executing the callbacks, note that this is not necessarily the same action that rendered the link and that is referenced using the keyword `self`.

`HtmlWriteStream` does not emit any unnecessary spaces into the output stream, which makes investigation in the HTML code somehow difficult. For this purpose, there is the possibility to enable the included pretty-printer with the method

```
HtmlWriteStream prettyPrint:true
```

The pretty-printer slows down the rendering process and might have unwanted effects on the output in the web-browser.

More advanced examples about html-rendering and callbacks can be seen in the Action-class `CallbackDemo`, that is part of the examples-bundle (see Section B.1). The user interface of this class is accessible by the url

```
http://localhost:8080/?action=CallbackDemo.
```

List of Figures

List of Tables

Bibliography

[1] Lukas Renggli. SmallWiki Collaborative Content Management, 2003. University of Bern, SCG Group. http://www.iam.unibe.ch/ scg/smallwiki/smallwiki.pdf

[2] Cincom VisualWorks Smalltalk. http://www.cincom.com/scripts/smalltalk.dll/.

[3] The WikiWiki behind Wikipedia. A free encyclopedia, collaboratively constructed over the internet. Uses PHP and MySQL database. http://wikipedia.sourceforge.net.

[4] MoinMoin. A nice and easy WikiEngine with advanced features written in Python. http://moinmoin.wikiwikiweb.de/.

[5] USEnet MODeration project. Perl-based. http://www.usemod.com/cgi-bin/wiki.pl.

[6] A Web Based Collaboration Platform written in PERL. Mature and full featured system, including revision control via CVS; with plugins. http://twiki.org.

[7] BoLeuf, WardCunningham, The Wiki Way: Collaboration and Sharing on the Internet. Addison-Wesley. (2001).

[8] Zope. An open source application server for building content managements, intranets, portals, and custom applications. http://www.zope.org.

[9] The Mozilla project. Mozilla web and email suite and related products and technology. http://www.mozilla.org.

[10] Wiki Markup Standard. The 'basic set' of text formatting rules. http://www.usemod.com/cgi-bin/mb.pl?WikiMarkupStandard.

[11] A free collaborative hypertext authoring program, written in CommonLISP. http://www.cliki.net.

[12] WikkiTikkiTavi. A wikki engine implemented with PHP and using MySQL for the wiki page database. By Scott Moonen. http://tavi.sourceforge.net/WikkiTikkiTavi.

[13] Wiki-Wiki implementation in Smalltalk. http://c2.com/cgi/wiki?SmallWiki.

[14] Unified Modeling Language (UML) Resource Page. http://www.uml.org.

[15] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. The Design Patterns Smalltalk Companion. Addison Wesley, 1998.

[16] John Brant and Don Roberts. Smalltalk Compiler-Compiler (SmaCC). http://www.refactory.com/Software/SmaCC.

[17] A Wiki implementation based on Squeak. http://minnow.cc.gatech.edu/squeak.

[18] Recreation of the Squeak implementation with variations using VisualWorks Smalltalk. http://wiki.cs.uiuc.edu/VisualWorks/WikiWorks.

[19] Avi Bryant and Julian Fitzell. Seaside. http://www.beta4.com/seaside2.

[20] Risk Analysis A Model. By Per Rhein Hansen. http://www.itu.dk/courses/DSK/E2003/DOCS/risk_analysis.pdf.

[21] The Number One HTTP Server On The Internet. http://httpd.apache.org.

[22] SSL 3.0 SPECIFICATION an Internet Draft dated November 1996. http://wp.netscape.com/eng/ssl3.

[23] Editor MACroS, an extensible, customizable, self-documenting real-time display editor. http://www.gnu.org/software/emacs/emacs.html.

[24] Steve Pepper, Ontopia AS. The TAO of Topic Maps. http://www.ontopia.net/topicmaps/materials/tao.html.

[25] Authentication, Authorization and Accounting in Ad Hoc networks. Sami Levijoki. 26th of May 2000. Department of Computer Science, Helsinki University of Technology. http://www.tml.hut.fi/Opinnot/Tik-110.551/2000/papers/authentication/aaa.htm.

[26] Camp Smalltalk Project. Smalltalk Web Application Zoo (Swazoo). http://swazoo.sourceforge.net.

[27] Standard Performance Evaluation Corporation. http://www.spec.org/osg/web99/.

[28] An explanation of the SPECweb96 benchmark, December 1996. http://www.specbench.org/osg/web96/webpaper.html.

[29] Apache JMeter. A 100% pure Java desktop application designed to load test functional behavior and measure performance. http://jakarta.apache.org/jmeter/.

[30] The Benchmark for Web Servers. Mindcraft, Inc. http://www.mindcraft.com/webstone/.

[31] http://httpd.apache.org/docs/misc/perf-tuning.html.