SmallWiki — A Meta-Described Collaborative Content Management System

Stéphane Ducasse
Software Composition Group
University of Bern – Switzerland
ducasse@iam.unibe.ch

Lukas Renggli
netstyle.ch GmbH
Bern – Switzerland
renggli@netstyle.ch

Roel Wuyts
DeComp
Université Libre de Bruxelles – Belgium
roel.wuyts@ulb.ac.be

ABSTRACT

Wikis are often implemented using string-based approaches to parse and generate their pages. While such approaches work well for simple wikis, they hamper the customization and adaptability of wikis to the variety of end-users when more sophisticated needs are required (*i.e.*, different output formats, user-interfaces, wiki management, security policies, ...). In this paper we present SmallWiki, the second version of a fully object-oriented implementation of a wiki. SmallWiki is implemented with objects from the top to the bottom and it can be customized easily to accommodate new needs. In addition, SmallWiki is based on a powerful meta-description called Magritte that allows one to create user-interface elements declaratively.

Categories and Subject Descriptors

D.1.5 [**Programming Languages**]: Object-oriented Programming; D.2.10 [**Software Engineering**]: Design; D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Languages, Design

Keywords

Object-Oriented Programming, Meta-modeling, Design and Implementation, Seaside, Smalltalk

1. INTRODUCTION

A Wiki or wiki (pronounced [wiki:], [wi:ki:] or [vi:ki:]) is a website that allows users to add content, as on an Internet forum, but also allows anyone to edit the content. "Wiki" also refers to the collaborative software used to create such a website. *[http://en.wikipedia.org/wiki/Wiki]*

While wikis offer a significant degree of freedom to their users to edit and share contents fast [7], the underlying wiki implementations are often less flexible and powerful than the model they promote. Wikis are mostly implemented using string-based approaches (*i.e.*, regular expressions) to parse and generate their pages. While such approaches work well for straightforward wikis, they hamper the customization and adaptability of wikis to the variety of end-users that require more sophisticated needs (*i.e.*, different output formats, user-interfaces, security policies, ...).

One might think that advanced wikis are not really necessary, and hence that simple wiki implementations that only allow users to change the contents of pages suffice. Experience shows that this is not the case:

Input and Output. Most wikis provide users with a simple wiki syntax to create rich XHTML pages, however they hamper the possibility to use other input and output formats. This is the reason why SmallWiki is storing the contents of a page within an abstract document tree that can be traversed to emit different output formats such as XHTML or LaTeX. Wikis based on strings require to duplicate the parsing functionality for every new output format. For complex wikis, such as WikiPedia [11], there are so many slow regular expressions applied to the input, that they are forced to implement sophisticated caching algorithms for different output and search formats.

User Interface. An experiment using wikis in classrooms showed that children and teachers require different user interfaces and functionalities [2]. Students should have a simpler user-interface compared to the teachers, that should be able to lock all the pages of her students at once.

Management. Another example is the maintenance of wikis that typically requires sophisticated functionalities such as: searching for all the pages containing more than 10 external links or finding all pages that were edited on a certain date and that have more than twenty uploaded pictures. Since such activities are typically done by end-users themselves, they should be supported by the wiki itself (such as not to break the metaphor of the wiki medium).

Customizability. The metaphor of a wiki should not stop at the level of editing pages. Therefore we need *customizable* wikis with an underlying implementation that supports the definition of new wiki components, and not only of changing contents of pages. SmallWiki allows one to customize its look using meta-pages that can be edited such as any other page and to include active components within the wiki to enhance the user experience.

WikiSym '05, October 16-18, San Diego, CA, U.S.A.

This paper presents SmallWiki 2, a second version of a fully object-oriented implementation of a wiki. SmallWiki is written with objects from the top to the bottom and it can be customized easily to accommodate new needs. In addition, SmallWiki is based on a powerful meta-description, named Magritte, that allows one to create web user-interface elements declaratively. It is worth to mention that the design decisions taken during the first implementation of SmallWiki have been totally revisited during the second implementation.

The contributions of the paper are:

- the description of the architecture and the design decisions of the implementation of SmallWiki and the lessons we learned in the process,
- the description of Magritte the meta-model and framework we developed and on which SmallWiki is based.

Section 2 presents some SmallWiki instances. Section 3 presents the architecture and design of SmallWiki. Section 4 demonstrates how the current model can be extended. Section 5 presents Magritte, a meta-model to describe all the wiki elements that is the key to much of the advanced functionality offered by SmallWiki. Section 6 lists the lessons we learned by implementing the second version of SmallWiki, as presented in this paper.

