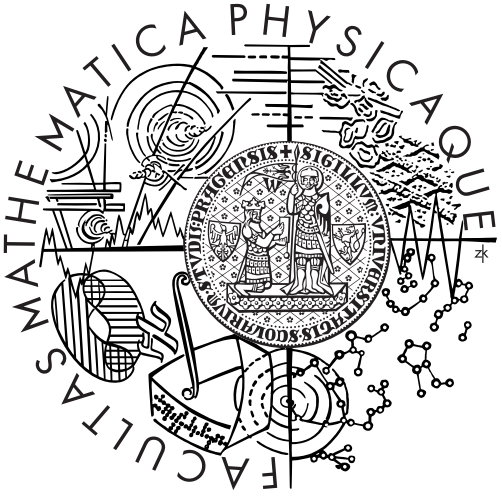
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Zdeněk Louženský

Wiki Synchronization Tool

Department of Software Engineering

Supervisor: RNDr. Michal Kopecký, Ph.D.,
Study Program: Computer Science, Software Systems

2008

I would like to thank my supervisor RNDr. Michal Kopecký, Ph.D. and my consultant Ing. Martin Matula for their valuable suggestions and observations, and for fruitful discussions and last but not least my family for their support in my life and studies.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on 11th December 2008 Zdeněk Louženský

Contents

List of Figures

List of Tables

Název práce: Wiki Synchronization Tool
Autor: Zdeněk Louženský
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.
e-mail vedoucího: Michal.Kopecky@mff.cuni.cz

Abstrakt: Ucelený přehled existujících wiki engines, jejich architektur, markup jazyků které používají a jejich analýza. Přehled a analýza API poskytovaných wiki engines pro integraci s ostatními aplikacemi. Popis existujících řešení pro migraci wiki obsahu. Návrh možných řešení konektorů pro wiki engines. Popis architektury použité při implementaci tohoto řešení a jejích vlasností. Popis jednotlivých modulů, algoritmů a jejich interfaců. Ukázkové implementace modulu a algoritmu. Diskuze a popis unifikovaného objektového modelu včetně jeho vlastností. Ukázková reprezentace wiki stránky v modelu. Budoucnost tohoto open-source projektu. Uživatelská dokumentace projektu.
Klíčová slova: wiki, markup language, synchronizace

Title: Wiki Synchronization Tool
Author: Zdeněk Louženský
Department: Department of Software Engineering
Supervisor: RNDr. Michal Kopecký, Ph.D.
Supervisor's e-mail address: Michal.Kopecky@mff.cuni.cz

Abstract: Compact overview of existing wiki engines, their architectures, markup languages that they use and their analysis. Overview and analysis of provisioning APIs provided by wiki engines for integration with other applications. Description of existing solutions for migration of wiki content. Suggestions of possible connectors for wiki engines. Description of architecture which was used for implementation of this solution and description of its features. Description of particular modules and algorithms including description of their interfaces. Implementation of sample module and sample algorithm. Discussion and description of unified object model including its capabilities. Sample representation of wiki page in the model. Future of this open-source project. User guide.
Keywords: wiki, markup language, synchronization

Preface

Original assignment

The goal of this thesis is to design and develop an automated web based tool that can be used to synchronize and/or migrate content from one Wiki implementation to another (e.g. between MediaWiki and JSPWiki). The tool will among others need to understand the different wiki formats, represent these formats in an unified object model and perform necessary conversions. It should also take advantage of any programmatic APIs exposed by the wiki servers to perform read and write operations. All conversions and data transfers have to be optionally logged and reported.

Scope of the thesis

The first chapter of this thesis focuses on the introduction to the problem, defines basic terms regarding it, brings the review of the top popular engines and describes the differences among wiki engines.
The second chapter tries to analyze the problem much deeper. If offers the detailed analysis of existing conversion solutions, analysis of the architecture of wiki engines, review of solutions for exchanging content between application and wiki, and possible use cases of this project.
The architecture of this project is discussed in the third chapter. Firstly, the requirements which should be satisfied are discussed there. Possible solutions for architecture are discussed then followed by a detailed description of the chosen architecture including description of all important modules and interfaces. At the end of the chapter there is an example how to extend the project with new modules and algorithms.
Unified object model is presented in the fourth chapter. The chapter begins with description of the requirements for object model followed by discussion of possible solutions. Detailed description of model elements is provided then and finally a sample object representation of wiki page is shown.
Fifth chapter focuses on future of this project. There are discussed extensions and future improvements.
User and installation guide is provided in the sixth chapter and the last seventh chapter is the conclusion of this thesis. It evaluates benefits which brings this thesis.

Chapter 1

Introduction

1.1 Phenomenon Wiki

Wiki is a collection of web pages designed to enable anyone who accesses it to contribute or modify content, using a simplified markup language. Wikis are often used to create collaborative websites and to power community websites. The collaborative encyclopedia Wikipedia is one of the best-known wikis (it is based on MediaWiki engine). Wikis are used in business to provide intranets and Knowledge Management systems. Ward Cunningham, developer of the first wiki software, WikiWikiWeb, originally described it as "the simplest online database that could possibly work".

1.1.1 History

According to [1] Ward Cunningham started developing WikiWikiWeb in 1994. It was named by Cunningham, who remembered a Honolulu International Airport counter employee telling him to take the "Wiki Wiki" shuttle bus that runs between the airport's terminals. According to Cunningham, "I chose wiki-wiki as an alliterative substitute for 'quick' and thereby avoided naming this stuff quick-web." In the early 2000s, wikis were increasingly adopted in enterprise as collaborative software. Common uses included project communication, intranets, and documentation, initially for technical users. Today some companies use wikis as their only collaborative software and as a replacement for static intranets. There may be greater use of wikis behind firewalls than on the public Internet. On March 15, 2007, wiki entered the online Oxford English Dictionary.

1.2 Popularity

Wikis soon became very popular and are used almost in every company or even in every project like I did when worked on this thesis. It is worth to know which Wiki Engines are the most popular as quality is usually reflected by popularity. In [2] their authors used Google results to compose a list of most popular wiki engines. To every wiki engine name like "MediaWiki" they added "wiki" in an effort to exclude

Figure 1.1: Wiki Wiki Bus

ambiguity. Then searched the Google using the resulting query without quotes. Their results can be seen in Table 1.1.

1.3 Benefits & downsides

There are many reasons why wikis are so popular nowadays. These are the most important:

- Anyone can edit them

- Are easy to use and learn
  Even young children are able to use them. The National Research Council of Canada published research results [1] proving that 8 - 9 years old children were able to write a nontrivial adventure game despite the fact that they never used wiki before. Adventures included images and hyperlinks among wiki pages. The most issues came from link creation and management but eventually almost all groups of students created the adventure successfully.

- Wiki pages are published immediately
  There is no need to wait for publisher to create a new edition or update information.

- Remotely accessible
  People located in different parts of the world can work on the same document.

[1] http://www.wikisym.org/ws2005/proceedings/paper-01.pdf

#	Wiki Engine	Google results	License
1	MediaWiki	133000000	GPL
2	TiddlyWiki	1420000	BSD
3	PukiWiki	1220000	GPL2
4	MoinMoin	1180000	GPL
5	DokuWiki	1160000	GPL 2
6	JotSpot	1140000	
7	TWiki	1120000	GPL
8	PmWiki	1100000	GPL2
9	TikiWiki	1090000	LGPL
10	Socialtext	1090000	
11	PBwiki	1070000	
12	Wetpaint	1030000	
13	Wikka Wiki	919000	GPL
14	Confluence	878000	Commercial
15	JSPWiki	875000	LGPL
16	SnipSnap	702000	GPL
17	PhpWiki	615000	GPL
18	MoniWiki	603000	GPL
19	Instiki	530000	Ruby GPL
20	Wikispaces	480000	

Table 1.1: Most popular wiki engines [2]

- Support versioning
  Usually the wiki engines keep track of every change made and so it is very simple to revert back to a previous version of a page.

- Wiki has no predetermined structure
  That consequently mean that wikis can be used for a wide range of applications such as presentation of company (CMS for free), managing tasks, project documentation, software project pages, knowledge base etc.

- Free
  There is a wide range of free or even open source wiki engines to choose from so licensing costs shouldn't be a barrier to installing them.

Nothing is perfect. Let's focus now on the problems which come with usage of wikis. Here is a list of main drawbacks:

- Anyone can browse and edit them
  They are not suitable for confidential information. However, it is usually possible to setup some kind of authentication and authorization to avoid this problem.

- Open to SPAM and Vandalism if not managed properly
  This is probably the biggest problem. It can be partly avoided by authorization

and versioning mechanisms.

- Users must be connected
  Doesn't matter for what purpose is wiki used (private wikis in companies or public on the internet), users always must be somehow connected to it.
- The flexibility of wiki's structure is not always an advantage
  The information could easily become disorganised as content grows.
- Need for a computer or mobile device

1.4 Wiki Engines

Wiki engines typically take advantage of three tier architecture (Figure 1.2) consisting of data layer where the content of wiki is stored, application layer which controls wiki functionality and a thin client which is represented by a web browser. When a wiki reader opens a wiki page the browser asks application server to provide a HTML source of the page. Business logic of the application layer then fetches page content from the data layer, processes it and returns HTML source of the page back to the web browser. This is a typical concept how wiki engines are implemented. However, all of the parts of this concept can vary on different engines. First of all, the data

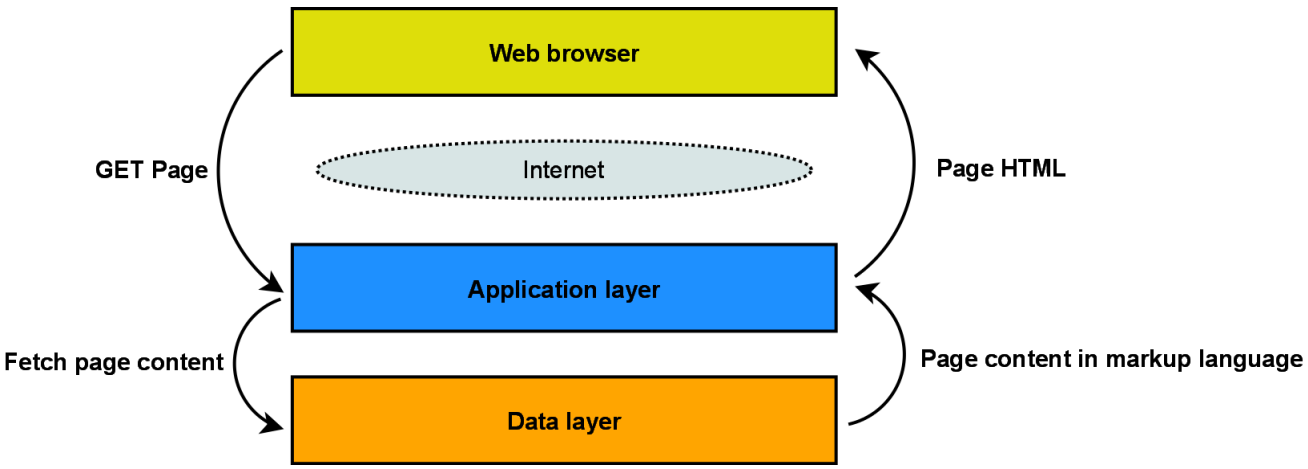


Figure 1.2: Typical three-tier architecture

layer is usually implemented using database, text files or some versioned files such as RCS. Most popular database engines are MySQL, PostgreSQL and Oracle; however it is possible to find wiki engines which use DB2, SQL Server and others. On top of that even if the database engine used by two different wiki engines is the same, database schema will certainly differ. Similar problem is with files because the format of the files is completely up to the wiki engine. Depending on what storage is used different access method to the data is required - SQL, file handlers etc.
Application layer can be even more diversified. Different application server can be

used - Apache, MS IIS, GlassFish etc. Application servers are suitable for different implementation languages. Mostly the wiki engines are implemented in PHP, Java, Perl, Python or ASP. Obviously distinct application servers require specific environment - typically Unix and MS Windows.
The web interface accessible through web browser also is not the only option. Some wiki engines provide other interfaces such as XML-RPC, REST which are supposed to be used for integration with other applications. However, even these interfaces bring new issues like encoding problems etc.
Wiki engines also differ in features which provide - some engines support page versioning some don't, version numbers are not standardized. In some wikis it is possible to create attachments and attach files to pages; In other wikis it is possible to attach them only to the whole wiki. Many wiki engines offer authorization mechanisms, some wikis support plugins, but the main distinction from the point of view of this project are different markup languages used by engines.

1.5 Wiki Markup Language

Wiki markup is a markup language that offers a simplified alternative to HTML and is used to write pages in wiki websites such as Wikipedia. There is no commonly accepted standard markup language although attempts were made. The grammar, structure, available features, used keywords and so on are dependent on the particular wiki engine.

1.5.1 Wiki Markup Mess

Typically every wiki engine uses own markup language which is somehow different from all the others. For example, all markup languages have a simple way of hyperlinking to other pages within the site, but there are several different syntax conventions for these links. Some wiki engines allow extensive optional use of selected HTML elements within content, others a smaller subset, and still others no HTML at all. There exist hundreds of different wiki engines and their markup lan-

```
==Section==
===Subsection===
====Sub-subsection====
```

Figure 1.3: MediaWiki Headings

```
! Level 1
!! Level 2
!!! Level 3
```

Figure 1.4: JSPWiki Headings

```
---+ Level 1
---++ Level 2
---+++ Level 3
---++++ Level 4
---+++++ Level 5
---++++++ Level 6
```

Figure 1.5: TWiki Headings

guages usually differ in at least a few elements. Heading can be taken as typical example (Figures 1.3, 1.4, 1.5).

Some wiki markups allow omitting the trailing equal signs some don't. Similarly count of equal signs can differ - the more equal signs the higher level of heading or vice versa. Wiki engines in case of headings don't differ in markup syntax but also in the number of allowed sub-headings. Usually are supported at least three levels of headings.

```
[[OtherInternalPage]]
[[OtherInternalPage | Title to display]]
[http://www.example.org]
[http://www.example.org Title to display]
```

Figure 1.6: MediaWiki Links

```
[OtherInternalPage]
[Title to display | OtherInternalPage]
[http://www.example.org]
[Title to display | http://www.example.org]
```

Figure 1.7: JSPWiki Links

Links are usually created using '[' and ']' symbols but count of these symbols may differ. Some wiki engines use one symbol '[', some two consecutive symbols '[['. Some wiki engines use one '[' for internal links and '[[' for external links. Let's take a look at a few examples on Figures 1.6, 1.7, 1.8. Text formatting is not an exception. Some wiki engines use HTML-like syntax for formatting elements, some mix of HTML elements and own elements and some use all non HTML elements (Figures 1.9, 1.10, 1.11). These few examples should be enough to outline the fact that there is a complete mess in markup languages and that a move from one wiki engine to different one can be very complicated not speaking about manual translation. Excellent tool for comparing attributes and markups of many wiki engines can be found at WikiMatrix[2].

[2]http://www.wikimatrix.org

```
[[OtherInternalPage]]
[[OtherInternalPage][Title to display]]
[[http://www.example.org/]]
[[http://www.example.org/][Title to display]]
```

Figure 1.8: TWiki Links

```
'''bold'''
''italic''
<u>underlined</u>
<tt>monospace</tt>
```

Figure 1.9: MediaWiki Text Formatting

1.5.2 Standardization Attempts

Now when we know about all the differences among different markup languages, it would be great to have one standardized markup used in all or at least most wiki engines. Contributing to several wikis, exchanging content among wikis, moving to different wiki engine etc. would then be much easier. First attempt to standardize wiki markup was done by TikiWiki in 2004. They tried to publish their markup as an IETF RFC [3]. Unfortunately their attempt wasn't accepted by others at all. The reason for that probably is that standardized markup should have tried to find some compromise among all the wiki markups and be more similar to MediaWiki's which is used by millions of people everyday on Wikipedia.

1.5.3 Wiki Creole

Much more successful attempt was made by Creole. During the WikiSym2006 the Wiki Markup Standard Workshop group came with a completely different proposal. Learned from TikiWiki unsuccess, Creole is not replacing an existing markup by its own like TikiWiki did but instead tries to base the markup on compromise among all main wiki engines and choose the best syntax for particular element. Moreover Creole is not intended to be the only markup used. It is expected that native engine markup remains and Creole will be supported as supplementary markup. Each Creole element was chosen after an intensive research and detailed discussion on the talk pages of Creole wiki. Discussed were goals and good practices formulated by a broad variety of wiki developers and users from different engines. Most important goals (based on [3]) were:

- Collision Free
  No collisions with content text because content text is interpreted as markup. No collisions with native markup because Wiki Creole is supplementary to

[3] http://tikiwiki.org/RFCWiki

```
__bold__
''italics''
%%(text-decoration: underline) text %%
{{monospace}}
```

Figure 1.10: JSPWiki Text Formatting

```
*bold*
_italic_
<u>underline</u>
=monospace=
```

Figure 1.11: TWiki Text Formatting

the native markup. No collisions with tradition and habits which would lead unintuitive construct.

- Cover the common things people need
  Headings, lists, tables, text formatting etc.
- Extensible by omission
  Keep something undefined like nesting of markup elements. Postpone the decision.
- Not new
  Should be based on existing markups as much as possible.

Good practices cover things like simplicity of learning and teaching of the markup, as well as ability to be typed fast and being readable and non-destructive. Basically markup should be easy, readable and strong enough to satisfy common expressive needs. Reasons for chosen syntax elements are fully documented. For instance for headings there were these options:

```
== header ==
== header
!! header
h2. header
```

Using equal signs '=' is the most popular wiki heading markup (Figure 1.12). Since there are more equal signs for smaller headers, sub-headers will become more indented making it easier to get a visual overview from the markup alone. Closing equal signs are optional, making Creole more flexible since many wiki engines do not use them.
There was a complaint for exclamation points '!!' that they do not indent further headers, making it difficult to keep an overview of the document content.
Syntax like 'h2.' looks too much like HTML.

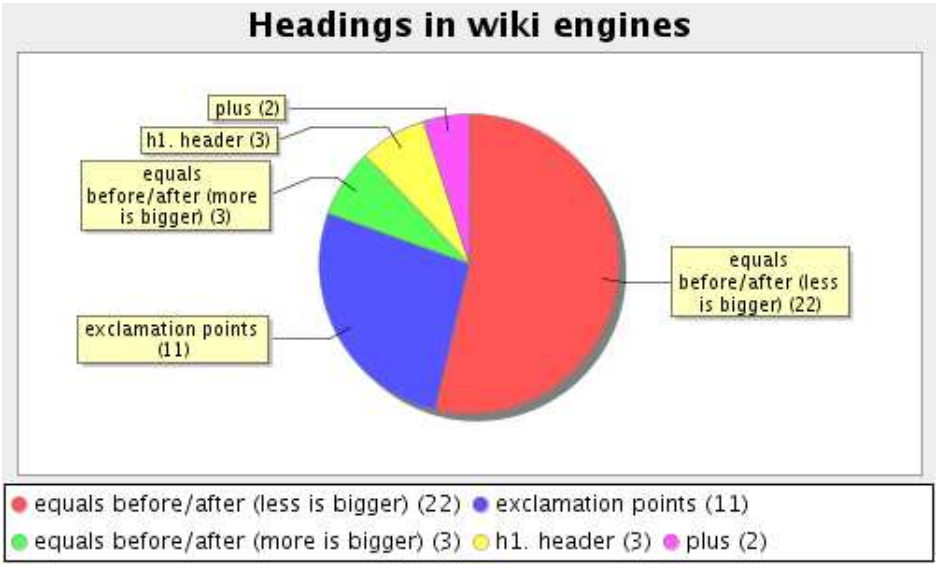


Figure 1.12: Popularity of headings' symbols

The demonstration of the chosen Wiki Creole Markup is placed in Appendix D.

Creole is being accepted by more and more engines recently. Here is a list of the top engines which already support Creole ([4]):

- MoinMoin - Creole 1.0
- DokuWiki - Creole 0.1 through plugin
- TiddlyWiki - Creole 0.1 through plugin
- JSPWiki - Creole 0.6
- PMWiki - Creole 0.4

1.6 Personal objectives

My personal objectives for this thesis is to hide all the differences among different wiki engines and provide an easy-to-use tool for migration and synchronization among different wiki engines with a minimal knowledge of those environments and markups, available as online web service. Moreover provide a flexible infrastructure for simple future algorithm and module extensions. Create an extensible object model for further manipulation and being an inspiration for new object model based wiki engines. The whole project is open sourced [4] waiting for new contributors to set up creative community.

[4] https://wikisync.dev.java.net

Chapter 2

Analysis of the problem

This chapter focuses on analysis of existing wiki converters and implementation solutions which can be used in this project.