2. SMALLWIKI IN ACTION

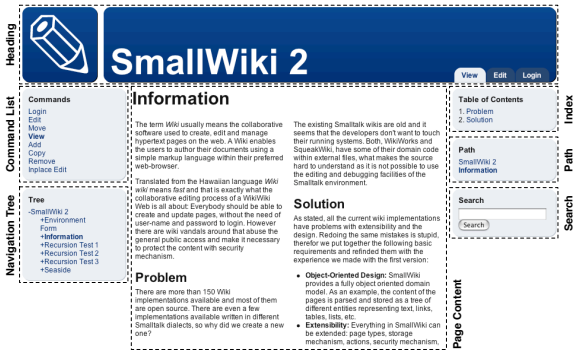

Figure 1: A page composed out of several components.

SmallWiki structures its layout out of different components: heading, command list, navigation tree, table of contents, search interface, page content, etc. For example in Figure 1 the page contains the header on top and the document in the center; on the left there is a list of possible commands and a tree view of the wiki; on the right a table of contents widget, the current navigation path and a small search box. Figure 2 shows another instance of SmallWiki where different components are used: a header, a list of commands, a login box and the document. Note that SmallWiki's look is based on Cascading Style Sheets (CSS), allowing each component to be skinned differently via its CSS specification.

Figure 3 shows a calendar: a more elaborated component that was developed by students following a software engineering course. Such a component shows that SmallWiki is at the border between a wiki with all the collaborative aspects of co-editing documents and content management systems. We see SmallWiki as a platform for building collaborative web-based applications.

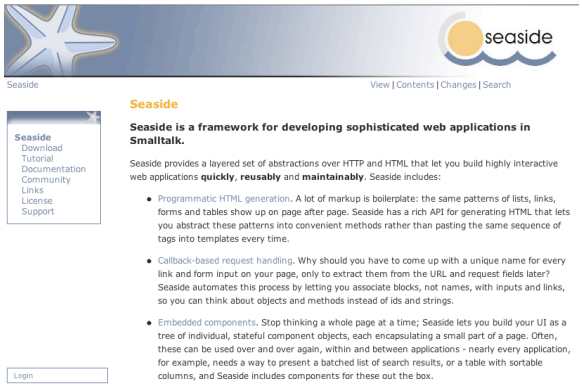

Figure 2: www.seaside.st shows another web site using SmallWiki.

Figure 3: A SmallWiki advanced component: a calendar.

3. ARCHITECTURE

SmallWiki's design has matured over the years [9]. During this process we tried to simplify it while making it more flexible. SmallWiki has been implemented and re-implemented from scratch by the second author in Squeak, an open-source Smalltalk [5]. We present here the key aspects of the implementation and architecture of SmallWiki 2.

3.1 Separation of Concerns

Web application development is difficult when having to deal with the shortcomings of the HTTP protocol as the right abstractions are missing [3]. Therefore we decided to use Seaside [10] as a framework of choice for the default view in SmallWiki. This approach greatly enhances the development of complex widgets. Since the user interface is built from Seaside components that automatically keep their state along a user session, it is now simple to implement, for example, a tree-widget that is displayed on every page and remembers its expanded nodes.

Since Seaside offers a proper separation between the model and the view, SmallWiki takes full advantage of it. As an example, it

is possible to use different non web-based view using the OmniBrowser [8] framework as shown in Figure 4; the same wiki pages can be browsed and altered via the OmniBrowser interface or via a web-browser.

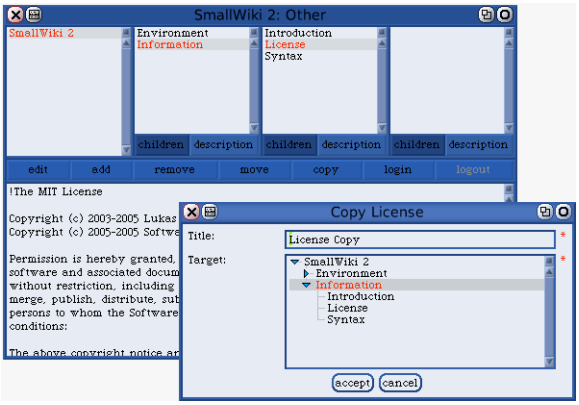

Figure 4: OmniBrowser view on a SmallWiki model.

3.2 Pages and Files

Unlike most other wikis, in SmallWiki structures can be nested arbitrarily into each other. The SmallWiki core implementation provides two basic structure types: pages and files, that build the main entities for any wiki. A page is a structure containing a document (see Section 3.3) that can be edited using the wiki syntax. A file is a resource, such as an Image-, Video-, Sound, or PDF-file, that has been uploaded. Other page types might be available if extensions to the base package have been loaded.

Links to other structures can be written either as absolute (*e.g.*, */Information/Copyright*) or relative (*e.g.*, *../Copyright*) paths. People being unfamiliar with this concept will create a link without any path elements and this will reference a child of the current page, what in most cases is desired anyway.

In addition the user can embed any linked object into the containing page, where the target can be another page or file that can be embedded into HTML. Note that embedding or nesting elements inside each other can lead to recursion which when not treated correctly would lead to infinite XHTML streams. SmallWiki detects possible recursion problems and in case of recursion just uses link-anchors instead of embedding. Element embedding is *transparent* to the user in the sense that it is expressed using the familiar wiki syntax *e.g.*, a page with two columns is achieved by creating a table embedding two different pages each into one column of the table, as we see in Figure 1. This greatly enhances the possibility to build complex layouts without bloating the wiki syntax with new features or using XHTML tags.

3.3 Document Objects

Every page consists of a title and a document representing its contents. The document is a Composite and includes all the basic elements to represent a text such as paragraph, ordered and unordered list, table, pre-formatted text, links, etc as shown in Figure 5.

When the user saves a text using the wiki syntax, it is then parsed using a parser built by the compiler-compiler SmaCC [1], and only the document tree is stored within the page. A visitor walking over

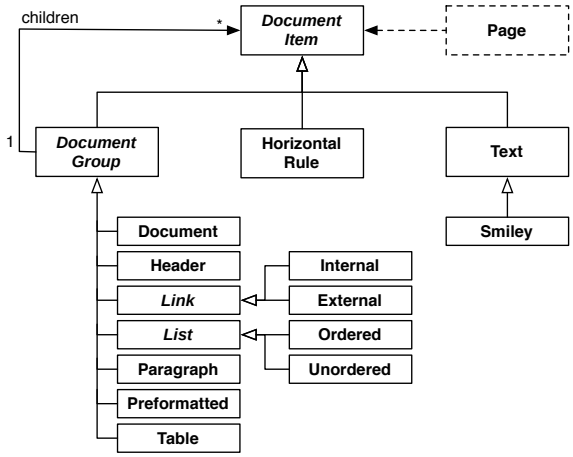

Figure 5: The document hierarchy: a simple composite.

this tree is able to transform this composite document back into an identical string that the user can modify again. Some nice features, such as the possibility to detect smilies, align table cells and add links everywhere, even within headings, greatly enhances the uniformity of the wiki input. Moreover, SmallWiki provides sophisticated in-place page editing facilities: unlike other wikis, where the user is forced to edit a page as a whole entity in one big text-area, in SmallWiki one can choose just to edit a specific paragraph that is then replaced within its context of the document with a smaller edit box; saving that paragraph parses the text and merges it back into the current document tree.

3.4 Visitors

The wiki structure *i.e.*, the page itself or an uploaded file is refined from WikiObject the root of the wiki elements that can be walked through using a Visitor. Having a full object-oriented representation of a document enables a lot of powerful features in SmallWiki *e.g.*, implementing a different output format is just a matter of writing a new visitor class and searching for broken links means to traverse and ask all the external link objects if they point to a valid internet resource, as we will demonstrate in Section 4.

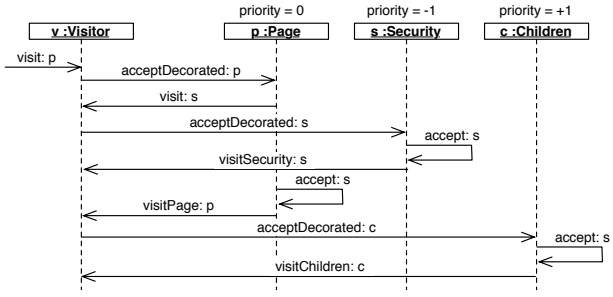

Figure 6: Visitor interacting with decorated objects.

In fact, a Page or a File is a DecoratedObject, an object using a Chain of Responsibility for certain aspects of its behavior, such as security or children. The decorations are tightly integrated into the visitors so that they can easily interact with the underlying model. Decorations contain a priority that is used by the visitor to deter-

mine the order in which the decorations are processed. The decorated object has a priority of 0. To ensure that it is processed first, the security decoration is assigned a negative priority. The children decoration has a positive priority, since it should be visited after the object containing the children, as seen in Figure 6.