2.1 Existing solutions

There existed only two maintained solutions for conversion at the time of writing this thesis - Pm2Media Converter [1] and Universal Wiki Converter [2] which appeared just recently.

2.1.1 Pm2Media Converter

Pm2Media is an open-source wiki converter reading contents of an existing PmWiki installation and converting it into MediaWiki syntax and then posting it into a MediaWiki installation. Pm2Media was never meant to be universal conversion tool, so its design is very simple. Pm2Media does not use any unified object model, but performs conversion through regular expressions which replace syntax elements of PmWiki with syntax elements of MediaWiki. This approach is very straightforward and thus quick to implement, on the other hand limits the possibility of future extensions or at least makes them much harder. Another feature of this tool is that for loading of PmWiki pages uses PmWiki's ability to display source code of wiki pages simply by adding parameter 'action=source' to the wiki page URL. List of all pages must be parsed from index page - the page which contains references to all the pages which are available on wiki. Similarly all the pages are stored to the MediaWiki using HTTP's POST method to specific URL (with adequate parameters). One big advantage of this attitude is that you do not need any specialized API for loading and storing pages, drawbacks are: it is necessary to parse index page, source wiki engine must provide a way to display page source which is not so common, there might be some authorization issues such as setting of HTTP cookies and definitely

[1] http://sourceforge.net/projects/pm2media/

[2] http://confluence.atlassian.com/display/CONFEXT/Universal+Wiki+Converter

there might appear evolution problems such as index page source, action parameter or POST URL might change from version to version.

2.1.2 Universal Wiki Converter

Universal Wiki Converter is an open-source solution from commercial Confluence wiki which appeared recently. Despite word 'Universal' in the tool name it is aimed to offer conversion from other wiki engines to Confluence wiki - vice versa is not possible. UWC is able to migrate content of almost all of the most popular wikis (JSPWiki, PmWiki, TWiki, DokuWiki, MoinMoin, MediaWiki etc) to the Confluence installation. Synchronization is not supported at all. Loading and storing of content is solved in different ways - some modules use database connections, some modules need to have access to wiki engine's local files which contain the content or in case of Confluence proprietary API is used. The configuration of particular modules must be modified in properties files because there is no support in GUI yet. The GUI is a standalone desktop application. Unfortunately web interface is not in this case provided because of some modules which need to have access to wiki's local files. The whole process of conversion in case of UWC starts by reading pages from source wiki engine and exporting them to local files if the files are not already accessible, then the regular expressions conversion to Confluence markup is performed and finally converted pages are stored to Confluence wiki using Confluence API. As a consequence UWC supports two kinds of modules: *exporters* (wiki data store -> local files) and *converters* (wiki markup -> Confluence markup).

Unfortunately neither of mentioned solutions is really universal and suitable for simple extensions in the sense of adding new functionality which does more than just conversion. Simply these solutions have no ambition to be more than converters.

2.2 Wiki Connector options

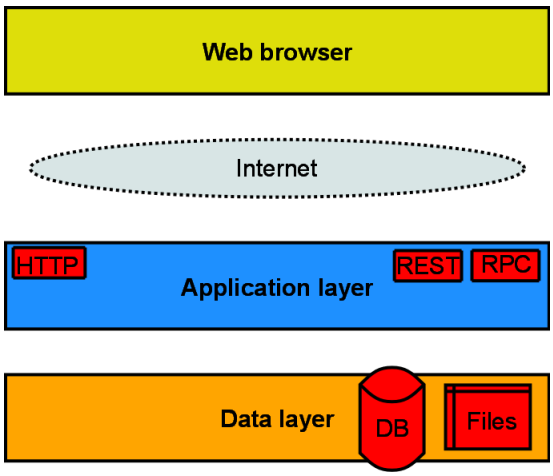


Figure 2.1: Architecture with possible connecting points

In this section I would like to focus on options how to connect to wiki engine to gather content from it. On Figure 2.1 it is depicted typical wiki architecture extended with possible connecting points in red.

Obviously, the easiest way how to connect to wiki is through some API which is provided by wiki engine. The expansion of wiki engines supporting some kind of API is growing. Common implementation protocol is in this case XML-RPC or REST interface. Such an API typically offers operations for:

- listing names of all the pages which are stored in the wiki engine
- loading page content (typically in engine markup, sometimes also in HTML) Sometimes it is also possible to load specific version of page - not only the actual one.
- loading page meta-information (name, author, version, last modification timestamp etc)
- authorization
- storing of a new content

Probably the most augmented API was published by JSPWiki ([5]). It is implemented using XML-RPC, currently is in version 2 and many other wiki engines implemented this API directly or through plugins:

- JSPWiki - direct support
- Usemod - plugin [3]
- TWiki - plugin [4]
- MoinMoin - plugin [5]
- Trac - plugin [6]
- DokuWiki - plugin [7]
- PHPWiki - plugin

This is the list of procedures suitable for purposes of this thesis (taken from [5]):

```
/* Returns a list of all pages.
 * The result is an array of utf8 pagenames. */
array getAllPages()
```

[3] http://www.decafbad.com/twiki/bin/view/Main/XmlRpcToWiki
[4] http://www.decafbad.com/twiki/bin/view/Main/XmlRpcToWiki
[5] http://www.decafbad.com/twiki/bin/view/Main/XmlRpcToWiki
[6] http://trac-hacks.org/wiki/XmlRpcPlugin
[7] http://www.dokuwiki.org/devel:xmlrpc

```
/* Get the raw Wiki text of page, latest version. */
utf8 getPage( utf8 pagename )

/* Get the raw Wiki text of page. */
utf8 getPageVersion( utf8 pagename, int version )

/* Returns a struct with elements:
 *  name (utf8): the canonical page name.
 *          lastModified (date): Last modification date, UTC.
 *          author (utf8): author name.
 *          version (int): current version
 */
struct getPageInfo( utf8 pagename )

/* Returns a struct just like plain getPageInfo(),
 * but this time for a specific version. */
struct getPageInfoVersion( utf8 pagename, int version )

/* Writes the content of the page.
 * The attributes-struct can be used to set any Wiki-specific things,
 * which the server can freely ignore or incorporate. Standard names are:
 *          comment (utf8): A comment for a page.
 *          minoredit (Boolean): This was a minor edit only.
 * Added in version 2.
 */
putPage( utf8 page, utf8 content, [struct attributes] )

/*
 * Returns version of the JSPWiki API - 1 or 2.
 * This procedure can be used for testing that connection to wiki is alive.
 */
int getRPCVersionSupported()
```

Note that there might be tiny differences among the implementations in some procedure signatures (specifically `putPage`'s `attributes` parameter) but more or less are the same. However, JSPWiki's API is not the only interface which might be used. Following engines support other kinds:

- MediaWiki API
  Very powerful REST interface. There is also a good wrapping Java library called Java Wiki Bot Framework [8] which might ease the use.
- XML-RPC interface of Confluence [9]
  Again, very powerful interface with similar operations like JSPWiki's. Also

[8] http://jwbf.sourceforge.net/
[9] http://confluence.atlassian.com/display/DOC/Remote+API+Specification

implemented by XWiki [10].

Question arises in case there is no API support. How should be the connection solved? There is variety of options.
First of all, many wiki engines use databases as a storage for their content. Naturally the schemas of different wikis are different and also database engines are different (MySQL, PostgreSQL etc.) but creating a database connector for particular wiki engine should not be hard. It should be even possible to create a generic database connector.
If the wiki doesn't use database as a storage it is possible to use the same approach as the Pm2Media Converter did. This converter simply simulates these operations by GETs and POSTs. Operation of listing all the pages can be achieved by parsing a wiki index page. Loading a page can be implemented as GETing an URL which contains page source (wiki page URL + parameter to display page source). Storing a page is simply achieved by POSTing needed content to corresponding URL together with fulfilled parameters. This process is much more problematic, not generic for more wiki engines, but still achievable. Unfortunately, this approach has many prerequisites: among others existence of suitable index page and ability of wiki engine to display page source.
If some of the prerequisites cannot be satisfied then there is an option to parse HTML (or other) source of displayed wiki pages. This should be always possible; however very complicated because the layout of the wiki pages is probable to change in the future.
For those wiki engines which use local files as a storage it could be possible to

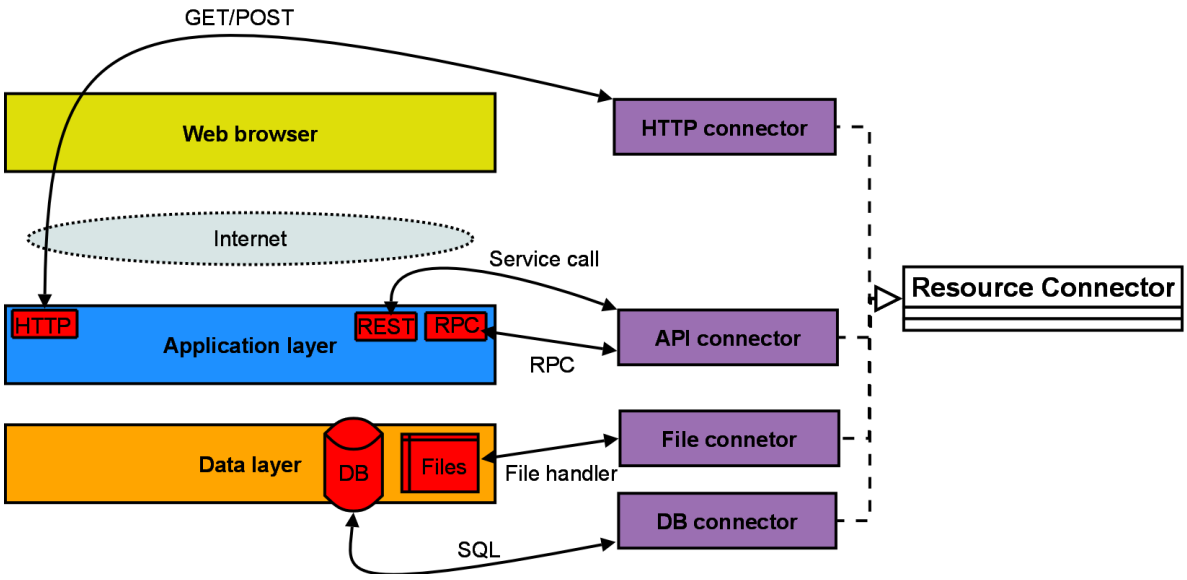


Figure 2.2: Architecture with connectors

implement a connector which will use these files but the connector must have access

[10]http://platform.xwiki.org/xwiki/bin/view/Features/XMLRPC

to them which might be a problem.
And the last option is to extend wiki engine functionality and implement some kind of API as a plugin for instance.

Now when the options how to connect to wiki engine and gather data from it are clear it is obvious that the connectors which will provide connection to wiki should implement common interface. This would enable other parts of application to use all the connectors the same way without any knowledge how the connection to the wiki is done. Figure 2.2 shows that all the connectors implement the same interface but use completely different connecting points, different protocols etc. Detailed description of the connector interface will follow in the next chapter.

2.3 Pages' comparison

Concerning pages' comparison the additional problem is caused by using different markup languages on different wikis. Two identical pages with the same visual content stored in two different wiki engines then might have completely different source code. This fact has very important consequence - pages cannot be compared according to their source codes. It means that typical textual diff algorithms cannot be used in this case. Pages' sources must be converted into a unified representation and the comparison must be done on this unified representation. For purposes of this project the unified representation is object model discussed in the forth chapter. Object model should contain operations to test equality of two instances and also some kind of diff operation.

It is obvious that the application will have to implement some parsers which will parse the source code of wiki pages and create a unified object representation of them and also some processors which will get the object representation and serialize them into different wiki markup. Whether both functions are implemented by the same module or not will be discussed later. Most of the parsers and processors will be engine-specific but there should be also some parsers and processors which can be used for more wiki engines. Creole parser and processor is one case.

2.4 Wiki Synchronization

For the purposes of this thesis migration is defined as a process of transferring complete content (all pages) of one wiki engine to another usually different wiki engine. Synchronization is the process of making sure that two or more different wiki engines contain the same up-to-date content. If a wiki page is added or changed in one wiki engine, the synchronization process will add or change the same page in the other engine. Note that deletion of pages is not in most of the engines possible; however it is definitely possible to set an empty content of a page. The difference between synchronization and migration is that during migration are transferred all the pages while during synchronization will be updated only pages which have changed or have been added. Migration is a special case of synchronization in which all pages have

changed. It is also expected that migration is usually performed only once whereas synchronization is performed periodically.

2.4.1 One-way vs. two-way synchronization

Wiki synchronization can be either one-way or two-way. In the one-way synchronization, also called *mirroring*, pages are copied only from a *source* wiki engine to a *destination* wiki engine, but no pages are copied back to the *source* engine. In the two-way synchronization, pages are copied in both directions, keeping the two engines in sync with each other. During two-way synchronization conflicts may appear. A conflict occurs when the same page is changed in both - *source* and *destination* - engines. Some conflicts can be resolved automatically in case that the changed parts of the pages are not overlapping. Otherwise user must resolve the conflict by combining the changes, or by selecting one change in favour of the other.
If we want to implement any type of synchronization we must solve a few issues.
First of all, it must be decided how to recognize that page content has changed at all. Wiki engines usually support some kind of versioning and save version numbers and/or last modification timestamps. The problem with version numbers is that the format of version number might differ engine by engine. Even if the format would be the same, it is not usually possible to store page content and set version number of this change. Let me demonstrate this on an example: we create a new wiki page (version 1) on one wiki and immediately edit it (version 2), then we perform synchronization with another wiki where the page didn't exist before. The page will be stored to the second wiki engine with version number 1. Next time we would want

	Wiki A	Wiki B
created page 'NewPage'	version 1	
updated page 'NewPage'	version 2	
'NewPage' stored to Wiki B		version 1

Table 2.1: Version issue

to synchronize these pages they will be again marked as non-synchronized because version numbers are different although pages actually are synchronized.

Using last modification timestamps is not much better. Firstly, nobody guarantees us that the times on both engines are the same. Another problem are different time zones. Times on both engines can be accurate but stored for different time zones. Let's suppose that we figure out both mentioned problems - we can ping both wiki engines at the beginning of the synchronization process, then compare both times to our local time and finally add or subtract the differences. Similarly to version numbers there is no way how to set modification timestamp of the change differently from the current time. This is not a problem in case of one-way synchronization because we will synchronize page only in case that the *source* last modification timestamp is higher than the *destination* one. In case of two-way synchronization we would need to have the times of synchronized pages equal on both engines which is not achievable.

2.4.2 Use Cases

Let's focus now on possible use cases of the tool which is being described in this thesis. Note that the goal is not to implement them all, but provide a suitable framework which can be extended to support them.

- Wiki migration
  It is necessary to migrate from one wiki engine to different one. This may be caused by many reasons: production environment has been changed (OS, web server, databases etc), current engine uses unsuitable markup, necessity of new currently unsupported functionality.

  It is necessary to migrate on different server.

- Backuping and exporting of content
  Wiki contains important information which must be backed-up but, unfortunately, most of wiki engines have no native support for it.
  Wiki content can be exported to whatever format is requested.

- Reporting
  Probably uncommon use case but it might be worth to know how many pages are stored in wiki, what their average size is, who the authors are, and who the most active contributors are etc.

- Replacements & filtration
  Sometimes it is necessary to filter out inappropriate content, SPAM or pages of particular author.
  Sometimes it is also necessary to rename some pages and update all the references which are pointing to them or replace some parts of the documents.

- Public & private wiki
  Many software development companies have own private wiki where can be stored information that should not be visible for customers. In case of one-way synchronization public wiki must be read-only.

- Mirrored wiki
  There exists one main wiki engine which is writeable and its readable mirrors.

Use cases where two-way synchronization is required:

- Off-line editing of wiki content
  One of the main drawbacks of wikis is that it is possible to modify their content only when we are connected to the network. It would be great to have a tool which allows us to modify content even when we are off-line and then synchronize the changes when connected.

- Merging of more wikis together
  Subteams of a huge project use their own wikis to store information related to their part of the project and it is necessary to consolidate it in one official project wiki.

- Replicated wikis
  Similarly to replicated databases, replicated wiki can be used in case it is necessary to improve its reliability, fault-tolerance or accessibility. Changes can be made to whichever wiki and then they must be propagated to the other nodes.

- Distributed or Proxy wiki
  Subwikis are accessible via one proxy. Pages are loaded on demand and changes made on proxy are propagated to particular wiki.

Since there exist useful use cases of two-way synchronization and because using of last modification timestamps or version numbers are not optimal solutions we should focus on additional possibilities.
One possible option would be comparing content of the page on both wiki engines. Based on the comparison it can be easily decided whether the content has been changed or not. However, if it has been changed, there is no way how to recognize on which wiki and thus the change will be always marked as a conflict. This could be partially solved by storing version of previous synchronization to the content of the page, but it is definitely not an ideal solution. Another problem is that this approach has pretty big net traffic overhead - page must be downloaded from both wikis even if it hasn't been changed. If we suppose that average size of page including protocol overhead is 10 KB, wiki contains 1000 pages and we have to download the pages from both wikis, the overall traffic is something like 20 MB which is not negligible especially if there were changed five pages of total size 50 KB.
The best resolution of this problem is introduction of some kind of caching. It should not be a problem to create a cache which will contain a version number of every page stored during previous synchronization. Next time wikis will be synchronized it will be easy to recognize which pages have been changed and which have not by comparing version numbers from previous synchronization with current version numbers.

2.5 Issues not solved

In this section I would like to mention things which are not solved in this thesis. The reason is that they are too complex and beyond the scope of this thesis or the wiki engines don't provide adequate support for their solution right now.

2.5.1 Attachments

Since the attachment support is very poor in wiki APIs, I have decided not to devote them so much extent in current version of the application which is being described in

this thesis. Nevertheless, the architecture and object model should be prepared for them when their support will be improved in existing APIs. Attachments also bring new issues to be resolved such as their downloading and uploading (size issue), their comparing and versioning (they are binary files) etc.

2.5.2 Locking

Another issue which cannot be addressed is locking of wiki content when synchronization or migration algorithm is executing. Again, there is no support for page locking in wiki APIs. Simulating of such functionality would be very complicated or even impossible in some cases because there is no entity to lock. The end user must take this feature into account and disable access to wiki during algorithm execution if he wants to avoid errors such as lost updates.

Chapter 3

Architecture

In this chapter I will focus on discussion of architecture options and description of finally chosen architecture.

3.1 Discussion of options

First of all, let me introduce a few requirements which the chosen architecture should comply:

- existence of unified object model
  Wiki pages should be translated to this unified object representation and vice versa. As discussed earlier comparing of pages is possible only in the unified representation, not with the source code.

- algorithms must be generic
  To achieve necessary generality of algorithms, they should operate only with the object model. No assumptions about the markup language of particular wiki engine are acceptable. No assumptions about the wiki engine are suitable. Actually, algorithms should not care whether they operate with wiki or some other resource.

- must be extensible
  The project must provide a way how to extend it to achieve a long life of it. It should be possible to extend project with support for new wiki engines, new algorithms etc.

- extensions should be simple
  Extensions must be simple enough to attract incoming contributors. Granularity of modules must be chosen appropriately. Implementation of four modules to add new basic functionality is not the best choice. However, repeating the same code in every module is also inappropriate.

- modules should be reusable
  Modules should be designed the way that they can be used by different algorithms and if possible for different wiki engines.

- common functionality at one place
  The goal is that the functionality which is common to all modules is implemented at one place and not by every module. This will reduce redundancy and thus modifications of this common functionality will be done only at one place.

- modules should be loosely coupled
  The intent is that every module knows as less as possible about other existing modules. This reduces dependencies among modules and makes the design more flexible and easier to modify.

- separation of backend modules from higher layers
  Every evolving project gets to the point that the design must be somehow adjusted. The goal is to propose such an architecture which will allow future changes of some parts without changing all the other. E.g. changes done in backend modules will affect higher layers which use the backend as less as possible and vice versa.

The description of possible solutions of backend layers with the discussion of advantages and drawbacks follows; Backend is meant to be the part of the architecture which is responsible for connection to the wiki engine and creation and processing of unified object model.

- monolithic backend (Figure 3.1)
  This is the simplest solution which comes to mind. The whole backend would

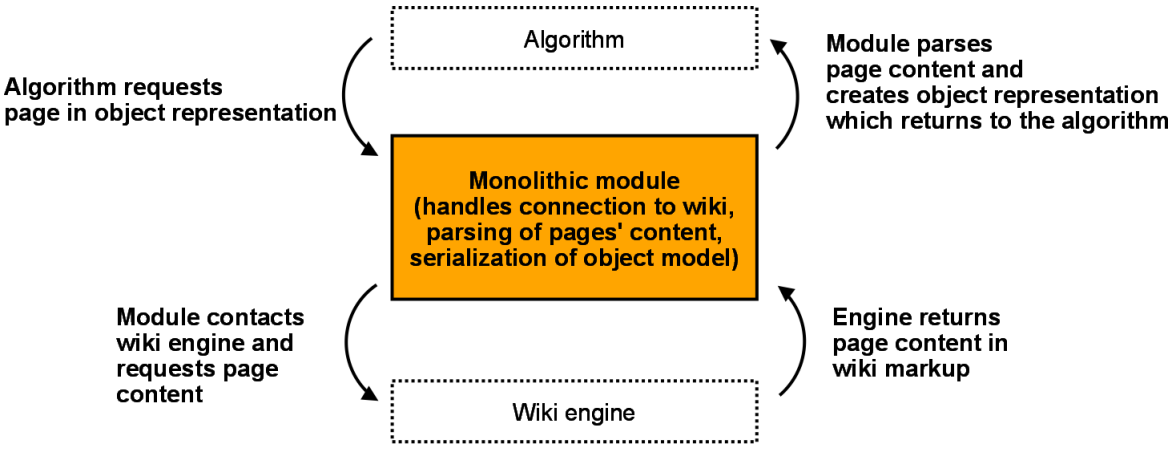


Figure 3.1: Monolithic backend

be consisted of one module per every wiki engine. Such a module would be responsible for creating some kind of connection to the wiki engine (XML-RPC,

REST, JDBC etc.), parsing of pages' content and creating of object representation and also for serialization of object model into the engine markup. Such an approach would have an advantage that all the engine specific functionality would be stored at one place. However, extensibility, reusability and flexibility of this solution would be very poor. In the previous chapter I've analyzed that many wiki engines use very similar XML-RPC interface and also the Wiki Creole markup is being spread among more and more wikis. In case of this solution it would mean that XML-RPC and Creole part of the module would be redundant in many modules. Also implementation of such monolithic solution would be more complicated and thus non-attractive for incoming contributors. On top of that this solution provides very poor flexibility of design. If we want to change the interface of this backend we will have to change all the modules which were already implemented. Similarly adding new simple functionality like logging to the backend would mean to add it to all the modules.

- separated resource connection and model building & processing (Figure 3.2)
  This solution divides previous monolithic backend into two parts: resource

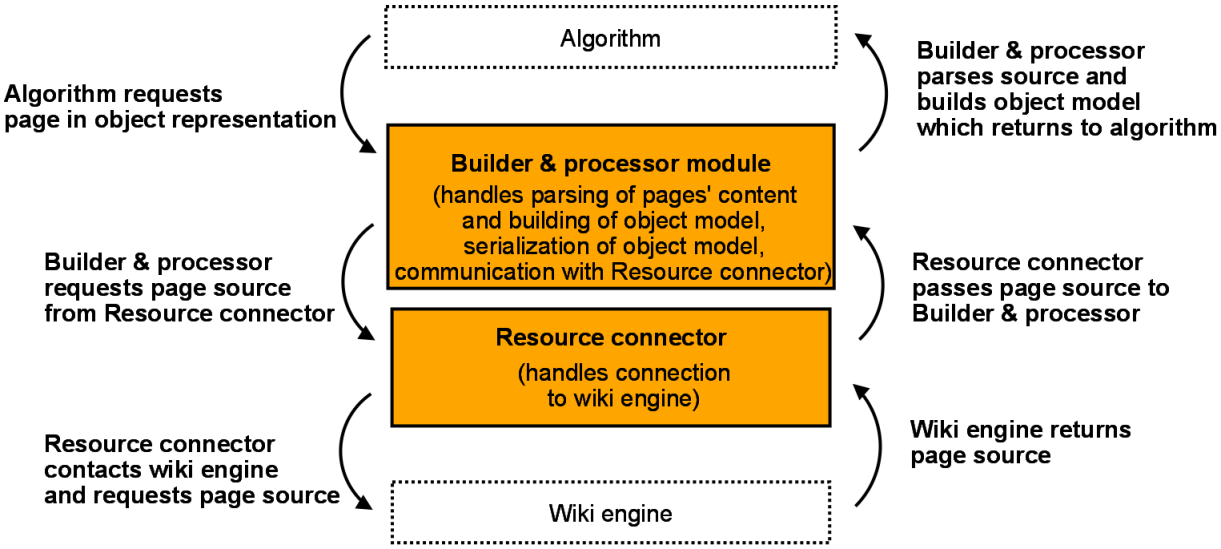


Figure 3.2: Separated connector and builder & processor

connection which may be the same for more wiki engines because of similar XML-RPC interfaces and model building and processing part which may be the same thanks to spread of Creole markup language. Technically it would mean that the backend interface would be implemented by model building & processing module which would make use of resource connector module. This solution decreases redundancy issue of previous approach because some modules like XML-RPC module might be reused for more wiki engines. Also the split of one big module into two smaller more specifically oriented makes the implementation simpler. However, even this solution is not flexible enough. Actually the flexibility issue remains the same as in previous case. There is again no integration point which would provide us with possibility to add

common functionality to all modules without changing them all or give us an option to easily redesign the backend interface with limited number of changes which have to be propagated to all modules.

- architecture with facade (Figure 3.3)
  This option brings the necessary integration point which was discussed in the previous option - the facade. The facade integrates all the resource connection, model building and processing modules together and brings one integrated backend operations interface to the higher layer. It means that algorithms don't have to care which module provides which operations, they do not even know it, because algorithms and modules would be completely separated from each other. This approach would offer really huge flexibility in the sense of future changes. The whole application backend can be completely redesigned and no changes should be propagated to the higher layer. Similarly no changes made in higher layer should affect backend modules. Another big advantage is that all the backend modules may also be separated from each other. It is possible to define independent modules which know nothing about each other and facade would take care of integration of these modules together to achieve the required operations functionality. It is possible to add completely new modules and define operations interfaces which would facade provide to higher layer - again with no affect to it. On top of that facade might implement common functionality to all modules like logging or synchronization, which would make modules much simpler to implement because it would allow us to set the level of granularity of modules as we want, modules don't have to take care of thread safety and logging, facade can provide thread pools for asynchronous operation calls etc. All this could be achieved with facade separating application backend from higher layer.

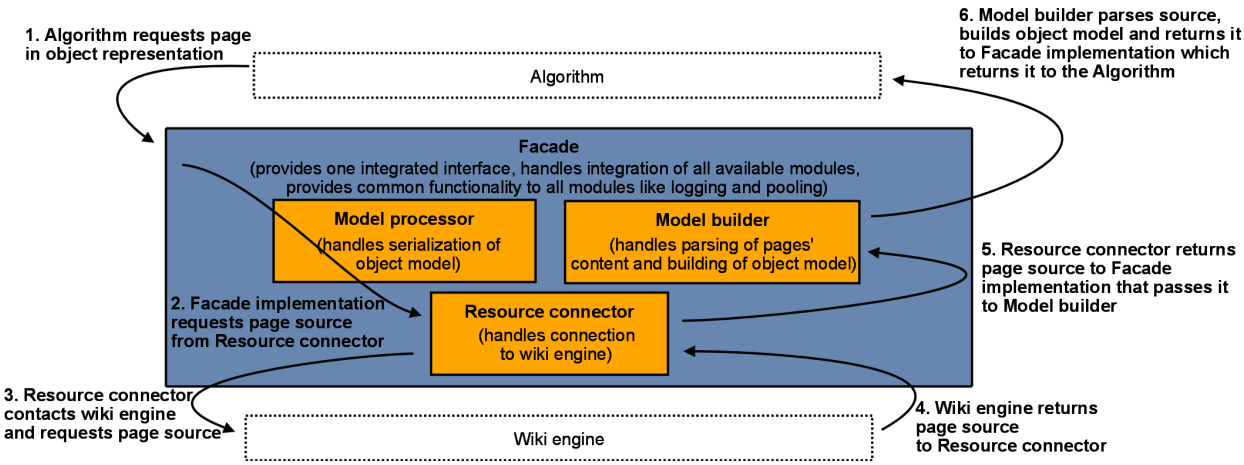


Figure 3.3: Architecture with facade

- component architecture (Figure 3.4)
  This option could provide almost unlimited flexibility. It is different approach

than the previous cases. In this case modules would be rather components. Every component provides some interfaces and it also may require some interfaces of other components. In this case achieving the required functionality would mean to chain a few components together. Obviously there must be one predefined component which would offer the backend interface used by higher layer and delegate jobs to other components but that is all. It is the only expectation. All the other components may provide and require whatever operations they want. However, such an uncontrolled growth might be also dangerous especially in open source project. The end user of the application would have to chain the components together and in case that for different wiki engines different component chains would have to be set it would be chaotic. To avoid this it would be probably necessary to define a pattern of chained components which should be used. However, it means that the main advantage - unlimited flexibility would be lost. This option has another drawback - it has no integration part, no place where common functionality could be implemented. Sure, the component which provides the backend interface might be used as an integration component of all the other components but then the backend component would do the same job as facade in previous option.

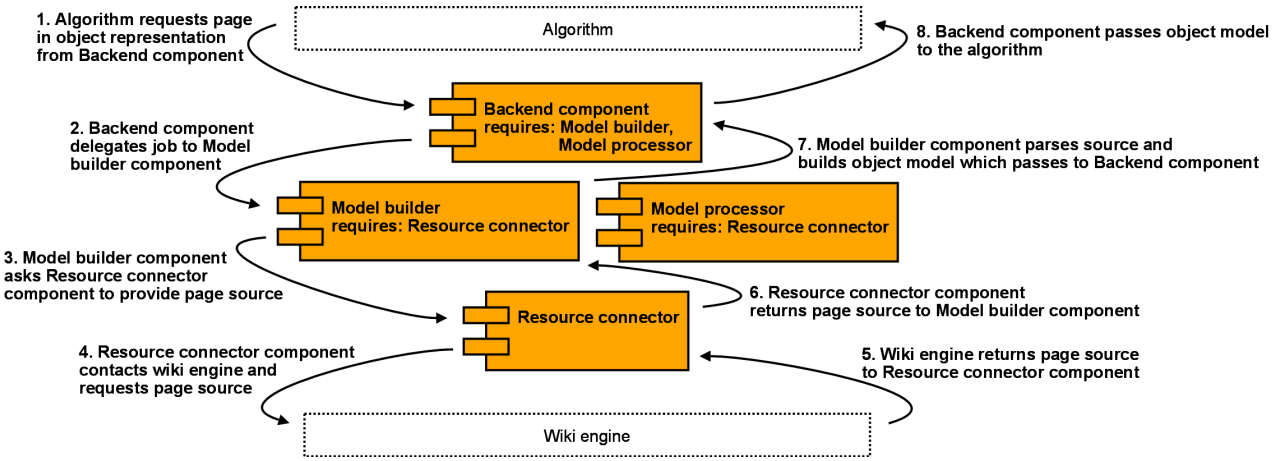


Figure 3.4: Example pattern of chained components

Table 3.1 contains summary of fulfilment of all the requirements by presented architectures.

3.2 Detailed architecture overview

Based on the fulfilments of requirements I've decided that the best solution is the architecture with facade which should comply all of the presented requirements. On the Figure 3.5 it is depicted detailed architecture example. Algorithm communicates only with source and destination facades. The number of facades required by algorithm can be different. It is the algorithm who states how many facades it requires

	monolithic	separated	facade	components
unified object model	X	X	X	X
generic algorithms	X	X	X	X
extensible	X	X	X	X
simple extensions		X	X	X
reusable modules		X	X	X
integrated common functionality			X	
loosely coupled modules	X		X	
separation of backend from higher layers			X	

Table 3.1: Fulfilment of requirements by architectures

and which operations should be supported by particular facades. Algorithm also uses unified object representation of wiki documents which gathers and passes to source or destination facades. It means that the algorithm has no idea what wiki engine or wiki markup is used, it can be totally generic and suitable for all kinds of source and destination wiki engines. Each of the facades takes care about integration of available

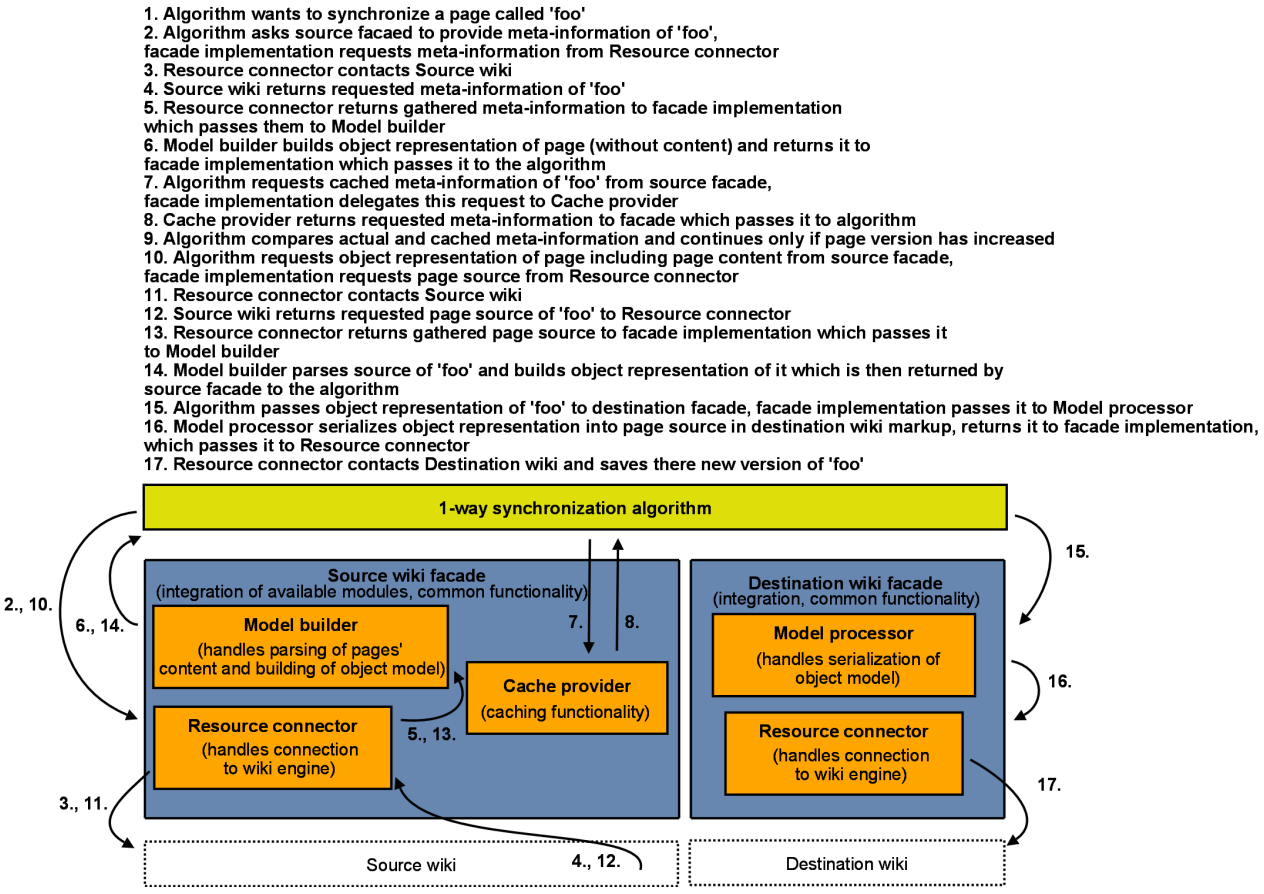


Figure 3.5: Architecture overview (model situation from one-way synchronization algorithm)

modules and provides necessary separation between algorithms and modules. This is also important, algorithms don't know anything about the modules which are used, they just say which operations are required by facades. This brings huge flexibility to the architecture and allows to completely redefine some parts of the design without propagation of these changes to other parts. Modules which are available to particular facades are then chosen based on required operations requested by algorithm. It means that if algorithm doesn't request cache operations for a facade, the Cache provider module doesn't have to be available to that facade. Basically, there are operations for loading of wiki content, storing of wiki content and caching. Supported modules are Resource connector which is responsible for connection to wiki engine, Model builder which parses page source and builds object representation of it, Model processor which is the opposite of Model builder and Cache provider which offers caching functionality.

Detailed description of supported operations, supported modules and list of modules which are required for implementation of particular operation will follow in the next sections of this chapter. Generated documentation of all classes and interfaces which will be presented is accessible on the thesis CD in the `javadoc` directory.

3.3 Operations facade

Let's start with operations which are currently supported. All the operations are descendants of interface `Operation`. `Operation` is an empty interface (no methods, no constants) and is useful for two reasons. Firstly, if we need to add common functionality to all the operations in the future we can put it into the Operation interface and secondly, it is very useful when working with generics. This way it is possible to easily recognize classes implementing operations, because they extend `Operation` interface (`Class<? extends Operation>`). From the Figure 3.6 it is obvious that load

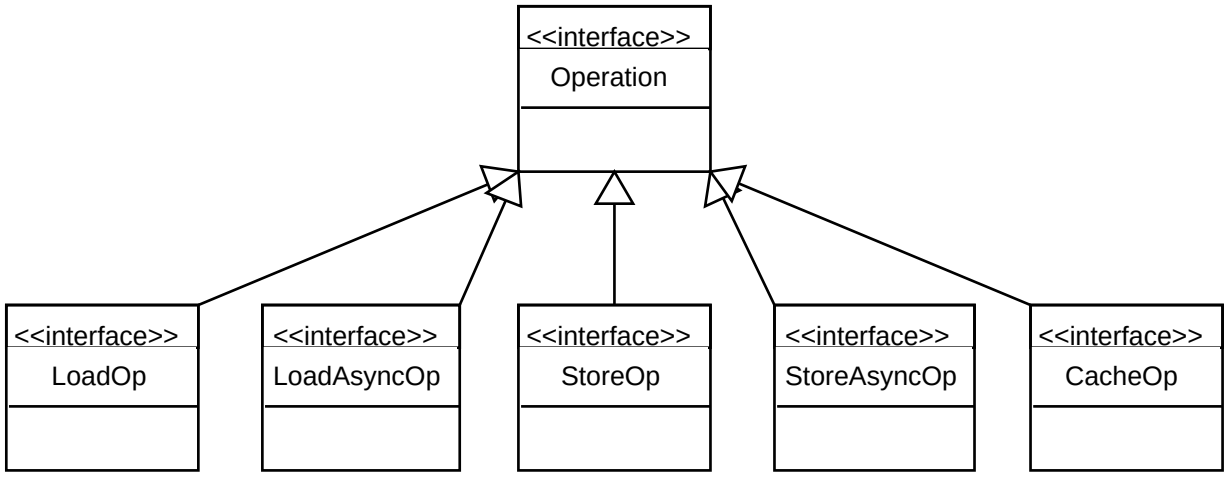


Figure 3.6: Current operation interfaces (without methods)

and store operations are provided in two variants - synchronous and asynchronous.

This is one of the features that are provided directly by Operations facade. Modules which are used to compose required functionality have only synchronous variant which makes them much simpler to implement. Asynchronous variant is provided by Operations facade. Operations facade has its own thread pool. It means that whenever asynchronous operation is called the request is placed to the thread pool queue and then served via synchronous module call.

Asynchronous operations are very useful if we want to offer algorithms optimized for network latency. The most of the time the algorithm is being executed is spent by waiting for response from remote API or database. However, if the algorithm uses asynchronous calls the requests are queued and then served by multiple threads in parallel. This way the total execution time of algorithm reduces significantly.

Besides thread pool Operations facade pools also modules. When Operations facade is created a number of modules is initialized and placed to the module pool. When Operations facade serves one of the operations, pooled modules are used. There is one exception - stateful modules. Most of the modules are stateless. However, there also exist stateful ones like Cache provider which provides operations for manipulation with cache. In case of these stateful modules only one instance of module can be pooled. The way these stateful modules are recognized will be described in the following sections of this chapter. Another very useful benefit of Operations facade is that it provides thread safety for module calls which consequently means that modules don't have to care about synchronization issues. There can be much more common functionality provided by Operations facade which can be implemented in the future.

In addition to all the methods defined in `Operation` interfaces `OperationsFacade` interface has one extra operation `terminateAll`.

```
public interface OperationsFacade extends CacheOp, LoadAsyncOp,
        LoadOp, StoreAsyncOp, StoreOp {
    void terminateAll();
}
```

Operation `terminateAll` is useful in cases when it is necessary to stop processing of all the requests. Besides stopping the requests it also clears all the queues of asynchronous requests waiting to be processed.

Note that adding a new kind of operations is very simple - just add another child of `Operation` interface and provide implementation in Operations facade using existing or by adding new modules, which definitely cannot harm neither previously implemented algorithms nor existing modules. Similarly it is possible completely redesign backend modules and no changes will affect algorithms because they will still use the same backend interface. Or vice versa reimplementation of all the algorithms and even changing backend interface should not affect modules implementations, just Operations facade which integrates them. This makes the design amazingly flexible.