3.5 Context and Commands

Most wikis keep all their application state as strings in the URL, its query parameters, in HTTP header fields and in associated session-cookies, exactly the way like most of todays web applications do. Using Seaside allows us to propose a much cleaner solution. Seaside provides a nice abstraction over this low-level protocol and we are now able to keep all our state within the application components themselves as proper objects. It is therefore not necessary to manually serialize and de-serialize our objects as strings. However, we still need a central place to remember and to change the context in which the user is browsing the wiki, such as the currently browsed structure, the running command and the user logged in.

SmallWiki introduces an immutable Context object, containing all the necessary information about the user browsing the site, all its associated security policies, the currently browsed page and the command being used with. Since every part of SmallWiki can modify the current context, say to navigate to a different structure, we must make sure that we do not loose the old context, since we might still need the original one for logging the changes with the persistency framework. Therefore sending the message goto:command: to a context returns a modified copy of the receiver, it is then the responsibility of the developer to make this context the current one. In Seaside the context is held in the top-level component of the wiki and can be requested or changed by raising a notification. We didn't want to use a global session object to remember the context, since we would like to keep the possibility to embed SmallWiki into an existing Seaside application that presumably already has its own session class. For example, in the OmniBrowser view (see Figure 4), the current context is kept within the browser model.

Calling the method execute on a command instance executes this command within a critical section, so that concurrent modifications of the domain model don't interfere with each other. In addition this ensures the modifications to be valid before processing and that they are logged in the persistency layer after execution. As an example let's have a look at the implementation of the command to add a new page to the wiki: doExecute is a hook method that is called from within the critical section in execute. The first line actually adds the newly created child to the current page and remembers the child within a temporary variable structure. It then tells the current context by sending goto:command: to go to in edit mode on the newly created child. However this new context won't be activated right away and it is remembered as the answer of the add command. In the meantime the SmallWiki persistency framework is able to log the executed command together with the old context, so that it can be undone or replayed if necessary.

```
AddCommand ≫ doExecute
  | structure |
  self structure children add: (structure := self newChild).
  self answer: (self context
                goto: structure
                command: structure editCommand).
```

The command hierarchy gives a clean interface to modify the domain model of SmallWiki. Actually every modification (or write access) to the model goes through a command, so that it can be logged and eventually undone at a later point in time. As we will see in Section 5.2.2, having an initial state of the model and a list of logged commands with their associated contexts allows the implementation of a prevalence or changeset-like persistency mechanism, in which each change is stored with a time-stamp. Hence, it is not even necessary to keep the old versions of a page explicitly in the domain model, because they can be easily obtained by going back through the history of commands selecting all edit-commands on a particular page.

3.6 Environment

SmallWiki unifies the look of the wiki-site with the wiki metaphor and allows one to define the look of the page using the wiki syntax itself, thus people only have to learn one concept that can be used seamlessly in different areas. Anywhere within the wiki one is able to define a special page called *environment* that is invisible to the casual user and that defines the look of a portion of the wiki. An environment is shared between all the children of the same page, unless a new environment is defined that replaces the previous one. Since the environment is a wiki-page, it can be edited and modified like any other page.

The default environment page creating the standard look of SmallWiki only consists of the following small piece of wiki text.

```
+Header+
| +Commands+ <br /> +Tree+ | +Contents+
SmallWiki 2 &mdash; Empowered by Seaside
```

The first line with +Header+ embeds the header widget that allows one to create an internal link that should be embedded into the target page. Even though the *Header* structure could be yet another wiki page, in this particular case we are using a Seaside component to draw and provide the necessary functionality. The next line creates a table containing the command and tree widget in a column on the left and the actual contents on the right. At the bottom we include some static text that will be displayed on every page.

Figure 7: Seaside SushiNet application in SmallWiki.

Furthermore in SmallWiki any Seaside component can be added exactly the same way as one would add a page into the wiki tree. In the above example we were using Seaside components that were particularly designed to be used within the wiki and provide its core functionality, however any other Seaside application can be added exactly the same way. In Figure 7 one can see a sushi web shop

that is included with the Seaside framework and is often used to demo the power of Seaside [3]. Without changing a line of code in the application itself and in SmallWiki it was added and is running within the wiki.

4. EXTENDING SMALLWIKI

As SmallWiki has been designed to be extensible and customizable, in this section we want to give some examples of small extensions.