3.3.1 Load operations

The `LoadOp` interface contains all synchronous operations for page loading:

```
public interface LoadOp extends Operation {
    public enum LoadType {META_ONLY, ALL}
    void listPagesNames(ResultsHandler handler, ResultsFilter filter);
    WikiPage loadWikiPage(LoadType type, String pageName);
}
```

The intent of `LoadOp#loadWikiPage` is to create an object representation of a page specified by its name based on page meta information (version, author, last modification timestamp etc.) and in case of `LoadType.ALL` also page content. The signature of method `listPagesNames` looks more complicated. The first intent was that the method signature will be simple, like this:

```
List<String> listPagesNames();
```

However, such an implementation would have many drawbacks. First of all, the returned list of pages' names could be pretty huge and could cause out of memory errors etc. Obviously, not in case of synchronizing wikis with thousand pages, but why should be wiki size which is synchronized somehow limited. English version of Wikipedia has more than 2.5 millions of pages. The second reason is that the code calling the `listPagesNames` method must wait till the method ends its invocation and after that is able to process the returned list of results. Much better solution is to process the page name immediately when it becomes available. It doesn't have to be always possible but in case of e.g. JDBC's `ResultSet` it is possible to handle results one by one instead of copying results to the list and then returning it. That's why `listPagesNames` has `ResultsHandler` as one of the parameters. `ResultsHandler` is a very simple interface, just with one method:

```
public interface ResultsHandler {
    public boolean handle(Object obj);
}
```

The method `handle` is called whenever another page name becomes available and the page name is passed as a parameter. Then it is up to the `ResultsHandler` implementation what will with the page name do. It can add it to a list of pages names or immediately process it. The `handle` method returns `boolean` value which means whether the `listPagesNames` method should continue handling another results. This is another benefit of `ResultsHandler` because it has an ability to interrupt the `listPagesNames` call. There may be many reasons to do that, e.g. you just want to return only first 20 page names. Here is an example of anonymous implementation of `ResultsHandler` in `listPagesNames` call:

```
void listPagesNames(new ResultsHandler() {
        public boolean handle(Object obj) {
                // here it is possible to immediatelly process the page name,
                // eg: call loadWikiPage(LoadType.ALL, (String)obj)
                // or add it to the list of pages' names
        }
}, null);
```

The second parameter of `listPagesNames` is `ResultsFilter`. The intent of `ResultsFilter` is to filter out results which should not be even handled. `ResultsFilter` interface is also very simple:

```
public interface ResultsFilter {
    public boolean filter(Object obj);
}
```

`filter` method is called before `ResultsHandler#handle` and only if `filter` returns `false ResultsHandler#handle` is called. Obviously, the functionality of both interfaces could be merged just into one `handle` method (which would also perform filtration); however, it is much more synoptical to separate them - then it is possible to implement different generic filters and handlers and combine them in the application.

`LoadAsyncOp` interface is very similar to `LoadOp` except that the operation is asynchronous:

```
public interface LoadAsyncOp extends Operation {
    void loadAsyncWikiPage(LoadType type, String pageName,
                LoadListener listener);
}
```

The result of `loadAsyncWikiPage` operation is the same as in case of synchronous variant. However, caller is informed about the result via `LoadListener` instead of return value of the method. `LoadListener` is again a very simple interface and it is expected to be implemented by the caller of asynchronous load operation and the reference to it should be passed to `loadAsyncWikiPage` method.

```
public interface LoadListener {
    void onLoadingWikiPage(LoadType type, String pageName);
    void onLoadedWikiPage(LoadType type, WikiPage page);
    void onLoadErrorWikiPage(LoadType type, String pageName, Throwable t);
}
```

Method `onLoadingWikiPage` is invoked whenever the processing of load request is started. The `onLoadedWikiPage` method is invoked whenever the page is loaded and the result of operation - created object representation - is passed to it. In case an error in loading process appears, the `onLoadErrorWikiPage` is invoked.

Now it is worth to say which modules are required to be available to Operations facade for implementation of `LoadOp` and `LoadAsyncOp` operations. It is Resource connector module and Model builder module. Figure 3.7 describes how Operations facade integrates these two modules and implements `loadWikiPage`.

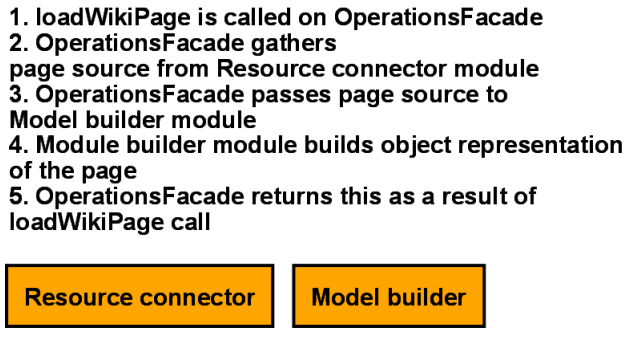


Figure 3.7: Required modules for LoadOp and LoadAsyncOp

3.3.2 Store operations

Store operation interfaces are very similar to load operation interfaces except that the object representation of wiki pages which should be serialized to wiki markup and stored on wiki are passed as a parameter.

```
public interface StoreOp extends Operation {
    void storeWikiPage(WikiPage page);
}
```

Asynchronous variant of store operation:

```
public interface StoreAsyncOp extends Operation {
    void storeAsyncWikiPage(WikiPage page, StoreListener listener);
}
```

Similarly to asynchronous load operation also asynchronous store operation is informed about the result of operation via listener. `StoreListener` interface again defines three methods - one for start of processing of the store request, one when store request is finished and one for errors which can appear during the request.

```
public interface StoreListener {
    void onStoringWikiPage(WikiPage page);
    void onStoredWikiPage(WikiPage page);
    void onStoreError(WikiPage page, Throwable t);
}
```

Operations facade requires Resource connector module and Model processor module to be available for implementation of store operations. Figure 3.8 describes integration of modules and implementation of `storeWikiPage`.

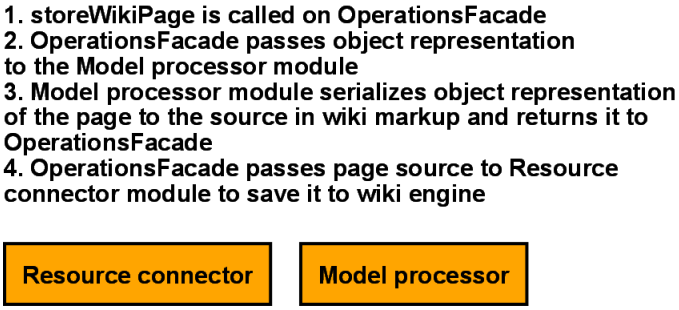


Figure 3.8: Required modules for StoreOp and StoreAsyncOp

3.3.3 Cache operations

The purpose of cache operations is to provide access to cached pages' meta-information and possibility to update the cache. Despite all previous operation interfaces `CacheOp` interface is not stateless and contains initialization (`loadCache`) and finalization (`storeCache`) methods:

```
public interface CacheOp extends Operation {
    void loadCache(CacheUid uid);
```

```
    WikiPage loadCachedWikiPageInfo(String pageName);
    void cacheWikiPageInfo(WikiPage page);
    void storeCache();
}
```

`CacheOp#loadCache` method should be invoked before any other method to initialize the cache identified by `CacheUid` parameter. Opposite method to `loadCache` is `storeCache` which should be run at the end of work with the cache to flush cache content to the cache storage. Methods `loadCachedWikiPageInfo` and `cacheWikiPageInfo` are used for manipulation with cache.

Cache provider module is the only required module to be available to Operations facade for implementation of cache operations. Figure 3.9 describes `loadCachedWikiPageInfo` implementation.

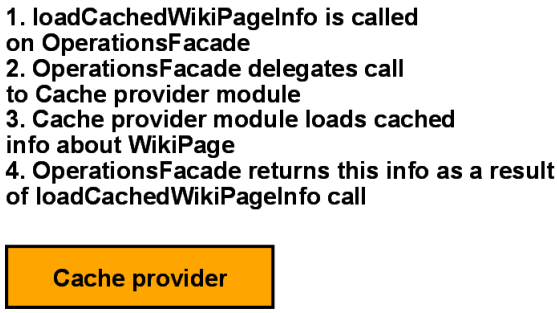


Figure 3.9: Required modules for CacheOp

3.4 Modules

Similarly to operations, all modules are descendants of `Module` interface. `Module` interface defines two methods:

```
public interface Module {
    void init(Configuration conf);
    void dispose();
}
```

`init` method should initialize module with the required module configuration and is expected to be invoked after the module is created. This method should also include some test functionality which checks that passed configuration is correct, e.g. test that connection to wiki can be created in case of Resource connector module. Descendant of `Configuration` interface is expected as configuration parameter of `init`. `Configuration` interface is another empty interface which is expected to be implemented by module configuration bean which contains all the configuration properties that are necessary to initialize particular module. For example when a module needs username, password and apiURL as its configuration properties, configuration bean would like as follows.

```
public class MediaWikiProviderConfiguration implements ProviderConfiguration {
    private String username;
    private String password;
    private URL apiURL;

    public String getUsername() {
        return username;
    }

    @ConfigurationProperty(required=false, order = 2, name= "User name")
    public void setUsername(final String username) {
        if (username == null) {
            throw new NullPointerException("User name can't be null.");
        }
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    @ConfigurationProperty(required=true, order = 3, name= "Password")
    public void setPassword(final String password) {
        if (password == null) {
            throw new NullPointerException("Password can't be null.");
        }
        this.password = password;
    }

    public URL getApiURL() {
        return apiURL;
    }

    @ConfigurationProperty(required=true, order = 1, name= "API URL",
        desc="URL of the REST interface provided by the wiki engine.
                        It is typically available on http://wiki/api.php.")
    public void setApiURL(final URL apiURL) {
        if (apiURL == null) {
            throw new NullPointerException("apiURL can't be null.");
        }
        this.apiURL = apiURL;
    }
}
```

Note that `ProviderConfiguration` is an empty interface which extends `Configuration` interface and is meant to be configuration super interface for all Resource connectors' configurations. From the source code it is also noticeable that setters of configuration properties are annotated with `@ConfigurationProperty` annotation. This annotation adds another specific meta-information to the property and is useful especially in integration with UI. Configuration properties can be required or optional (`required` element), can be displayed in particular order (`order` element) and as a display name is used value specified in a `name` element of `ConfigurationProperty` annotation. Annotating properties with this annotation is not mandatory, however

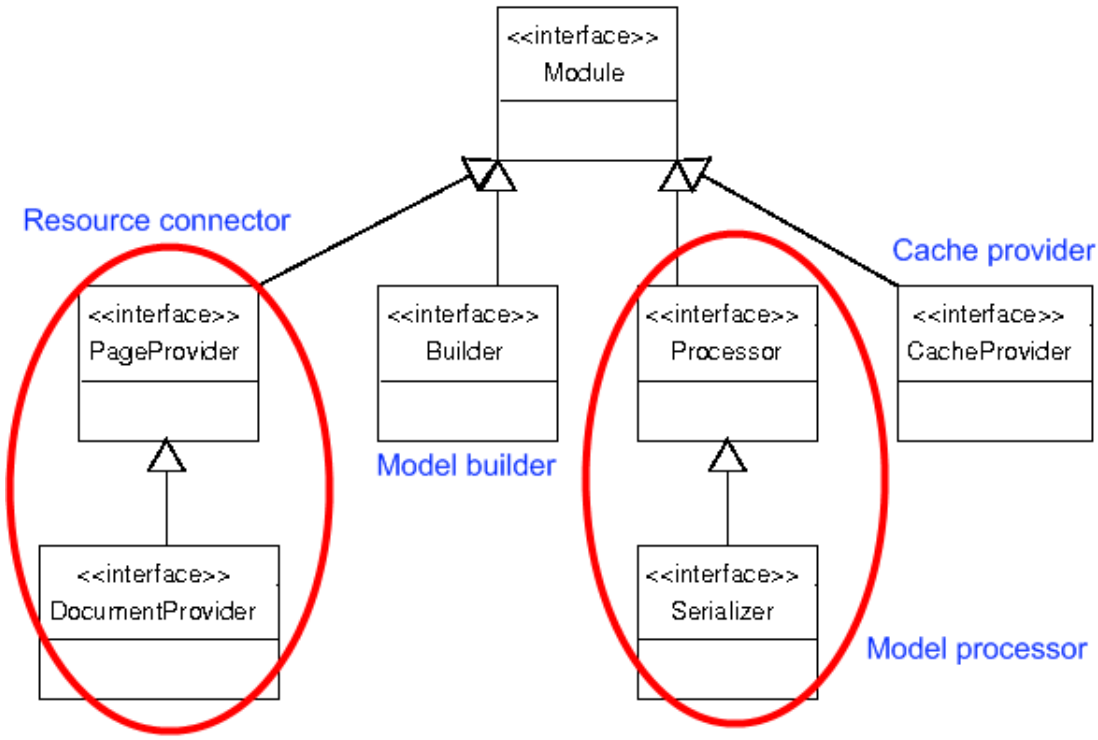


Figure 3.10: Current modules with names of interfaces

is more than advised. If the annotation is missing defaults are used - `required` = `true`, `order` is undefined, the display name is property name and `desc` is empty.

My initial idea was that instead of configuration beans the `Map<String, String>` template will be used, where the first element is a property name and the second is its value, but this approach has many drawbacks. It offered poor binding to the UI - no display names, no ordering of properties, no required/optional properties, no description of properties. Type conversion from `String` would have to be done in modules and it would have to be solved which properties are mandatory and which not. In case of configuration beans all this is very simple when Java reflection is used.

Every module implementation should also be annotated with `@ModuleInfo` annotation. This annotation has following elements: `name`, `description` and `stateless`. `name` and `description` elements are mainly used by UI. If they are not provided then class name is used as module name and description of module remains empty. The `stateless` element is used by Operations facade implementation to distinguish stateful modules which cannot be pooled. By default all modules are stateless. Here is the sample usage of `@ModuleInfo` annotation:

```
@ModuleInfo(name="JSPWiki", description="Resource connector implementation
                which takes advantage of JSPWiki's XML-RPC interface v.2 which
                is defined on http://www.jspwiki.org/wiki/WikiRPCInterface2.",
                stateless=true)
public class JSPWikiPageProvider implements PageProvider {
        // module implementation
}
```

3.4.1 Resource connector

Resource connector is a module which hides the differences among different wiki APIs and provides pages' content in native wiki markup to the other modules. In case that wiki doesn't provide API different approach can be used: connect directly to wiki database, make use of wiki web UI to display page's source or use files in which wiki stores its content. It is the task for the author of a module how to get to the content of the wiki. Resource connector is spread to two interfaces - `PageProvider` and `DocumentProvider`. `PageProvider`'s operations involve only page manipulation whereas `DocumentProvider` should be implemented in case the author finds a way how to also work with attachments. Firstly, let's look at the `PageProvider`:

```
public interface PageProvider extends Module {
    void listPages(ResultsHandler handler, ResultsFilter filter);
    String loadPageContent(String pageName);
    PageInfo loadPageInfo(String pageName);
    void storePageContent(String pageName, String content);
}
```

The meaning of `listPages` operation is the same as `LoadOp#listPagesNames` and also the `handler` and `filter` parameters are the same. Actually, `LoadOp#listPagesNames` is just the delegation to `listPages`, however facade can cover the call with some additional functionality. `loadPageContent` operation should return page content of specified wiki page in wiki's markup language. `storePageContent` is a contradiction to `loadPageContent` and it is expected to store the page content which is already translated to correct wiki markup. And, finally, `loadPageInfo` returns `PageInfo` structure containing: page name, page author, version, last modification timestamp etc. Most of the attributes of this structure are optional except for page name and last modification timestamp.
`DocumentProvider` interface extends the functionality of `PageProvider` with support for attachments:

```
public interface DocumentProvider extends PageProvider {
    void listAttachments(ResultsHandler handler, ResultsFilter filter);
    byte[] loadAttachment(String attachmentName);
    AttachmentInfo loadAttachmentInfo(String attachmentName);
    void storeAttachment(String attachmentName, String pageName,
            byte[] content);
}
```

The function of `listAttachments` is obvious. `loadAttachment` operation is supposed to return the bytes which form the attachment content, `storeAttachment` is expected to store the attachment. And `loadAttachmentInfo` should return an info structure containing: attachment name, size, version etc. - again mostly optional.

3.4.2 Model builder

Model builder is a module whose responsibility is to hide the differences among wiki markups or other formats used for storing of wiki content. The input of this module is a page source in known format - particular wiki markup, some XML or binary

representation - and the output is unified object representation of that page. Builder interface is even simpler than in case of Resource connector:

```
public interface Builder extends Module {
    WikiPage build(String pageSource, PageInfo pageInfo);
}
```

The only method `build` is expected to get the page source in particular format and `PageInfo` structure (name, last modification date, author, version, comment, size) containing at least mandatory page name and last modification timestamp meta-information and produces `WikiPage` object. `WikiPage` is an object representation of the page. The detailed description of the object model will follow in the next chapter.

3.4.3 Model processor

Model processor is a module for processing of the object model. Object model can be processed in many different ways. It can be serialized to a textual representation - HTML, XML, markup etc. HTML representation of wiki content can be used for presentation purposes for instance. As mentioned in the beginning of this thesis, one of my personal goals for this thesis is to provide easily extensible architecture which can be used for many various purposes, not only migration and synchronization of wiki content or definition of unified object model. One of the possible extensions might be off-line wiki editor and it would definitely need to present wiki content in some user-friendly format. XML representation can be used e.g. for exporting purposes - one application will export the content to defined XML document and another application can import it. Purpose of markup serialization is obvious and it is used during migration and synchronization of wiki pages on different wiki engine.

Object representation can also be serialized to binary representation. This binary representation can be used for instance when saving object model to files. XML representation would be also suitable for this purpose, but binary representation would be much smaller (less bytes).

Object model of wiki content can also be somehow retransformed. Some elements of the model can be replaced with other elements. Some can be completely removed, updated or added. This retransformation processing could be used in case it is necessary to filter out SPAM from wiki content (by author, by SPAM patterns, ..), divide or join several pages together, add table of content on wiki pages, rename wiki page and update references to that page, replace some page elements with other elements etc.

Object model can also be processed by some kind of reporting processor which will collect information about the content such as number of wiki pages, number of attachments, most active authors, newest documents, identification of draft documents etc.

More and more examples of Processor implementations could be presented but the most important one for purposes of this thesis is the serialization to destination wiki markup.

`Processor` interface contains one `process` method per each model element. Each of the process methods returns `Object`. The reason for such generic choice is the

existence of many use cases of model processor. Processors suitable for serialization will probably return `String`, processors which are used for model transformations will probably return the adjusted element which was originally passed as a parameter.

```
public interface Processor extends Module {
    Object process(WikiPage page);
    Object process(WikiList list);
    Object process(WikiTable table);
        .....
}
```

To distinguish different kinds of processors, subtypes of `Processor` interface are introduced. One of them is `MarkupSerializer` interface which is an empty interface but it is expected that implementations of this interface will return `String`s. Actually the `String`s produced by `MarkupSerializer` interface should be the textual representation of model element in particular wiki markup.

```
public interface MarkupSerializer extends Processor {
}
```

Similarly to serializers it would be possible to define for example `Transformer` interface whose expected return values are the objects passed to the process method.

3.4.4 Cache provider

Cache provider is a module which provides the necessary functionality for `CacheOp`. The implementation of `CacheOp` by Operations facade actually just delegates `CacheOp` operation calls to this module.

```
public interface CacheProvider extends Module {
    void loadCache(CacheUid uid);
    WikiPage loadCachedWikiPageInfo(String pageName);
    void cacheWikiPageInfo(WikiPage page);
    void storeCache();
}
```

3.5 Algorithms

Integral part of presented architecture are algorithms. Algorithms make use of the mentioned modules through `OperationsFacade`s. Every `OperationsFacade` should stand for an entity used by algorithm - source wiki, destination wiki, cache. Algorithms must implement interface `Algorithm`:

```
public interface Algorithm {
    void init(OperationsFacade[] facades, String profile);
    AlgorithmContext getContext();
    void start(AlgorithmListener listener);
    AlgorithmState getState();
    void terminate();
}
```

Similarly to modules also algorithms have initialization method - `init`. All the `OperationsFacade`s that algorithm requires are passed to this initialization method. Every algorithm should be annotated with `@Facades` annotation. `@Facades` contains array of `@OperationSet`. `@OperationSet` is another annotation which defines which operations are required and which optional for the particular facade.

```
@Facades({
    @OperationSet(name = "Source Wiki", required = { LoadAsyncOp.class,
        LoadOp.class}, optional = {}),
    @OperationSet(name = "Destination Wiki", required = { StoreAsyncOp.class },
        optional = {})
})
public class Migration implements Algorithm, LoadListener, StoreListener {
    // algorithm implementation
}
```

There is the definition of `OperationsFacade`s for migration algorithm in the previous example. First facade must be able to perform `LoadAsyncOp` and `LoadOp` operations while the second one is responsible for `StoreAsyncOp`. Besides required and optional operations it is also possible to define facade's names displayed in UI. When the algorithm will be initialized the first element of the array will be 'Source Wiki' facade and 'Destination Wiki' facade will be the second. 'Source Wiki' facade will be consisting of modules capable to perform `LoadAsyncOp` and `LoadOp` operations whereas 'Destination Wiki' will provide modules necessary to implement `StoreAsyncOp` operations.
`Algorithm#getContext` returns context of the algorithm. `AlgorithmContext` is an empty interface because it is impossible to define common context of all algorithms. `AlgorithmContext` implementation can contain values like currently loading and storing pages, list of errors, number of migrated pages etc.
`Algorithm#start` starts the algorithm execution. The `AlgorithmListener` is passed to this method as a parameter.

```
public interface AlgorithmListener {
    void onStateChanged(AlgorithmState state);
}
```

`AlgorithmListener#onStateChanged` is called whenever `AlgorithmState` changes. It is again expected that algorithms can have different algorithm states, however there are four predefined states that every algorithm should achieve.

```
public class AlgorithmState {
    public static final AlgorithmState INITIAL =
                                        new AlgorithmState("Initial");
    public static final AlgorithmState FINISHED =
                                        new AlgorithmState("Finished");
    public static final AlgorithmState TERMINATED =
                                        new AlgorithmState("Terminated");

    protected String name;

    protected AlgorithmState(String name) {
        this.name = name;
```

```
    }

    public String toString() {
        return name;
    }
}
```

Algorithm is in `INITIAL` state since its initialization until it is started with `Algorithm#start`. After algorithm finishes its execution it is in `FINISHED` state. And when algorithm is terminated (`Algorithm#terminate`) it is in `TERMINATED` state. If the algorithm requires more states then it is assumed that the `AlgorithmState` class will be extended.

Similarly to modules, algorithms should be annotated with `@AlgorithmInfo` annotation. `@AlgorithmInfo` annotation has following elements: `name`, `description` and `viewId`. `name` and `description` elements are used by UI. If they are not provided then class name is used as algorithm name and description of algorithm remains empty. The `viewId` element is used by UI logic as an identifier of the view which should be loaded when algorithm is initialized. The reason is that it is expected that with every algorithm implementation also the algorithm UI will be provided. Follows the sample usage of `@AlgorithmInfo` annotation:

```
@AlgorithmInfo(name = "Migration", description = "Algorithm migrates
                all wiki pages from source wiki to destination wiki.
                Pages already present in destination wiki are overwritten.",
                viewId = "migration")
public class Migration implements Algorithm, LoadListener, StoreListener {
        \\ algorithm implementation
}
```

3.6 Managers

Managers are helper classes which can be used for listing of implemented modules and algorithms and getting information about them. They use Java reflection to inspect classes of algorithms and modules, their annotations and configuration beans. There are two managers implemented: `ModuleManager` for getting information about modules and `AlgorithmManager` for getting information about algorithms.

`AlgorithmManager` searches for algorithms in the sub-package of package `net.java.dev.wikisync.algorithms` whereas `ModuleManager` searches for modules in the sub-packages of packages in which are placed interfaces of modules. So for example `Processor` interface is placed in the package `net.java.dev.wikisync.model.processors` and thus Model processor implementations should be placed in the sub-package of `net.java.dev.wikisync.model.processors`.

The information which is provided about each module is based on `@ModuleInfo` annotation, module class and module configuration bean. It means that the information will contain module name, module description, whether is module stateful or stateless, reference to module's `Class` object and description of all configuration properties. The description of configuration properties includes information such as

property name and description, whether is property required or optional and type of the value of the property. On top of that, configuration properties are ordered according to `@ConfigurationProperty`'s `order` element.

The information provided about algorithms is based on `@AlgorithmInfo` annotation, algorithm class and `@Facades` annotation. The information will include algorithm name and description, viewId, reference to algorithm's `Class` object and list of the operations required by each of the facades.

`ModuleManager` also contains mappings from operations interfaces to interfaces of modules and provides helper methods which can be used to determine which modules are required for implementation of particular operation.

3.7 Extending project

In this section I would like to demonstrate what the developer should do to add new module and algorithm implementation.

3.7.1 Implementing new module

When implementing a new module, two new classes should be added. The implementation of the module which implements particular module interface and configuration bean which implements `Configuration` interface. If the module doesn't require any configuration the configuration bean doesn't have to be provided. All the module classes should be placed to the sub-package of the package in which the module interface is placed. In this particular case to the sub-package of package `net.java.dev.wikisync.connectors`. The module implementation class should be annotated with `@ModuleInfo` annotation which provides basic information about the module - module name, module description and a switch stating whether the module is stateless or stateful. Note that module configuration is not module state. Stateful modules cannot be pooled by Operations facade implementation.

I've prepared stateless implementation of the Resource connector module, particularly implementation of the `PageProvider` interface. This fake implementation is very simple. It returns page name, page content and last modification date of the page which are stored in the module configuration bean. The initial values of the configuration bean are set by the user during configuration of the algorithm.

```
@ModuleInfo(name="SampleConnector", description="Sample Resource connector
                implementation which returns and updates page information stored
                in connector's configuration.",
                stateless=true)
public class SampleConnector implements PageProvider {

    private SampleConnectorConfiguration config;

    public void init(Configuration conf) {
        // save reference to module configuration
        config = (SampleConnectorConfiguration) conf;
    }
```

```
    public void dispose() {
        // nothing to dispose
    }

    public void listPages(ResultsHandler handler, ResultsFilter filter) {
        // returns the only page name which is available
        // in the module configuration bean
        String pageName = config.getPageName();

        if (filter != null) {
            if (!filter.filter(pageName)) {
                // page name was not filtered out, let's handle it
                // if handle returns false no more pages should be handled
                // it will happen anyway in this case but
                // it is a good practise to do it anyway
                if (!handler.handle(pageName)) return;
            }
        } else {
            // no filter was passed, let's handle the page name
            if (!handler.handle(pageName)) return;
        }
    }

    public String loadPageContent(String pageName) {
        // return page content of the page
        if (pageName.equals(config.getPageName())) {
            return config.getPageContent();
        } else {
            return null;
        }
    }

    public PageInfo loadPageInfo(String pageName) {
        // return meta-information of the page
        if (pageName.equals(config.getPageName())) {
            return new PageInfo(config.getPageName(),
                                config.getLastModification());
        } else {
            return null;
        }
    }

    public void storePageContent(String pageName, String content) {
        // update page content
        if (pageName.equals(config.getPageName())) {
            config.setPageContent(content);
            // update last modification date of the page to "now"
            config.setLastModification(new Date());
        }
    }
}
```

After the module instantiation the `SampleConnector#init` method is called to ini-

tialize the module. This fake implementation of the module only saves the reference of the configuration bean but typical implementation of the module should initialize module's resources and in case of the Resource connector module it should also test that the connection to the resource is available. If some of the properties is not properly configured or the connection to the resource cannot be established, the `init` should throw an instance of `RuntimeException`.

When the module is used some of the `SampleConnector#listPages`, `SampleConnector#loadPageContent`, `SampleConnector#loadPageInfo` or `SampleConnector#storePageContent` methods is called.
The `listPages` method implementation handles (passes the name to the `ResultsHandler#handle` method) the page name which is stored in the module configuration bean if it is not filtered by passed `ResultsFilter`.
The `loadPageContent` method implementation returns page content of the page which is stored in the configuration bean.
The `loadPageInfo` method implementation returns `PageInfo` structure containing meta-information of the page, again, stored in module configuration bean.
The `storePageContent` method implementation updates page content which is stored in configuration bean and updates last modification date of the page.

When the module is being disposed, the `SampleConnector#dispose` method is invoked. In case of this fake implementation nothing happens but usually the module should dispose all the resources which were initialized in the `init` method.

Let's take a look now on configuration bean of this module.

```
public class SampleConnectorConfiguration implements Configuration {

    private String pageName;
    private String pageContent;
    private Date lastModification;

    public String getPageName() {
        return pageName;
    }

    @ConfigurationProperty(required=true, order = 1, name= "Page name")
    public void setPageName(String pageName) {
        if (pageName == null) {
            throw new NullPointerException("Page name cannot be null");
        }
        this.pageName = pageName;
    }

    public String getPageContent() {
        return pageContent;
    }

    @ConfigurationProperty(required=true, order = 2, name= "Page content")
    public void setPageContent(String pageContent) {
        if (pageContent == null) {
            throw new NullPointerException("Page content cannot be null");
        }
```

```
        this.pageContent = pageContent;
    }

    public Date getLastModification() {
        return lastModification;
    }

    @ConfigurationProperty(required=true, order = 3,
                           name= "Last modification date of the page")
    public void setLastModification(Date lastModification) {
        if (lastModification == null) {
            throw new
            NullPointerException("Page last modification date cannot be null");
        }
        this.lastModification = lastModification;
    }
}
```

It contains three configuration properties - `pageName`, `pageContent` and `lastModification`. All these properties are required as stated by `@ConfigurationProperty` annotations which are annotating bean setters. The `@ConfigurationProperty` annotations also provide names of configuration properties which will be displayed in the UI and also determine the order in which the properties should be displayed in the UI.

3.7.2 Implementing new algorithm

Basically, three classes should be added when implementing new algorithm. The class which implements `Algorithm` interface, the class which implements `AlgorithmContext` interface and the class which extends `AlgorithmState` class. All the algorithm classes should be placed to the sub-package of package `net.java.dev.wikisync.algorithms`. The class which implements `Algorithm` interface should be annotated with `@AlgorithmInfo` annotation which provides basic information about the algorithm and an identifier of initial algorithm view. The class should be also annotated with `@Facades` annotation which says how many facades are required by the algorithm and which operations must be available by the facades.

I've prepared a simple implementation of migration algorithm which uses only synchronous operation calls. The source code of all implemented classes is available below.

The first method which is called after instantiation of the algorithm is `Algorithm#init`. Instances of facades are passed to `init` method (in the order in which were requested in `@Facades` annotation) and algorithm changes its state to `INITIAL`. After the `init` call the algorithm UI should load the initial algorithm view whose identifier is stored in `viewId` element of the `@AlgorithmInfo` annotation. Note that algorithm UI must be provided by algorithm developer.

```
@AlgorithmInfo(name = "Migration", description = "Sample example of
                migration algorithm.", viewId = "migration")
@Facades({
        @OperationSet(name = "Source Wiki", required = { LoadOp.class},
                optional = {}),
```

```
        @OperationSet(name = "Destination Wiki", required = { StoreOp.class },
                optional = {})
})
public class Migration implements Algorithm {

    private OperationsFacade sourceWiki;
    private OperationsFacade destinationWiki;
    private MigrationContext context;
    private AlgorithmState state;
    private AlgorithmListener listener;
    private boolean terminate;

    public void init(OperationsFacade[] facades, String profile) {
        // initialize facades
        sourceWiki = facades[0];
        destinationWiki = facades[1];

        // algorithm state is INITIAL
        state = MigrationState.INITIAL;
    }

    public AlgorithmContext getContext() {
        return context;
    }

    public AlgorithmState getState() {
        return state;
    }

    public void start(AlgorithmListener listener) {
        this.listener = listener;

        // algorithm was started, change state to MIGRATING
        state = MigrationState.MIGRATING;
        listener.onStateChanged(state);

        // the main algorithm loop
        // list source wiki pages, load them and store to destination wiki
        sourceWiki.listPagesNames(new ResultsHandler() {

            public boolean handle(Object obj) {
                String pageName = (String)obj;
                // load page from source wiki
                WikiPage page = sourceWiki.loadWikiPage(LoadType.ALL, pageName);
                // store to destination wiki
                destinationWiki.storeWikiPage(page);
                context.migratedPages++;

                // check that algorithm was not terminated
                synchronized (this) {
                    if (terminate) return false;
                }

                return true;
```

```
            }

        }, null);

        // all pages are migrated, change state to FINISHED
        state = MigrationState.FINISHED;
        listener.onStateChanged(state);
    }

    public void terminate() {
        // terminate all requests - this is not necessary to call because
        // this implementation uses only synchronous calls but it is
        // a good practise
        sourceWiki.terminateAll();
        destinationWiki.terminateAll();
        synchronized (this) {
            terminate = true;
        }

        // change state to TERMINATED
        state = MigrationState.TERMINATED;
        listener.onStateChanged(state);
    }

}
```

When user starts algorithm (clicks on start button in the algorithm UI for instance), the `Algorithm#start` method should be called. Algorithm changes its state to `MIGRATING`, notifies `AlgorithmListener` about state change and algorithm begins its execution. The `AlgorithmState` class contains only three basic algorithm states - `INITIAL`, `FINISHED` and `TERMINATED`. In this example one extra state - `MIGRATING` - is required and thus the `AlgorithmState` class must be extended.

```
public class MigrationState extends AlgorithmState {
    public static final MigrationState MIGRATING =
                                        new MigrationState("Migrating");

    protected MigrationState(String description) {
        super(description);
    }
}
```

This simple algorithm implementation has the main algorithm loop in `ResultsHandler` anonymous implementation in `Algorithm#start` method. Algorithm lists all pages which are available in the source wiki, loads their content and stores them to destination wiki. Everytime a page is migrated from source wiki to destination wiki, the `MigrationContext#migratedPages` property value is increased. The `MigrationContext` class is supposed to be shared with the algorithm's UI. The UI can use properties stored in the context to display their values in the algorithm UI.

```
public class MigrationContext implements AlgorithmContext {
    public int migratedPages = 0;
}
```

After algorithm migrates all wiki pages from source wiki to destination wiki, algorithm changes its state to `FINISHED` and notifies `AlgorthimListener` that the state has changed. Algorithm execution has just finished.

Anytime the user decides to terminate the execution of the algorithm (clicks on terminate button in the algorithm UI for instance), the `Algorithm#terminate` method is called and algorithm execution is terminated.

Chapter 4

Unified Object Model

This chapter will focus on object model which forms core part of this project. The object model is a unified object representation of wiki content. If two pages have the same content then the object representation of both pages must be the same despite the fact that page source is different.
First of all, let's start with a few requirements which must be fulfilled. The object model must:

- be unambiguous
  The object model must be unambiguous so that two pages can be compared and decided whether are the same or where they differ. If the model would be ambiguous the comparison algorithm would be very complex.

- be extensible
  It is absolutely natural that things evolve and wikis are not different. That's why the model must be easily extensible so that newly supported elements of wiki engines can be added to the object model.

- be intuitive
  It is the simplicity which attracts people to use and extend things. If the model is simple and intuitive then all the modules and algorithms will be simple too and as I mentioned many times this is the goal. Making the model intuitive means that it will be much easier to understand it and remember it - which is definitely necessary to be able to implement modules and algorithms.

- be easily processable
  Intuitive model doesn't automatically mean that it is also easily processable. I mean that also working with the model must not be complex. It should provide a way how to move across the model elements and process them adequately.

- cover all common elements
  Although the model must be extensible it should also already cover all the elements which are usually supported by wiki engines including nesting of elements (list inserted into table cell etc).

- have support for unrecognized elements
  Another important goal of this project is to provide adequate level of reporting. In case some wiki elements are not supported by object model, such as wiki plugins, the model must offer a way how to tag these elements so that they can be reported as unknown.

4.1 Discussion of solutions

There exist two main approaches to the model structure - DOM and SAX. In case of DOM the model elements form tree based representation of wiki content. Page for instance is represented by one element which consists of page section elements which consist of elements like headings, paragraphs, lists, tables etc. SAX representation is event based. A representation of the same page would consist of page start and page end elements. Inside these page elements there would be page section start and page section end elements and inside these there would be headings' starts and ends, lists' starts and ends etc.

Both approaches have some advantages and also downsides. Both models are unambiguous and extensible. The implementation of SAX parser and processor is simpler because it just creates and processes elements as they come. On the other hand, DOM model is easier to compare and diff with another model and it's also more intuitive. Another drawback of SAX model is that when the part of the model is lost the part which remained doesn't have to have the complete information. In case of DOM model this is not true because every element of the DOM model contains complete information about itself. And finally, if we want to design a model editor DOM representation would be much more suitable for this. Such a model editor might be useful in off-line wiki editor for instance.

Based on the facts discussed above I have decided to use DOM-like representation of the content.

4.2 Model description

Object model is divided into two main parts:

- structural elements
  To this group belong elements which form structure of the document like page element, page section element, block elements such as tables, lists, headings etc.

- inline elements
  Inline elements are elements which form the content of structural elements such as formatted text, references, special symbols etc.

The reason for this distinction is that it is necessary to represent also the textual content in the unified form, not only the structure. Different wiki engines use different formatting marks, references are also written differently. To be able to compare two

wiki pages stored in different markup or translate a page to different markup it is necessary that also the text is represented uniformly.

4.2.1 WikiModelNode

Every element of the object model extends abstract class `WikiModelNode` (see Figure 4.1). This brings many advantages - it is possible to define common functionality for all model nodes and it is also possible to define common operations on nodes returning or accepting common superclass - `WikiModelNode` - as a parameter.

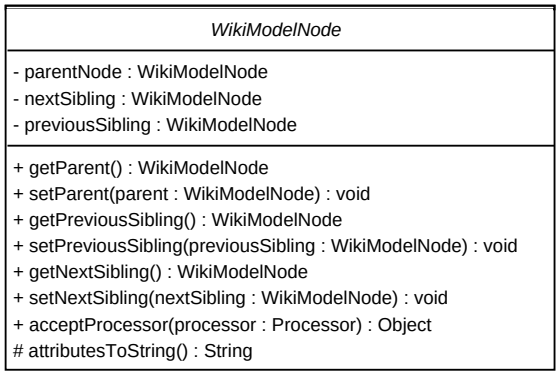


Figure 4.1: WikiModelNode

`WikiModelNode` implements `Cloneable` and `Serializable` interfaces to ease the manipulation with the model. Every `WikiModelNode` contains references to its *parent node*, *previous sibling* (sibling who is before the current node in parent's child list) and *next sibling*. Obviously, if the node has no *parent* or some of the *siblings* is missing, `null` is the expected value. Someone might wonder that there are no references to child nodes in `WikiModelNode`. That's because these references are explicitly defined only for nodes which have some children. Besides the references to neighbours `WikiModelNode` also overrides `equals`, `hashCode` and `clone` methods and defines `acceptProcessor(Processor)` method. It is an acceptor method of Visitor design pattern (Figure 4.2). The visitor in our case is Model `Processor` and acceptors are model nodes. This pattern gives a `Processor` developer option to avoid constructs such as:

```
if (node instanceof WikiTable) {
        process((WikiTable)node);
}
else if (node instanceof WikiParagraph) {
        process((WikiParagraph)node);
}
else if (node instanceof WikiHeading) {
        process((WikiHeading)node);
}
...
...
```

and instead use:

```
node.acceptProcessor(this);
```

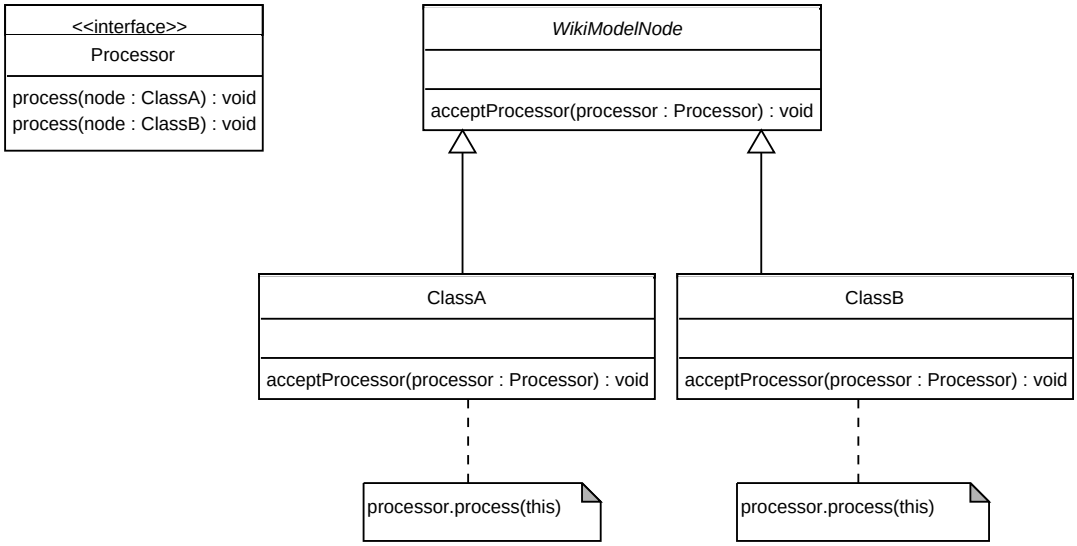


Figure 4.2: Visitor design pattern

However, it is up to the taste of the developer which option prefers.
The last method defined by `WikiModelNode` is `attributesToString` which is only a helper method of `toString` method. It should transform all node's attributes to `String`.