4.1 Fixing broken links

Since URLs and associated resources are changing from day to day it is a common issue that web-pages contain invalid links. There are plenty of tools available that address this issue by going trough a web-site, parsing the HTML and checking the validity of the links. In SmallWiki we are able to address this issue simply by creating a subclass of Visitor and overriding the message visitExternalLink: to ask the link wether it is pointing to a valid resource and collect the broken ones within a collection. A user-interface might then start this visitor, display the broken links within a report and allow the responsible user to edit the links from one central place without being forced to go into every page and fix them manually. The only method to be implemented looks like the following one. As we will see later on in Section 5.2.1 we might also use the query engine and specify a query like *kind = 'ExternalLink' AND isBroken = true* to achieve the same result.

```
BrokenLinkCollector ≫ visitExternalLink: anExternalLink
  anExternalLink isBroken
    ifTrue: [ collection add: anExternalLink ].
```

4.2 Converting documents

It can be very convenient to convert a particular page or even a whole wiki tree in a different format than HTML, for example for exporting or printing. Since all the pages and documents are kept in one tree of objects it is trivial to write a visitor that walks this tree of entities and exports the contents in a format like LaTeX, OASIS (Open Document Format for Office Application, OpenOffice) or RTF (Rich Text Format, Microsoft Word). In fact this is exactly the same way as for creating the wiki syntax or the HTML view for the web browser. The following code extract shows the part of the rendering visitor emitting lists within a document as LaTeX:

```
LatexRenderer ≫ visitOrderedList: anOrderedList
    stream nextPutAll: '\begin{enumerate}'; cr.
    self visitAll: anOrderedList children.
    stream nextPutAll: '\end{enumerate}'; cr.

LatexRenderer ≫ visitUnorderedList: aUnorderedList
    stream nextPutAll: '\begin{itemize}'; cr.
    self visitAll: aUnorderedList children.
    stream nextPutAll: '\end{itemize}'; cr.

LatexRenderer ≫ visitListItem: aListItem
    stream nextPutAll: '\item '.
    self visitAll: aListItem children.
    stream cr.
```

4.3 Photo Gallery

There are a lot of photo gallery implementations around that allow one to upload pictures and automate the repetitive task of creating thumbnail pictures and linking them to original pictures. However usually this approach fall short of supporting text annotations, addition of extra links to other pictures or external sites, or inclusion into an existing web-site. A web gallery is currently being developed on top of SmallWiki. By subclassing File, we get a Picture class that can be added anywhere in the wiki and provides additional methods such as to scale and rotate or to query the width and height of the picture. Creating a subclass of Page called Album allows us to display a collection of pictures as thumbnails and navigate them easily. Of course these two new structure types can be added and transparently linked from anywhere within the wiki.

5. MAGRITTE

Lot of applications consist of a big number of input forms and dialogs, that need to be built and validated manually. Developers need a way to specify how objects are structured and how they can be modified so that views and editors can be created almost automatically. In SmallWiki, each domain element is described by a meta-description. The meta-description framework is called Magritte. Having such a description not only allows us to automatically create Seaside components as views for the web, but also to build other user interfaces without having to write a single line of code. It automates searches on our domain model, implements persistency, etc. Moreover, when changing the structure of a class one has to change the description at one single place and all the parts of the model and the user-interface that rely on the provided descriptions immediately adapt to the new requirements, avoiding to re-factor different parts of the code.

5.1 Magritte in a Nutshell

Magritte is a meta-description framework, describing domain instances and their respective fields [6]. It contains not just type information, but also more semantic information: it specifies how the field is accessed, an optional label and comment; furthermore it defines boolean properties like if it the field is required, read-only, visible, persistent, etc.

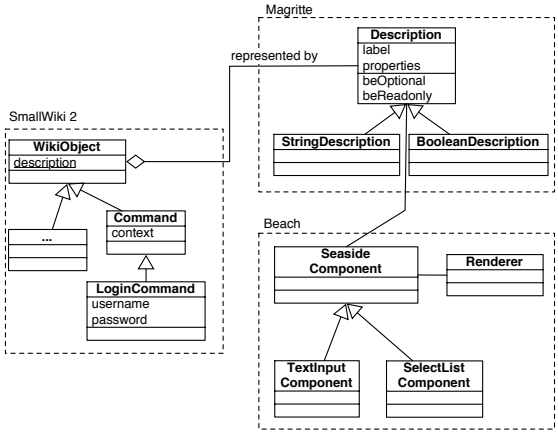

Figure 8: Magritte Description of the Command Hierarchy.

Meta-descriptions in SmallWiki are not only used for describing the domain model itself but also for the SmallWiki back-end object representation objects, such as the command objects mentioned in Section 3.5. As an example let's have a look at the copy command in SmallWiki. On the class side there are two methods each returning a description. Both methods are initialized with a selector to access the value of the model; *i.e.*, for the title description only the read-accessor title is specified, but Magritte will automatically define the write-accessor title:. The descriptions of the title and target

are tagged to be required, which means that both fields cannot be left empty.

```
CopyCommand class ≫ descriptionTitle
    ^(MAStringDescription selector: #title label: 'Title' priority: 100)
        beRequired;
        yourself.

CopyCommand class ≫ descriptionTarget
    ^(SW2StructureDescription selector: #target label: 'Target' priority: 200)
        beRequired;
        yourself.
```

When asking an instance of such a copy command for its description, Magritte collects all the methods on the class side starting with the name description and returns a composed description consisting of the two elements as seen above. The value of the priority is used to sort the descriptions as preferred to give a consistent look in the user-interface.

Figure 9: Automatically created web-interface to copy a page.

There are multiple uses of such meta-descriptions. The most immediate one is that a description is used to create a visual Seaside component. Getting a Seaside component allowing the user to edit the command instance is as simple as sending asComponent. Usually the returned component is then decorated with a form, displaying a save and a cancel button, and a validator, catching and displaying any validation errors if necessary. Figure 9 shows how this component looks like with the default style-sheets used. We have already seen a very similar picture in Figure 4, where we sent asMorphic instead of asComponent to the same command to get a dialog for the OmniBrowser user-interface.

The following code presents how the copy page command shown in Figure 4 and Figure 9 were created:

```
" Morphic View "
result := aCopyCommand asMorph
                    addButtons;
                    addWindow;
                    callInWorld.

" Seaside View "
result := self call: (aCopyCommand asComponent
                            addValidatedForm;
                            yourself).
```

Descriptions are not just static elements that are solely defined on the class side and that never change. They can also be composed, modified and created at runtime. Composed descriptions understand the Smalltalk collection protocol, hence different views can be easily built by selecting only a certain sub-set of elements of a domain object. In addition descriptions of different domain-objects can be combined to a bigger one.

Another feature of Magritte is to give the power to create, add and modify certain descriptions to the user: since descriptions are described too with meta-description, a simple interface enables users to build forms from within their web browser, without writing a single line of code. Figure 10 shows an example of such an interface, in which the user defined a description for an address database. Scripting environment like HyperCard were very popular around 1990, Magritte and SmallWiki are a first step to enable such an approach in the context of web-applications.

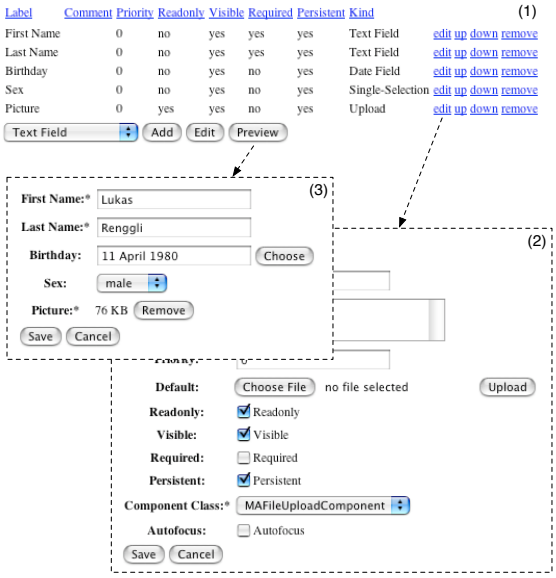

Figure 10: Building forms from the web browser: (1) the form definition with all its descriptions, (2) editing a description, (3) a preview form build from the currently defined descriptions.

5.2 Power of Magritte

In the following sections we discuss the implementation of the search engine and the persistency framework with the help of Magritte.

5.2.1 Searching

Wikis tend to grow over time, hence it becomes very important to have sophisticated ways to locate the desired information. In most cases we want to search for a page containing a particular sub-string, however sometimes it would be more precise to only look for pages that satisfy a certain condition. SmallWiki with the help of the meta-descriptions of Magritte allows one to write such conditions in the search field and display the matching pages. The query *kind = 'Table' AND rowCount > 3* returns all the pages with tables that have more than 3 rows and *url matches: '*.ch*'* returns pages with external links having a swiss domain.