4.2.2 Structural elements

As mentioned earlier structural elements are those elements which form the structure of the content. The root structural element is `WikiModel` (see Figure 4.3). `WikiModel` typically is not present in wiki content. However, it is nice to have the top most element, the root. `WikiModel` can have assigned a *name* and a *description* and mainly `WikiDocument`s belonging to the particular `WikiModel`. `WikiDocument` is an abstract class and right now there are two subtypes of `WikiDocument` - `WikiAttachment` and `WikiPage`. The purpose of both subtypes is obvious. `WikiDocument` must have assigned a *name* and a *version*, optionally an *author of the document*, *date of last modification*, *comment of last modification* and *size* (in bytes) of document. Besides that `WikiAttachment` has mandatory attribute *data* which contains the actual bytes of the attachment.
In most wiki engines attachments and pages are related - attachments are assigned to particular wiki pages. However, there exist some exceptions, PmWiki for instance, has attachments that are not assigned to particular wiki pages but rather are assigned to the whole wiki. Object model supports both cases.

Pages consist of page sections. In most engines new page section starts with level 1 heading. If the wiki engine doesn't support page sections then it is expected that

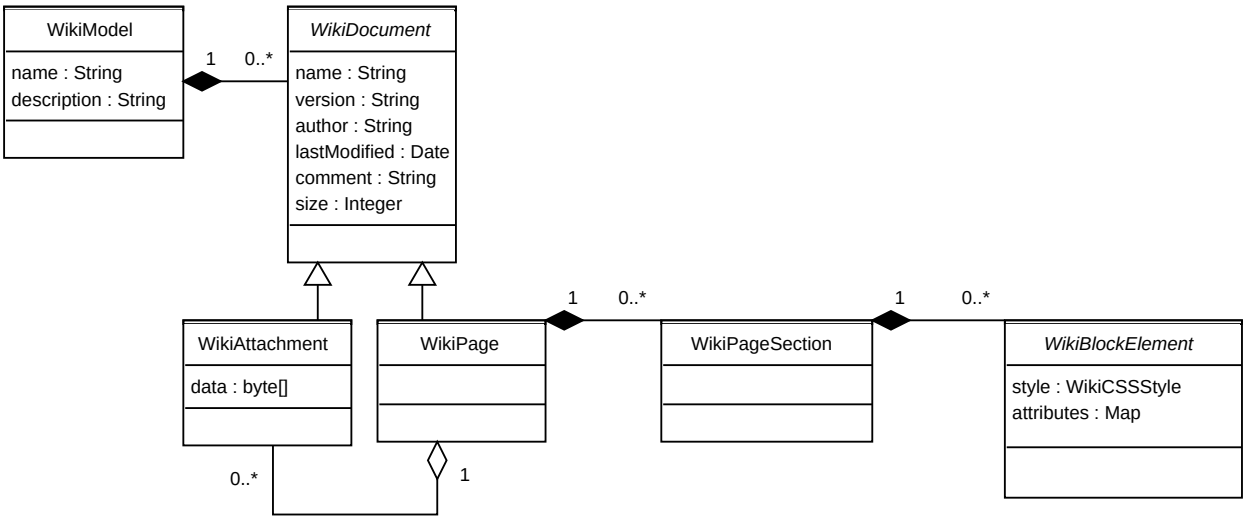


Figure 4.3: Structural elements, part 1

the page will contain only one page section.
Page sections are composed of `WikiBlockElement`s. All the block elements are shown on Figure 4.4.

`WikiBlockElement` is an abstract class. Every block element can have defined *style* and *attributes*. `WikiCSSStyle` currently contains only a `Map<String, String>` of attributes' names (eg: background-color) mapped to attributes' values (e.g: red). Originally, I thought that it will also contain CSS style class and CSS style id but these values are engine-dependent which consequently mean that they must be translated to correct *attributes*. `WikiBlockElement#attributes` is another `Map<String, String>` which is expected to contain attributes which are not CSS styles (alignment, cellspacing etc) plus some predefined attributes. All currently predefined attributes are listed in the `WikiBlockElement` javadoc, indentationLevel is a good example of predefined attribute.

`WikiTerm` consists of *term* and its *definitions*. *Term* is of type `WikiContent` which means that the *term* can contain both formatted and non-formatted text, references etc. *Definitions* are `WikiBlockElement`s, usually `WikiTextElement`s.
`WikiList` has two extra attributes - *type* and *list items*. List type can be `ORDERED` (list items starting with numbers) or `UNORDERED` (list items starting with bullets). List items are again `WikiBlockElement`s. To avoid ambiguity following list:

```
* item1
** item1.1
** item1.2
* item2
```

is expected to be represented as a `WikiList` containing three *items* - `WikiTextElement` (item1), another `WikiList` (containing two `WikiTextElement`s) and `WikiTextElement` (item2) as in case of the representation in HTML:

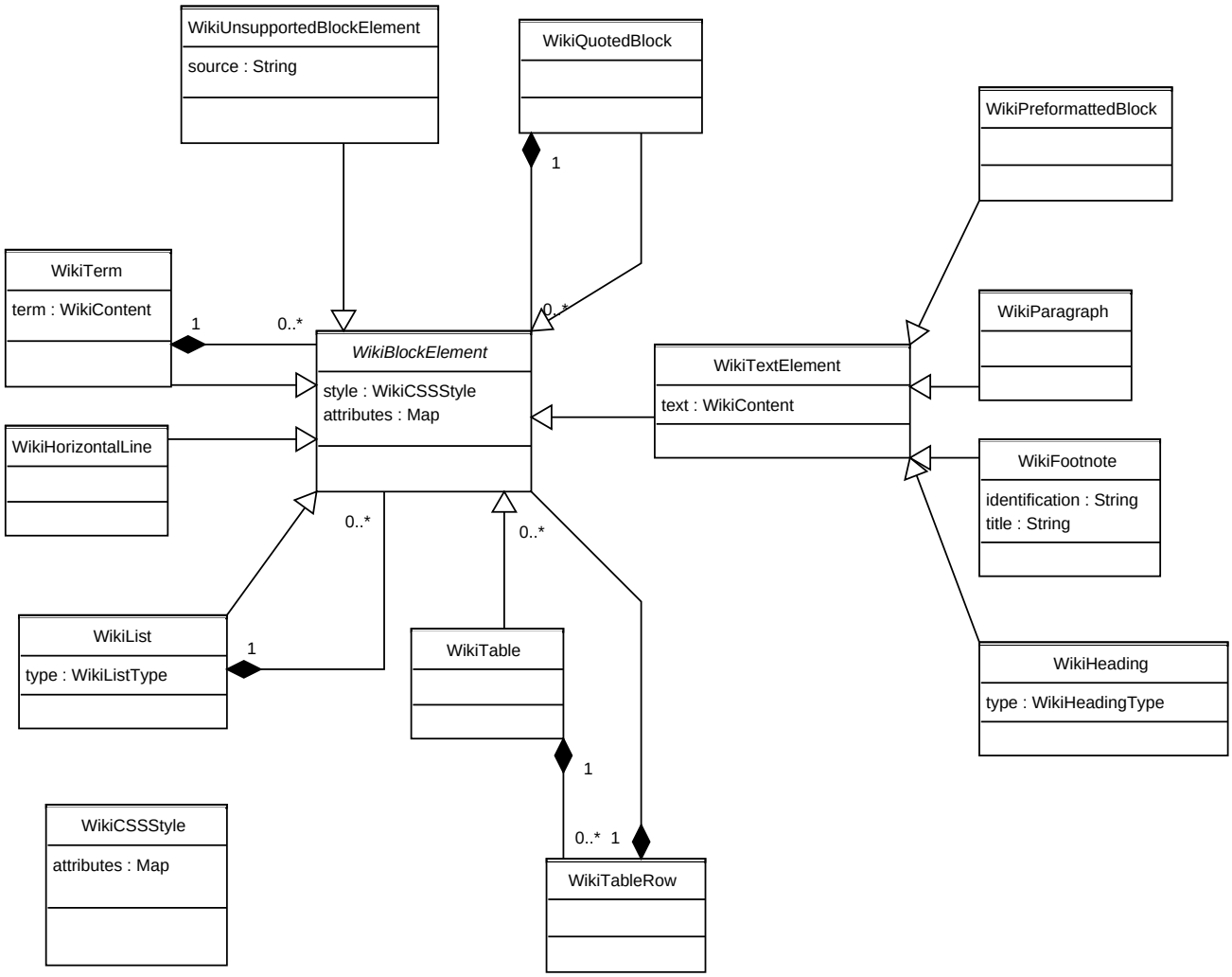


Figure 4.4: Structural elements, part 2

```
<ul>
  <li>item1</li>
  <li>
    <ul>
      <li>item1.1</li>
      <li>item1.2</li>
    </ul>
  </li>
  <li>item2</li>
</ul>
```

`WikiTable` is composed of `WikiTableRow`s whose *cells* are again `WikiBlockElement`s. `WikiTextElement` has one extra attribute *text* which presents the content of the element. `WikiTextElement` is further divided into `WikiPreformattedBlock`, `WikiParagraph`, `WikiFootnote` and `WikiHeading`. `WikiPreformattedBlock` is block whose spaces

and linebreaks are displayed literally as in case of HTML's `<pre>` tag. `WikiParagraph` is simply a paragraph. Paragraphs are usually in wiki markups separated with one empty line. `WikiFootnote` consists of *title* and *identification*. *Title* is the footnote text which is being referenced by *identifier*. `WikiHeading` has one extra attribute - *heading type*. Possible *heading type* values are `LEVEL1` - `LEVEL5`. `LEVEL1` is expected to be the most important heading as `<h1>` in case of HTML. `WikiQuotedBlock` contains another `WikiBlockElements` that should be quoted.

Finally `WikiUnsupportedBlockElement` is the element which should be used in case that object model doesn't have a block element for some extraordinary one supported by wiki engine. In this case the content including starting and ending block element mark should be stored in *source* attribute.

4.2.3 Inline elements

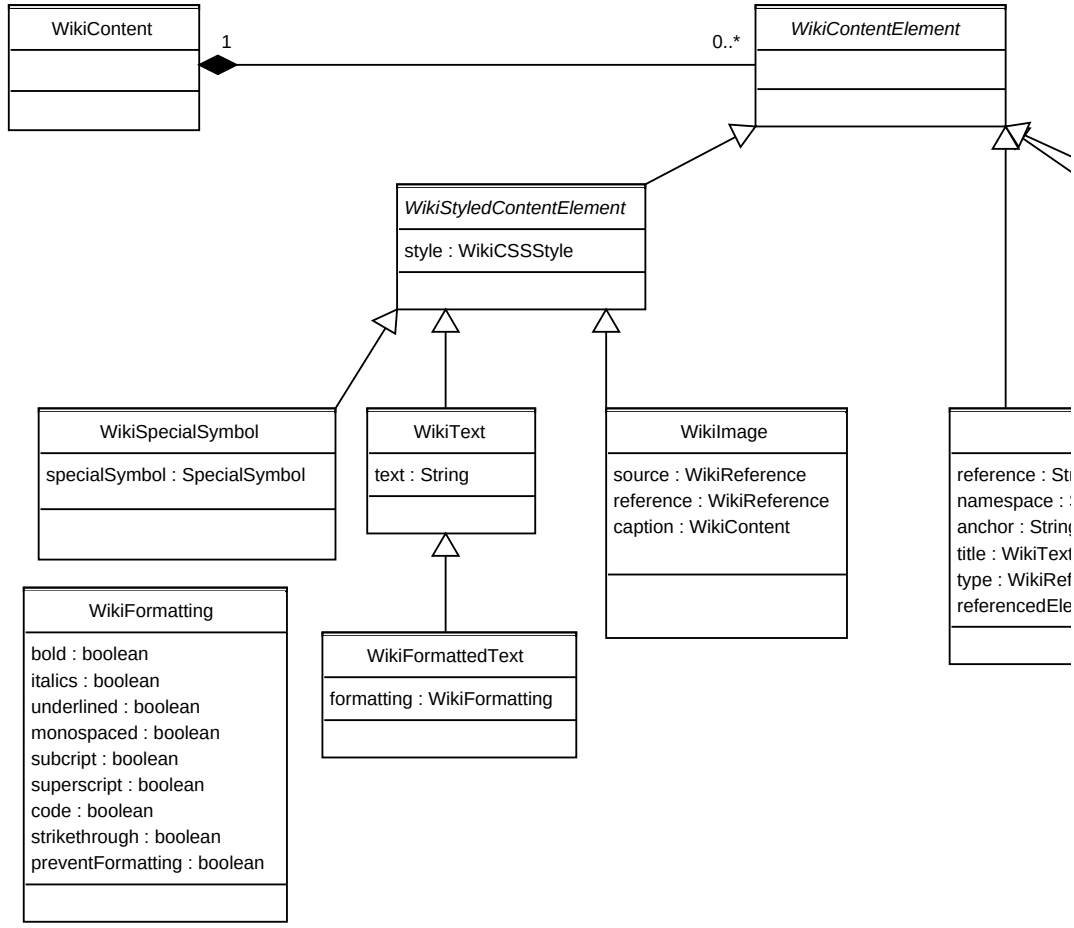


Figure 4.5: Inline elements, part 1

`WikiContent` is the root inline element of the object tree which represents the content of some block element. `WikiContent` consists of *content elements* which are descdendants of abstract class `WikiContentElement`. `WikiContentElement` is

divided into *styled content element* and the rest. `WikiStyledContentElement` has one common attribute - *style*. `WikiCSSStyle` class was already described earlier in this chapter. *Styled elements* are *special symbols*, styled *text* and *images*. *Special symbol* is linebreak for example. In most wiki engines linebreak is written with \\, however some engines also use `<br>` or any other marker. Text can be represented by `WikiText` or `WikiFormattedText` classes. Text is stored in plain form in *text* attribute. In case text is formatted it is stored as `WikiFormattedText` which in addition contains *formatting* attribute. `WikiFormatting` is an enumeration of flags such as *bold*, *italics*, *monospaced* etc.

`WikiImage` has several attributes. Note that none of them are the bytes which actually form the image itself - the binary representation. `WikiImage` is just the captioned reference to an image which is stored either as an attachment or is placed somewhere on the internet. It might seem that image is rather block element than inline element, but after deeper studying of wiki engines and their markups I've found out that it is inline element. Images that look like a block elements are actually paragraphs or other text elements which contain only one element - the image. `WikiImage` has following attributes: *source*, *reference* and *caption*. *Source* is a

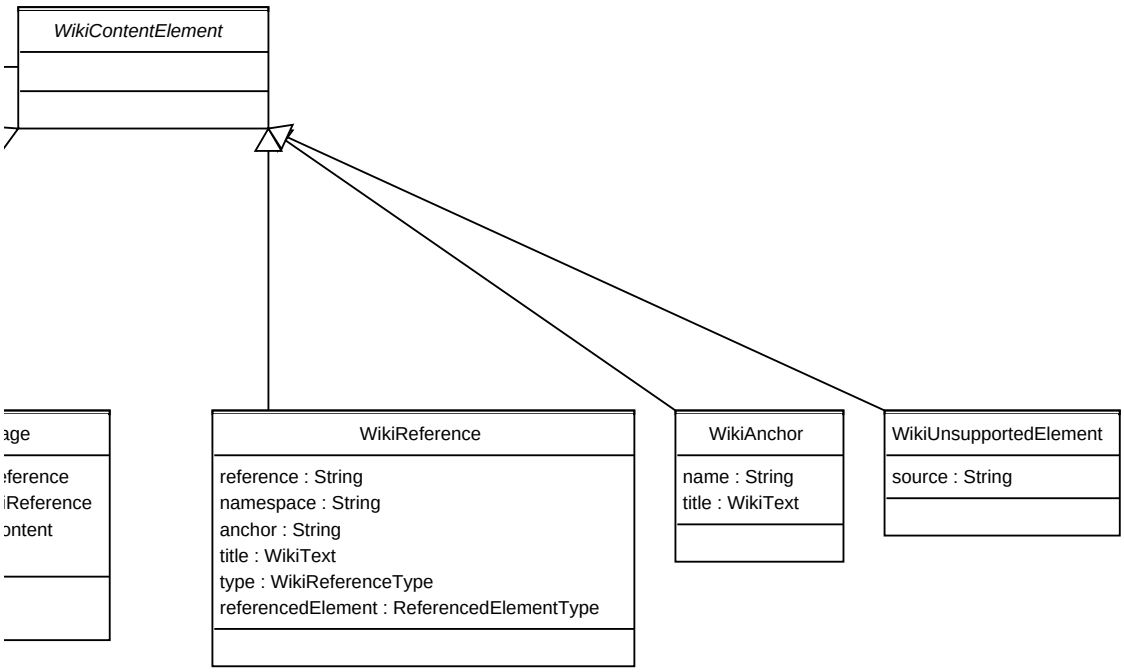


Figure 4.6: Inline elements, part 2

`WikiReference` pointing to the stored image - attachment or URL. *Reference* is again a `WikiReference` in this case the reference which is used after clicking on the image. And finally *caption* is another `WikiContent` element which forms the caption describing the image. Note that in case that image doesn't have any of the mentioned attributes `null` is the expected value.

First of non-styled content elements I'll describe is `WikiReference`. Someone might wonder that reference is non-styled element. The reason is that it is not the reference itself which have assigned a style but the *title* which is displayed and *title* is of type `WikiText` which may have *style*. Most important `WikiReference` attribute is *reference*. *Reference* can be name of another wiki page within the wiki - such a reference is called internal. Or reference can point to a page on the internet - external reference. Actually there is also option three which is called interwiki reference. Interwiki references point to another page which is hosted by different wiki engine. Nevertheless, object model currently doesn't support this kind of references because the referenced wiki engines must be somehow configured by the end user and another reason is that they are not used much. Type of the reference is stored in attribute *type*. Some wiki engines support *namespaces* for references. The reason is that in single namespace page name must be unique. For conversion of reference from wiki engine which supports namespaces to wiki engine which does not support namespaces one possible solution is to prefix page name with *namespace* name. Obviously this is only possible if all the references are converted this way.
Anchor is a special tag within a page which can be referenced. When the user clicks on reference which has specified anchor browser loads the page and jumps to the anchor.
`WikiAnchor` have two attributes - *name* under which it can be referenced and *title* which can be displayed in the place where anchor is placed.

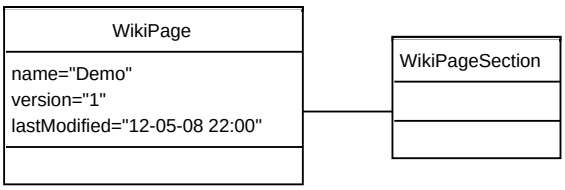
The last content element is `WikiUnsupportedElement` which should be used in case that wiki engine supports some special element which is not or cannot be supported in the object model. Typical example is a plugin element which is obviously engine-dependent and thus cannot be represented uniformly.

Generated detailed documentation of the whole object model is accessible on the thesis CD in the `javadoc` directory.

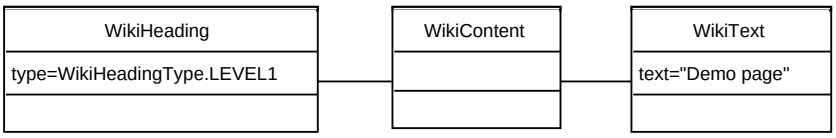
4.3 Sample representation

In this section I would like to show how particular wiki page is represented in unified object model. Figure 4.7 shows a screenshot of a demo page stored in JSPWiki. The page contains various block elements - headings, horizontal line, list, table, text elements and term with definition.
Object representation of the page forms tree structure of objects. Root node of the tree is `WikiPage` element which contains some meta-information about the page and references to page sections. This particular example contains only one page section.