To implement this functionality we have written a parser that reads the search expression and builds a tree of relations. When we send

isSatisfiedBy: to the root node of this tree, we either get true if the argument matches the criteria or false if it doesn't match as a return value. The relation tree is evaluated recursively using an escaper to abort the evaluation immediately if nothing can change the result of the expression anymore, what largely enhances the speed of the query processing. To determine if a certain basic condition such as title beginsWith: 'SmallWiki' is met the meta-descriptions come into play again: the model object to be checked is asked for its descriptions and it is checked if it has got an attributes called title and if this attribute is useable with the relation beginsWith:. If those two preconditions are fulfilled the value is read from the domain model, the comparison is done and the result is returned. Again we have a visitor that walks over the wiki tree and collects all the possible matches. A simple widget is used to display the result of the query.

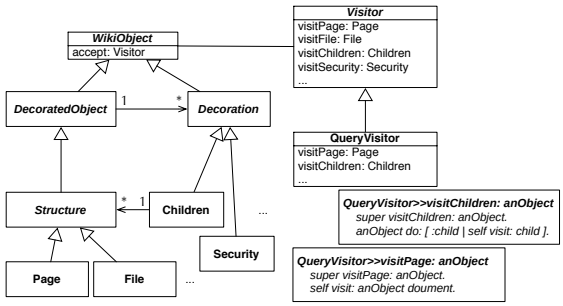

Figure 11: Walking trough the SmallWiki model using a visitor.

The following example shows how the query string is parsed and passed into the visitor, that descends into each wiki structure, as seen in Figure 11, and then collects and returns all the matching structures.

```
SearchWidget ≫ search
  searchResults := QueryVisitor
                start: self context kernel root
                query: (MARelationParser parse: self queryString).

DescribedRelation ≫ isSatisfiedBy: anObject
  | description |
  description := anObject descripition
                    at: self selector
                    ifAbsent: [ ^ false ].
  ^ super isSatisfiedBy: (description read: anObject).
```

5.2.2 Persistency and Versioning

SmallWiki takes a prevalence [12] or change-set-based approach to version its information. The idea is to keep the whole data in RAM – if there isn't enough RAM on the server it will be transparently swapped out by the operation system – so the system runs very fast as no objects have to be de-serialized. To avoid loosing data, every night, or in any reasonable period, a snapshot of the whole wiki tree can be saved. In addition all commands that are executed on the model are serialized immediately after being processed. The meta-descriptions of the command tells the persistency layer how the object has to be serialized and eventually restored later on. During crash recovery, SmallWiki retrieves its last saved state from the snapshot and then reads in the commands and applies them to the model exactly as if it had just come from the user interactions.

With this approach we get versioning- and undo-facilities for free. Suppose we want to see all the changes that have been made to a specified page, we just have to go trough the command log and select all the edit-commands of this particular page. Loading them allows us to see the changes of that page, and restore any old version by re-applying the command.

6. LESSONS LEARNED

During the implementation of the two versions of SmallWiki, we learned that having objects objects down to the roots is the key aspect of efficiently implementing an extensible wiki that adapts to a wide variety of needs. In the following paragraphs we compare some implementation details from SmallWiki 1 to SmallWiki 2 and to other wiki implementations:

Testing. Compared to the first version, SmallWiki 2 increased the number of unit tests from 200 to more than 1200, covering the whole model of SmallWiki, including Magritte. This makes it possible to change and verify the code and comes in extremely useful when porting SmallWiki to other Smalltalk dialects or when writing extensions that could break existing code.

Parser. Using a parser to read the wiki input, to build a proper object model and to walk trough it using visitors saves a lot of code: the current implementation of SmallWiki featuring scanner, parser and document hierarchy only consists of 550 lines of Smalltalk code, whereas the same functionality implemented for WikiPedia [11] using regular-expressions counts more than $3'000$ lines of code (excluding comments). Moreover these regular-expressions are duplicated trough-out the code-base of WikiPedia, *e.g.*, to implement the query engine, what makes it extremely difficult to change and enhance the syntax.

Structures. In the first version of SmallWiki, we distinguished between a folder (*i.e.*, a page with children) and a page. This led to problems because it was difficult to change the structure of a wiki after the fact. In the new version, we only have pages and no folders but any page can be decorated to get children. Hence any page can play to role of a folder and vice versa dynamically a page can loose its children. This means the user is not enforced to decide upfront how his wiki will be structured, but is able to add and remove children later on as wished. SmallWiki also provides an interface to move and copy whole subtrees to different locations easily.

Separation. SmallWiki 1 was designed to be used within a web context [9]. It was built on top of its own web framework. However, web application development is difficult when having to deal with the shortcomings of the HTTP protocol as the right abstractions are missing [3]. In SmallWiki 1 the model and the view were strongly coupled. For example an action to be performed on a page was a mixture between a Command design pattern [4] and the associated web view. It was then nearly impossible to use a command in a different view. Now SmallWiki 2 cleanly separates the model and the view in different packages that can be loaded and used independently.

View. In SmallWiki 1 all the application state was kept as strings in the URL, its query parameters, in HTTP header fields and in associated session-cookies, exactly the way like most of todays web applications do. Using Seaside as a default view

allows us to propose a much cleaner solution. Seaside provides a nice abstraction over this low-level protocol and we are now able to keep all our state within the application components themselves as proper objects. It is therefore not necessary to manually serialize and de-serialize our objects as strings.

Embedding. Structures can be embedded into each other by creating a special kind of link, this greatly enhances the possibilities to layout and structure the wiki. SmallWiki supports absolute and relative links, so that editors can easily create navigation facilities between the nested structure.

Commands. Modifying the model trough the use of a clean implementation of the command pattern allows the implementation of a prevalence like framework. Furthermore having the whole command history available gives us the possibility to undo modifications and restore the state of the wiki at any point in the past.

Meta-Description. We learned that having a powerful meta-model brings a lot of flexibility in different areas into the framework. Without writing additional code we are able to alter different parts of SmallWiki, such as the views, the search and the persistency solely by changing or adding a few lines of meta-descriptions.

Persistency Persistency and versioning is a crucial part of any wiki. In SmallWiki 1 we were using a simple snapshot mechanism, dumping out all the structures in user defined intervals. The obvious problem here is that if the computer crashes just before doing a snapshot all the changes since the last snapshot are lost. The versioning of the pages was achieved by keeping a collection of all the old pages within the model, that has disadvantages as well: since old versions are only accessed rarely and therefore it is not efficient to keep them in memory all the time. In addition, the memory footprint of SmallWiki 1 never shrunk, since all the changes have to be versioned, therefore even deleting parts of the wiki didn't reduce its actual size. This could lead to performance problems while saving and loading a snapshot of a huge wiki with lots of mutations over the time. SmallWiki 2 provides prevalence based approach, so that every change is stored to the filesystem so that any point after a snapshot can be easily restored by replaying the applied commands.

7. CONCLUSION

Wikis are a quick and efficient way to collaborate via simple web-browser interfaces. However, as wikis grow up, more advanced functionalities need to be incorporated (such as advanced management, maintenance and search operations). Current implementations of wikis that are based on string manipulation are badly suited to support this new generation of wikis.

This paper presents SmallWiki, a fully object-oriented wiki and content management system that is described using Magritte as a meta-model and that uses the Seaside framework to overcome traditional HTTP-limitations. Magritte forms the conceptual backbone of the implementation. Seaside lets SmallWiki cleanly divide domain model from UI, alleviating the need for object serialization. The resulting combination was shown to be very customizable.

Our long term goal of SmallWiki is to define an environment to enable user scriptable web applications, similar to HyperCard in its time.

Acknowledgment. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061-655.00/1).

8. REFERENCES

[1] J. Brant and D. Roberts. SmaCC, a Smalltalk Compiler-Compiler. http://www.refactory.com/Software/SmaCC/.

[2] S. Ducasse and F. Ducasse. De l'enseignement de concepts informatiques. *Journal de l'association EPI Enseignement Public et Informatiques*, 4(97), Sept. 2000.

[3] S. Ducasse, A. Lienhard, and L. Renggli. Seaside - a multiple control flow web application framework. In *Proceedings of ESUG Research Track 2004*, Sept. 2004. To appear.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[5] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326. ACM Press, Nov. 1997.

[6] R. E. Johnson and B. Woolf. Type object. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

[7] B. Leuf and W. Cunningham. *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley, 2001.

[8] C. Putney. OmniBrowser, an extensible browser framework for Smalltalk. http://www.wiresong.ca/OmniBrowser/.

[9] L. Renggli. Smallwiki: Collaborative content management. Informatikprojekt, University of Bern, 2003.

[10] Seaside, developing sophisticated web applications in Smalltalk. http://www.seaside.st.

[11] WikiPedia, a web-based, free-content encyclopedia. http://www.wikipedia.org.

[12] K. Wuestefeld. Prevayler, a prevalence layer for Java. http://www.prevayler.org.