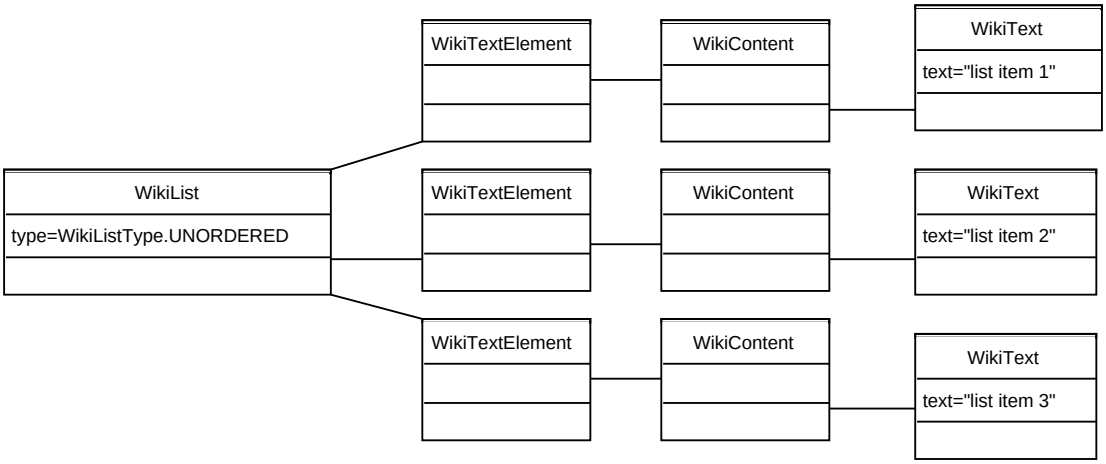
Figure 4.7: Demo page in JSPWiki



Page section consists of block elements which forms it. Following pictures describe block elements one by one in the order in which they're stored within page section. First block element is 'Demo page' level 1 heading.

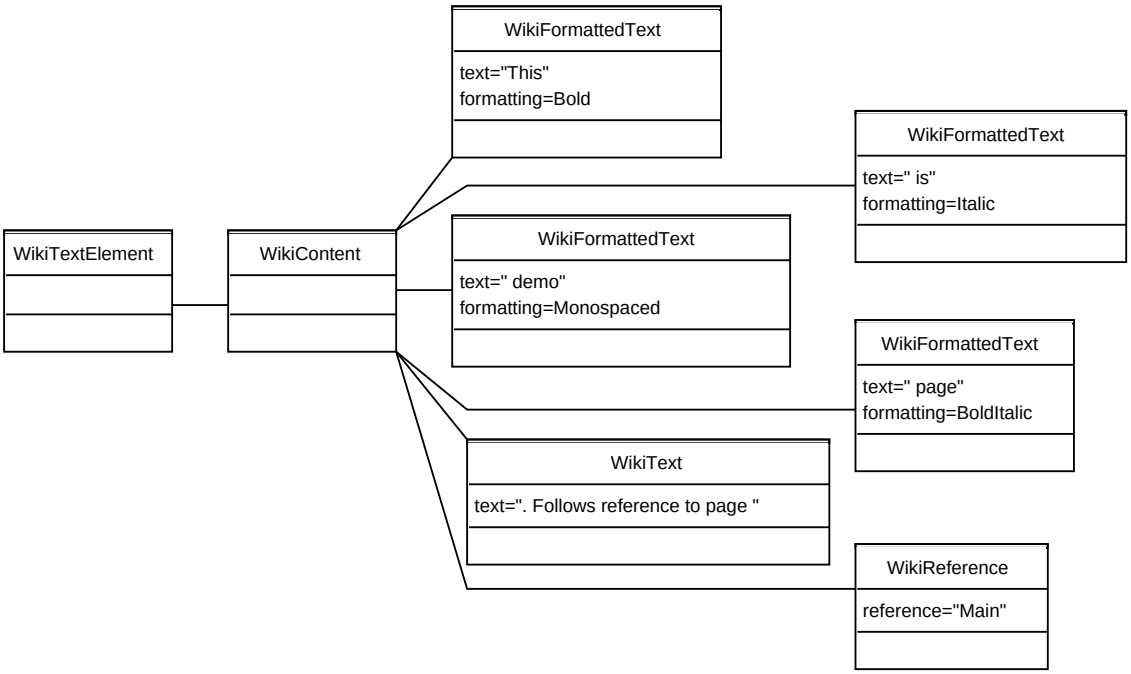


After the heading follows unordered list with three list items - 'list item 1', 'list item 2' and 'list item 3'.

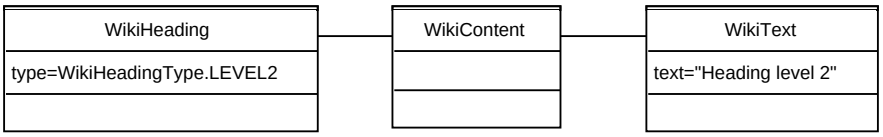


List is followed by `WikiHorizontalLine` which has no extra attributes. Horizontal

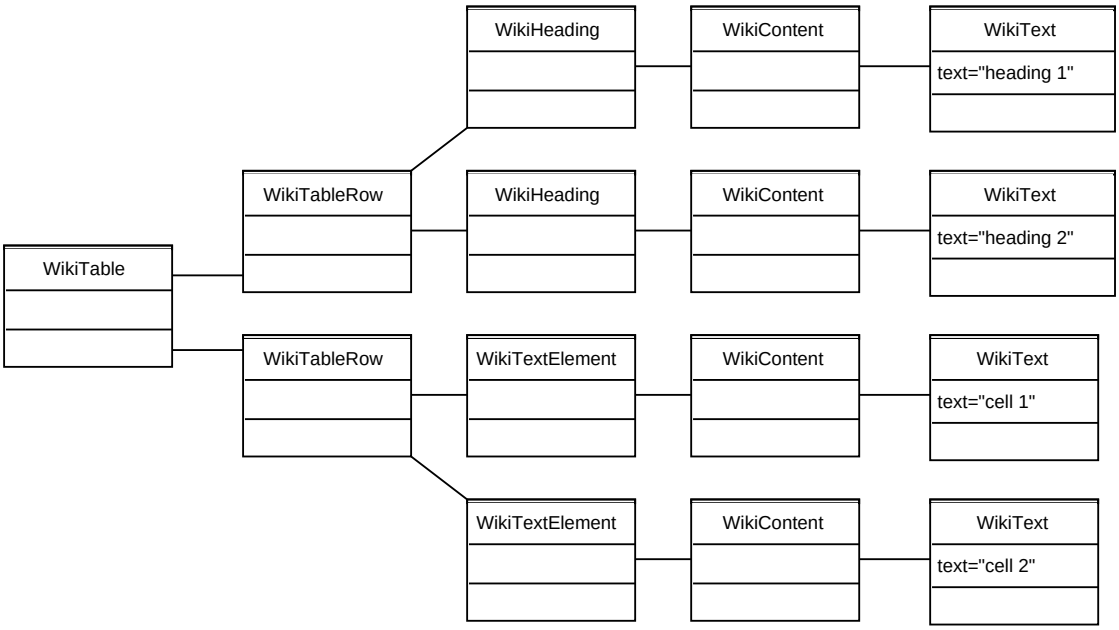
line is followed by text which is formatted using different formatting. **This** is in bold, *is* is in italics, `demo` is monospaced and ***page*** is both bold and italics. Formatting is represented as references to formatting classes (Italic, Bold, BoldItalic, Monospaced) in the picture. Regular text and a reference to 'Main' page follows after the formatted texts.



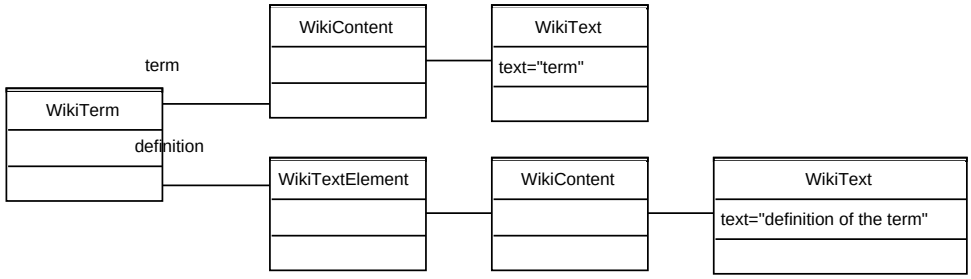
Then follows level 2 heading with text 'Heading level 2'.



WikiTable is the next block element. It is formed by two rows. First row contains two cells - both headings. Second row also contains two cells which are regular in this case.



The last block element within the page section is the term 'term' and its definition



'definition of the term'.
This was the complete object representation of the demo page. Figure 4.8 shows 'Demo' page after its migration from JSPWiki to MediaWiki.

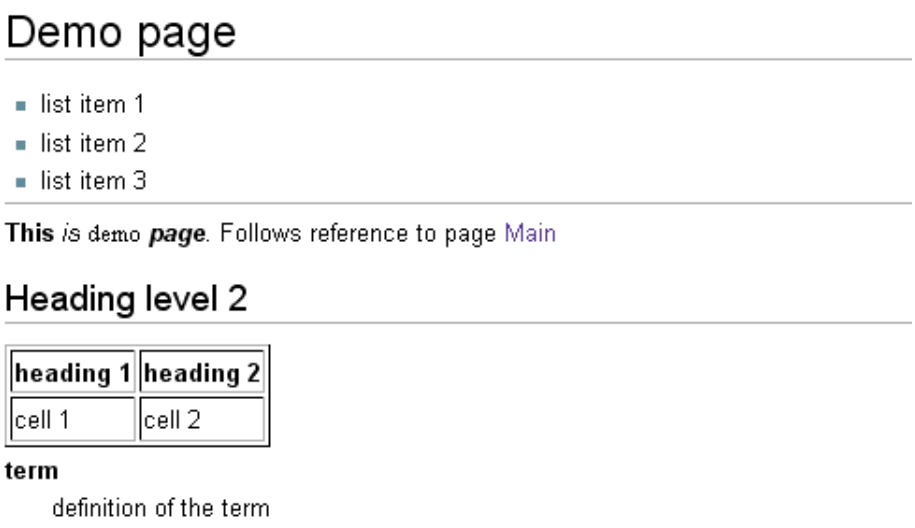


Figure 4.8: Demo page in MediaWiki

Chapter 5

Future of the project

This chapter focuses on future extensions and improvements of the project. Most of these improvements and extensions cannot be made right now because they require deep analysis of the problem which would be beyond the scope of this thesis.

5.1 Add new algorithms and modules

First of all, new modules and new algorithms should be implemented. My idea would be to be able to migrate and synchronize content among most of the popular wiki engines. This requires a lot of new modules to be implemented. Resource connectors are required to be able to gather content from wiki engines, Model builders and Model processors are required to translate the content to different wiki markups. Although some modules can be reused by more wiki engines - the same RPC interfaces and support of Creole markup - there are plenty of other wiki engines which don't support them. This is the reason why some generic modules should be implemented. Actually, there already is one generic Resource connector module implemented. It is called DBConnector and it is possible to use this Resource connector module to gather content from wiki engines which use databases as their storage. Configuration of this module is more complicated than configuration of other modules because it requires a few SQL statements to be defined, JDBC connection URL and credentials to the database to be provided but the configuration is done only once and then it is stored under the profile. Similarly, it might be implemented Model builders and Model processors based on grammars or at least rules. Such modules might be used to parse and process all wiki markups. The configuration would be again more complex but it is done once. Example of a set of rules which might be used by generic Model builder and Model processor follows.

```
<HEADING1> = ! <HEADING_TITLE>
<HEADING2> = !! <HEADING_TITLE>
<HEADING3> = !!! <HEADING_TITLE>
<ORDERED_LIST> = # <LIST_ITEM>
<UNORDERED_LIST> = * <LIST_ITEM>
<HORIZONTAL_LINE> = ----
```

```
<TERM> = ;<TERM_TITLE>:<DEFINITION>
<BOLD_TEXT> = __<TEXT>__
<ITALIC_TEXT> = ''<TEXT>''
```

Obviously, this would need much deeper analysis how the rules should look like. This is just an example.

There is also many algorithms which should be implemented, some of them were mentioned in the second chapter. First of all, two-way synchronization algorithm should be implemented. This algorithm requires solving of many issues such as diffing of object models, providing smart UI which shows only the differences between two pages and provides an easy way to merge contents. It also requires more modules to be implemented because both wiki engines are read and written.
Furthermore, it would be nice to have algorithms which can produce reports based on wiki content containing information about the wiki - how many pages does it contain, who are the most active authors, which wiki pages are just drafts, which pages are the newest etc. Algorithms which are able to identify SPAM or other kinds of vandalism and remove it, algorithms which can be used for mass changes such as renaming of wiki pages (including updates of references to the pages), adding tables of contents to wiki pages and other replacements.

5.2 Object model diff

Current version of object model offers only method which tests two models for exact equality. This is definitely useful but some more comparison and diff methods would be needed. Let's suppose that page 'Foo' contains text 'Bar' in the first wiki engine, the same page on another wiki contains text 'Bar ' (the same content just with one more white space). These two pages are not exactly equal but they are at least very similar. The process of markup translation is not trivial - firstly the source content must be parsed, then object representation of it created and finally the object representation is serialized into different markup. Such tiny differences between pages, especially regarding white spaces, can occur very easily. This is the reason why should be the object model equipped with methods to test also similarity. The method signature might look like:

```
boolean similar(int deviation)
```

The `deviation` parameter would mean how much can be two pages different. The metrics of it would need further analysis and also the algorithm which will test it. The `deviation` could be the number of symbols which can be different in the whole object representation or it could be percentage or rate of the portion of the content which differs or anything else.

Another method or set of methods which would be in the future required are the diff methods. These methods are necessary requirement of two-way synchronization algorithm where conflicting pages may occur. When the conflict occurs the GUI should show the parts of both pages which are different so that the user can resolve

the conflict. Again a very deep analysis of the whole problem must be done. The issues which need to be solved involve the algorithm which will be used for diff, what are the return values of diff methods, would there be also necessary some `deviation` parameter etc. The algorithm might serialize the object model into some textual representation (e.g. XML), then perform textual diff and eventually build object representation of differences. However, the algorithm which would somehow directly compare object representations would be preferred. The return values of diff methods could be list of references to objects which differ or it could be reference to the node in the object tree which is the root of the sub-tree which is different etc.

5.3 Provide better UI

Since the UI has never been the goal of this thesis it should be improved. Especially the support for profiles should be extended. Right now the whole configuration of the algorithm (including modules' configurations) is saved and it can be reused next time. However, it would be nice and I would say almost necessary to provide also more granular profiles of configurations per modules. These profiles would be very useful when generic modules are implemented. Users could save profiles of configurations per modules and compose algorithm configurations from them. This way users could define a DBConnector configuration for MediaWiki, JSPWiki and other engines. Similarly, they could define profile configurations for generic Model builders and processors. On top of that, these profile configurations of modules could be distributed with the application and users would no longer need to create them. They will just compose their algorithm configurations and save them.

It would be also nice to provide more UIs than just the web UI. Web UI is definitely one of the most popular ones nowadays but it is not suitable for tools such as off-line editor of wiki content. Off-line editor should be rather a desktop application. It might seem that such an application would be so much different than the one which is presented in this thesis but opposite is true. Most of the architecture would remain the same. There would be one Resource connector module which would provide access to remote instance of wiki and one Resource connector which would load and store content to local files. Model builders and processors would be the same. It would just require two-way synchronization algorithm implementation to synchronize content between local and remote wiki instance and some WISIWYG editor which would allow editing of wiki content and update object model according to changes which the user makes.
Command-line interface would be also useful. User can prepare algorithm configuration profile via graphical UI and then use this profile and run algorithm from command-line interface.

5.4 Build project community

This is not usual improvement of the project but it is tremendously important. To achieve a real success of this project people must use it and extend it. Building a

project community is not easy. The project must be attractive for people so that they start using it and contributing to it. The design of the project must be simple and intuitive on one hand and on the other hand it must be also flexible enough to survive the growth of the project. All these assumptions should be satisfied by this project but the future will show if the assumptions were correct or not.

Chapter 6

User guide

This chapter describes what is required for installation of the application, how to install it and how to use it. I would like also mention that the User interface provided with the thesis never was a goal of this thesis; however I decided to offer at least a simple one to be able to use the application.

6.1 Installation

Wiki Synchronization Tool is standard web application and it is distributed in web archive file named `wikisync.war` which is placed on thesis CD. This file should be deployed to an application server. There exist many application servers and it is up to the user which chooses, the application should work with all of them. I'll describe how to deploy it to GlassFish v2UR2 application server. Installation files of GlassFish v2UR2 are also placed on the thesis CD within `glassfish` directory. Before start of installation process of GlassFish please make sure that JDK 5.0 (or newer) from Sun Microsystems is installed on the computer installation and that `JAVA_HOME` environment variable points to the installation directory of it. Note that JRE is not enough for GlassFish installation. The current version of Java installed on the computer can be obtained by running `java -version` command. Installation files of JDK from Sun Microsystems can be downloaded from `http://java.sun.com/`.

1. Copy installation file of GlassFish v2UR2 from glassfish directory on the thesis CD to the location where the server should be installed. Choose correct installation file for your operating system.

2. Run `java -Xmx256m -jar filename.jar`
   `filename.jar` is the name of installation jar you've copied. This command unbundle GlassFish and create a new directory structure rooted under a directory named `glassfish`.

3. `cd glassfish`

4. If you are using a machine with UNIX operating system or its derivate, set the execute permission for the Ant binaries that are included within the GlassFish bundle and run ant setup script:

```
chmod -R +x lib/ant/bin
lib/ant/bin/ant -f setup.xml
```

If you are running Windows, just run ant setup script:

```
lib\ant\bin\ant -f setup.xml
```

5. Copy `wikisync.war` from the thesis CD to `glassfish/domains/domain1/autodeploy` directory.
6. Go to `glassfish/bin` and start application server with: `./asadmin start-domain` (or for Windows: `asadmin.bat start-domain`)
7. After a while application is deployed to the server. New file named `wikisync.war_deployed` will appear in `glassfish/domains/domain1/autodeploy` directory.
8. Open your web browser on `http://localhost:8080/wikisync/` and you'll see main page of Wiki Synchronization Tool.

Undeployment of the application is very simple, just delete `wikisync.war` from `glassfish/domains/domain1/autodeploy` and wait a while. To stop the application server run `./asadmin stop-domain` (or for Windows: `asadmin.bat stop-domain`) from `glassfish/bin` directory. To remove the application server from your system completely stop it and then delete installation directory of it.

Installation process was tested on Windows XP SP2 and Ubuntu 7.10.

6.2 Using WikiSync

This section describes how to use UI provided with thesis. The first subsection will describe how to configure algorithm including all the modules required by the algorithm. The second subsection will describe UI of the Synchronization algorithm.

6.2.1 Algorithm configuration

In this section I would like to describe how to configure WikiSync and how to run algorithms. Note that this user documentation assumes that the user has at least basic knowledge of the architecture which is used by WikiSync. The whole WikiSync UI which is provided with the thesis is based on AJAX requests including the main configuration page. It means that the pages are not refreshed as the whole but only parts of them as a reaction to user actions or algorithm updates. E.g. user decides to use different module than the currently selected one, chooses different module and module configuration properties are automatically reloaded via AJAX.

The main WikiSync page is also the main configuration page. This is the page where an algorithm is selected, where necessary modules are chosen and module configuration is set up. It is divided into several sections which can be seen on Figure 6.1. The first section is selection of algorithm (1.) and after it follow configurations of

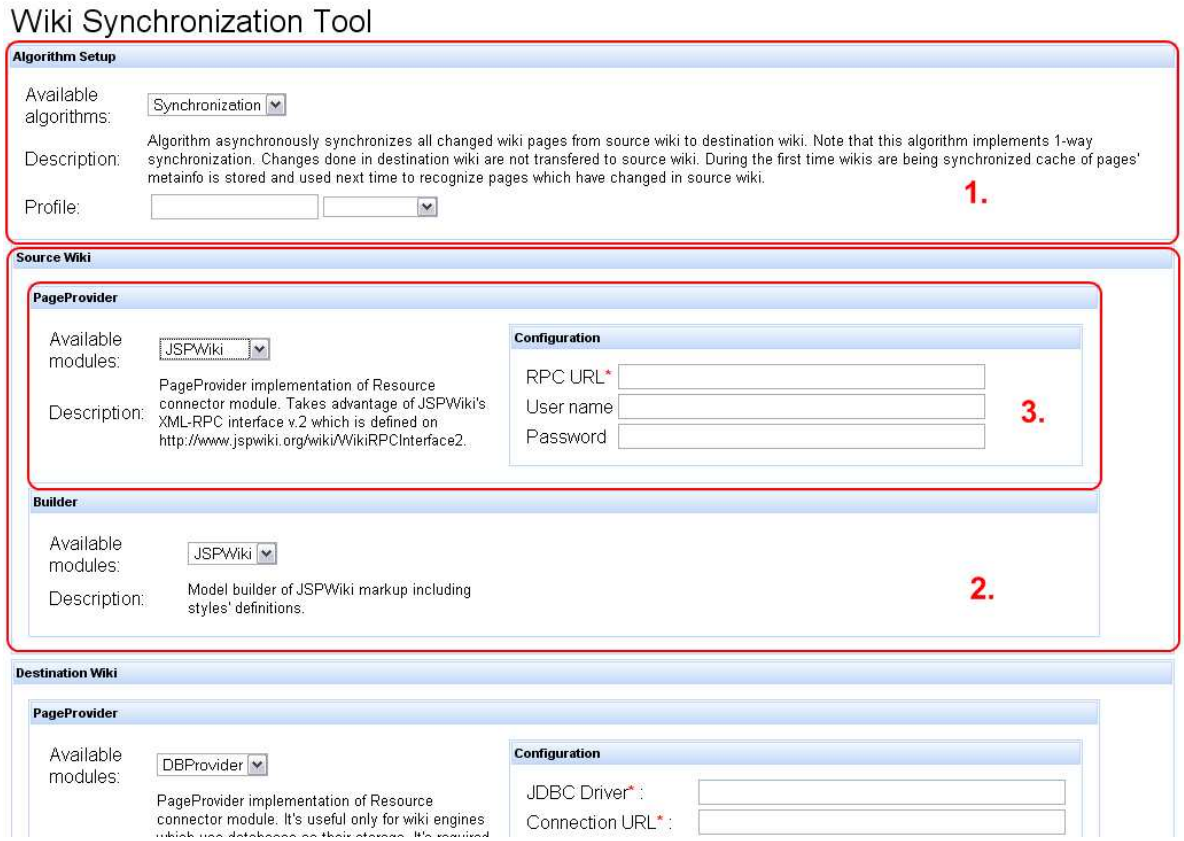


Figure 6.1: WikiSync main page

individual facades (2.). Every facade configuration is also divided into several parts - configurations of particular modules required by the facade (3.).

I'll focus on each of the parts and describe it in detail.

Algorithm selection section shown on Figure 6.2 allows the end user to select algorithm which will be run. Note that if user selects different algorithm all the other parts of configuration page are reloaded to reflect algorithm's configuration. Algorithm selection section also offers short description of currently selected algorithm and an input for profile name. Profile name is an identifier of the current configuration which is being created and it can be reused the next time. The user just simply selects one of the saved profiles which are available on the right from profile input and the configuration which was used last time will be loaded. Facade configurations con-

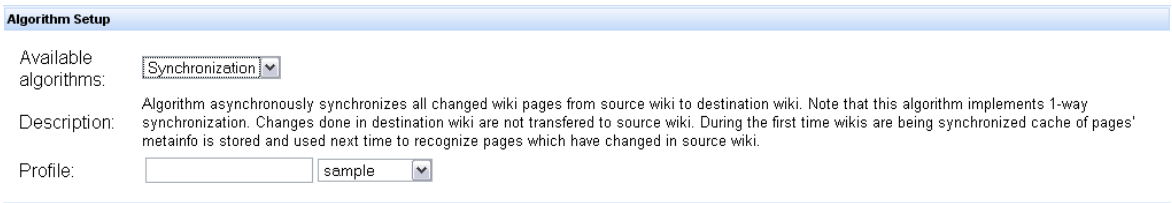


Figure 6.2: Algorithm selection

sist of modules' configurations which are required in order to implement operations requested by the algorithm. To distinguish facades they have assigned names like 'Source wiki', 'Destination wiki' etc. These names should help identify the end user what kind of configuration is expected. The purposes of facades should be described in algorithm description. Please take a look at Figure 6.3 to the left top corner to see the facade name. Every module configuration consists of three parts - module imple-

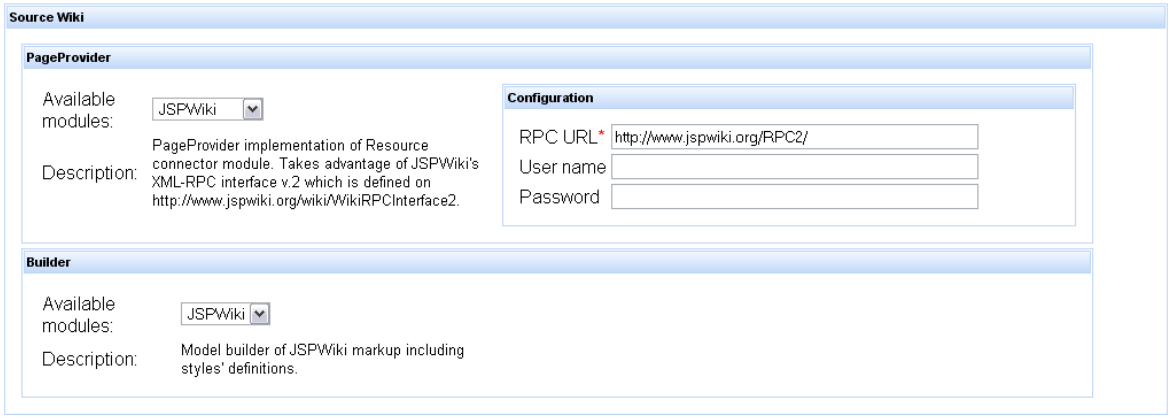


Figure 6.3: Facade configuration

mentation selection, module description and configuration of module properties. If the end user selects different module implementation, module description and module configuration properties are reloaded. Module description should contain basic information about the module necessary for correct configuration such as what kind of connection to wiki is created etc. Module configuration properties are displayed only in case module requires some configuration. There are two types of configuration properties - mandatory which are marked with red star next to the property name and optional. If some of the mandatory configuration properties is missing the configuration is incomplete and must be corrected. The names of configuration properties should be self-explanatory but sometimes it is worth to provide further description about particular configuration property. This description is shown when user moves the cursor above configuration property name as shown on Figure 6.4. When the configuration of algorithm and all the facades and modules is prepared

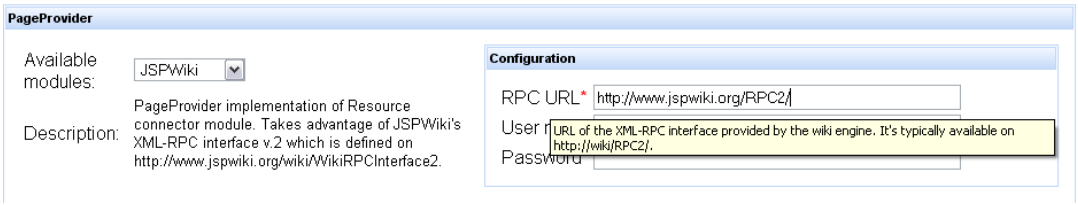


Figure 6.4: Module configuration

the user is expected to click on the 'Next' button on the bottom of the main page. If the configuration was correct - all mandatory properties were fulfilled and the values were right - the algorithm main page is loaded. If not the error is shown at the top of

the main configuration page and user is expected to fix the configuration and resend it.

6.2.2 Synchronization algorithm

In this subsection I would like to describe UI of One-way Synchronization algorithm. Current version of application also implements Migration algorithm whose UI is simpler variant of Synchronization algorithm's and thus I don't think it is necessary to describe both.

When algorithm is properly configured and button 'Next' on the main page is clicked, the algorithm main page will be loaded. Figure 6.5 shows this page. In its initial state it contains only 'Overview' section with control buttons. The 'Overview'

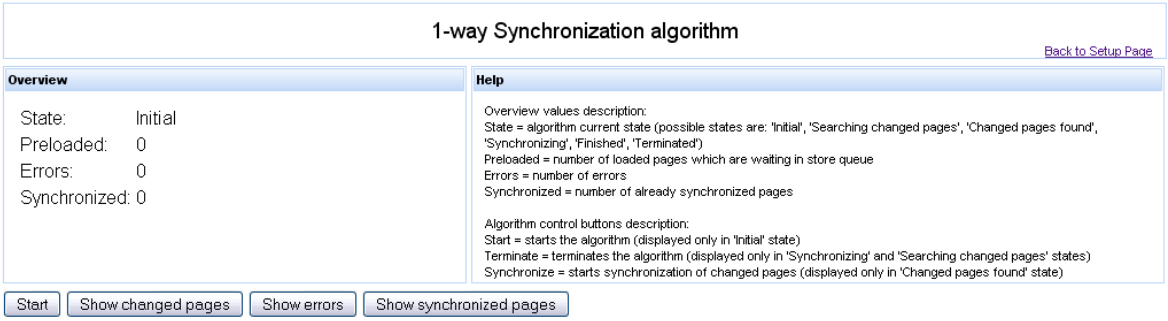


Figure 6.5: Algorithm control panel

section contains overview information about the progress of the algorithm. There are information such as current state of the algorithm, number of preloaded pages waiting in store queue, number of errors which have appeared till now and a number of already synchronized pages. Description of these information is also provided on the right in the 'Help' section. There is several control buttons under the 'Overview' section. The first of them is state control button. Possible values of are 'Start', 'Synchronize' and 'Terminate'. It is displayed in case the end user is allowed to change current state of the algorithm. The rest of buttons are switches which are used to show and hide other sections of the UI.

Transitions among states of One-way Synchronization algorithm are: 'Initial' -> 'Searching changed pages' -> 'Changed pages found' -> 'Synchronizing' -> 'Finished'. There is one more state called 'Terminated' which can be achieved from states 'Searching changed pages' and 'Synchronizing' by clicking on 'Terminate' state control button. User is required to change algorithm state from 'Initial' state to 'Searching changed pages' state by clicking on 'Start' state control button and then from 'Changed pages found' state to 'Synchronizing' state by clicking on 'Synchronize' button.

After algorithm is started from its 'Initial' state by clicking on 'Start' button it starts to search for pages which have changed since last synchronization. To recognize changes it uses a cache of meta-information of pages created last time algorithm was run. To be more specific the algorithm compares current page version number with

the version number which is cached and if it is different then marks the page as changed.

When searching of changed pages is finished, algorithm changes its state to 'Changed pages found'. The user then can display list of all changed pages by clicking on 'Show changed pages' button and decide which pages should be really synchronized and which not. Figure 6.6 shows this section. By default are all pages in the

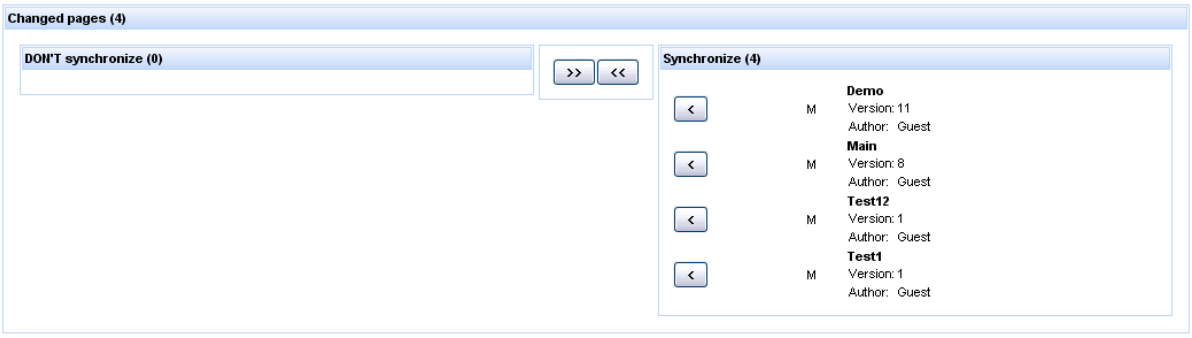


Figure 6.6: Changed pages section

'Synchronize' section but the user can move them to 'DON'T synchronize' section by clicking on '<' (move just the page) or '<<' (move all pages) buttons. When the user is decided which pages should be synchronized and which not he clicks on the 'Synchronize' state control button and algorithm starts synchronization process.
There are two extra sections displayed when algorithm is in the 'Synchronizing' algorithm state. First one is the section with currently loading wiki pages from the source wiki engine. Second one with currently storing wiki pages to the destination wiki. These two sections are visible on Figure 6.7. Loading pages section displays

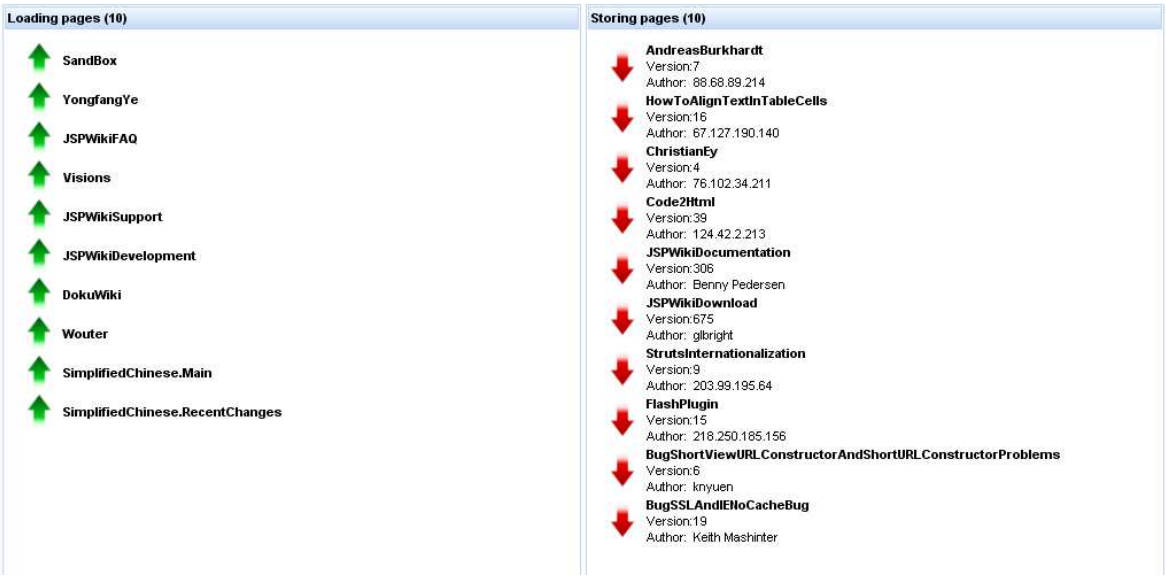


Figure 6.7: Loading&Storing pages

only the names of wiki pages. Storing pages section displays the names of wiki pages

plus Author name and version number of the page in the source wiki engine. This information cannot be displayed in the Loading pages section because they're not available yet.

Anytime the user clicks on 'Show errors' button, the section with errors that occurred till now will appear. You can see the section with one error on Figure 6.8. The information about error says that the error occurred during loading phase of

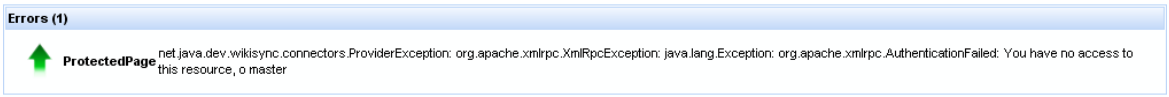


Figure 6.8: Errors during synchronization

'ProtectedPage' page from source wiki (green arrow). In case the error occurred when page was saved to destination wiki red arrow would be visible. The information also contains what caused the problem. In this case the error occured due to failed authentication - the page is not readable for everyone.

The last section which can be displayed to the user is list of pages which were already synchronized. This section can be displayed by clicking on 'Show synchronized pages' button. Example screenshot is on Figure 6.9. When the algorithm state

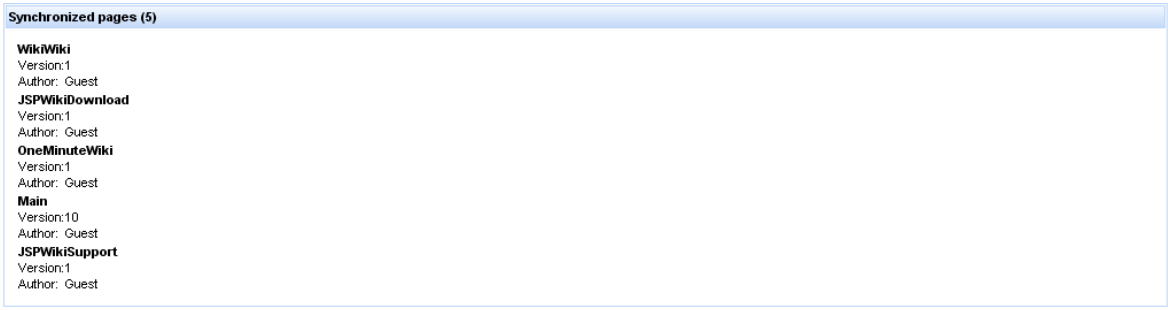


Figure 6.9: Synchronized pages

changes to 'Finished' it means, that all the pages which should have been synchronized are synchronized or an error occurred during their synchronization. This is the final state of the algorithm.

As mentioned earlier Migration algorithm is simpler variant of this algorithm. It has only following states: 'Initial' -> 'Migrating' -> 'Finished' and 'Terminated'. The UI control is the same is in case of Synchronization algorithm, there are just two steps missing - searching of changed pages and choosing which pages will be synchronized and which not. It is implemented this way simply because the Migration algorithm migrates all the pages. The rest is the same.

Chapter 7

Conclusion

The work on this thesis wasn't easy. I had to study different wiki engines, their architectures, markup languages that they use and analyze the differences among them properly. My knowledge of Java programming language had to be improved to be able to suggest sensible and modern implementation solutions. I think that I have succeeded in most of these skills and created a work which offers compact overview of the whole problem. The work which brings the review of existing wiki engines, their architectonical solutions, markup languages and provisioning APIs.
This work in my opinion fulfils all the requirements which were stated in the original assignment and goes beyond that. The solution which is being described in this thesis is an automated web based tool that can be among other things used to migrate and synchronize content between different wiki implementations. In contrast to existing solutions this project is the only one which is really engine-independent.

The presented tool offers very sophisticated and modern design solutions suitable for future extensions. Modules are simple, reusable, independent and loosely coupled. Algorithms are also simple and totally generic and thus usable for various purposes, not only for migration and synchronization. On top of that modules and algorithms are separated from each other which makes the whole architecture very flexible and prepared for future improvements and extensions. The presented unified object model of wiki content is unambiguous, extensible and intuitive which makes of him ideal solution for newly originating wiki engines whose content is represented in object form. It is true that the user interface is not perfect and also that the current version of the application could contain more implementations of modules and algorithms. On the other hand neither the user interface nor the exhaustive number of implemented modules and algorithms were the goals of this thesis. It is the concept that is worth. New implementations of algorithms and modules will be hopefully added by incoming contributors because the whole project is open-sourced.
I believe that this project has a long-lasting bright future ahead of it and that the base of users and contributors will expand soon.

Appendix A

Installation CD layout

`glassfish` - GlassFish installations for various OS
`javadoc`
- `core` - generated documentation of core classes
- `ui` - generated documentation of UI classes

`source`
- `core` - core source codes, tests
- `ui` - UI source codes, JSF pages, web configuration files etc.

`README` - description of files on the CD
`thesis.pdf` - digital version of this thesis
`wikisync.war` - distribution web-archive of the application

Appendix B

Implemented modules

Resource connectors

- JSPWiki - `PageProvider` which provides connection to JSPWiki. It takes advantage of JSPWiki's XML-RPC interface v.2 which is defined on http://www.jspwiki.org/wiki/WikiRPCInterface2.
- MediaWiki - `PageProvider` which provides connection to MediaWiki. It takes advantage of MediaWiki's REST API whose definition can be found on http://www.mediawiki.org/wiki/API.
- DBConnector - `PageProvider` which provides connection to wiki engines that use databases as their storage. It is required that user has access to the database (via JDBC connection) and knows its schema to be able to define correct SQL statements required by this module.

Model builders

- JSPWiki - Model builder which parses content in JSPWiki markup including styles' definitions and builds object representation of it.

Model processors

- MediaWiki - `MarkupSerializer` which serializes object representation of wiki content to textual representation in MediaWiki's markup language.

Cache providers

- ServerFile - Cache provider module implementation which uses files saved on server as a cache storage.

Appendix C

Implemented algorithms

- Migration algorithm - Algorithm migrates all wiki pages from source wiki to destination wiki. Pages already present in destination wiki are overwritten. The implementation uses asynchronous operation calls.
- Synchronization algorithm - Algorithm synchronizes all changed wiki pages from source wiki to destination wiki. Note that this algorithm implements one-way synchronization. Changes made to destination wiki are not transfered back to source wiki. The implementation uses asynchronous operation calls.

Appendix D

```
**bold**
//italic//
{{{monospace}}}

* bullet list
* second item
** sub item

# numbered list
# second item
## sub item

== Large Heading
=== Medium Heading
==== Small Heading

[[URL|Title]]

horizontal line:
----

line break here\\another line

{{{
//This// does **not** get [[formatted]]
}}}
```

Figure 7.1: Demonstration of WikiCreole Markup

Bibliography

[1] http://en.wikipedia.org/wiki/Wiki

[2] http://www.wikicreole.org/wiki/WikiPopularity

[3] http://www.wikicreole.org/wiki/Goals

[4] http://www.wikicreole.org/wiki/Engines

[5] http://www.jspwiki.org/wiki/WikiRPCInterface2