Independent Coursework

Support software for Wiki content migration

Daniel Senff

s0514457

HTW Berlin / University of Applied Science

Internationaler Studiengang Medieninformatik

International Media & Computing

Summer semester 2011

Introduction

Abstract

Wikis have become a very important tool of collaborative knowledge documentation. Many different software exist that each have a unique approach, each with its advantages and disadvantages. Over the years software solutions emerged and discontinued, which creates a problem for people using the system, since there are very often little means of transferring the contents and history into a different wiki solution. The only notable exception is *MediaWiki* whose community is large enough to support such special utilities. Smaller wiki-engines without large community or even legacy wiki-software without any continued support usually do not have the means. In one way or another the content of the Wiki including its history is stuck in this system.

This project will attempt to address the issue by creating a command-line tool to easily migrate and archive Wiki contents from legacy software systems. This includes the topics data storage, formatting syntaxes, configuration and will also document the solutions proposed for this software based on the requirements described in later chapters. The goal is to create a software that allows to export all of the history of a Wiki into a new common modern git-based wiki-system to archive all of its data in a way that retains the history and is quickly accessible.

This paper will focus on the technical issues involved in data migration of Wikis and will not feature the social and community aspects that can arise by such a move.

Motivation

Having used Wikis as a tool in private as well as in professional contexts very successfully for many years I run into the problem, that wikis have little means of exporting data to other wiki engines. Since I did not want to discontinue my original wiki and start a new on a new software, I was looking for means to transfer the data. My wiki running *WikkaWiki* in particular had no bridges to other Wiki-databases and when a similar problem came up at my company, I saw the opportunity for a new utility that could help creating and giving an easy way of configuration to export Wiki into a new general-purpose container.

Basic

Wiki software

The term and idea are attributed to Ward Cunningham. In his own words he describes a Wiki as “a piece of server software that allows users to freely create and edit Web page content using any Web browser. Wiki supports hyperlinks and has a simple text syntax for creating new pages and cross-links between internal pages on the fly.”(Ward Cunningham) [1]

Wikis are a software for collaborative text work. Wikis became popular with the ascent of *Wikipedia*, an online Encyclopedia based on the Wiki-technology. Today, Wikis are a common tool for documentation and serve as knowledge databases for projects and organizations. [2]

Over the past 10 years many different Wiki engines has been developed. Among the most popular is *MediaWiki*[1], which is the technical base for Wikipedia. Other software projects did not reach similar popularity and many discontinued.

The basic premise of Wikis is that articles are edited in a web-based interface either by a defined group of users or open to everyone. This editing process is serial in contrast to later tools like *Google Wave*[2] or *Etherpad*[3] which allow parallel editing by multiple users.

A wiki can contain multiple articles, which are stored under a unique name and link to each other thereby creating a network of articles. Whenever an article is edited, the changes are saved in a new revision, thereby saving the full article history within the database. Each revision has an associated author and an optional message describing the changes done. Many Wikis allow for very detailed user rights management.

1 Website of MediaWiki: http://www.mediawiki.org/

2 Website of Google Wave: http://googlewave.blogspot.com/

3 Website of the Etherpad project: http://etherpad.com/

Other than these common features of Wikis are many special features that do not have wide support. Namespaces are a way to create articles in special contexts and allows for a hierarchical data structure, instead of a flat data structure without namespaces. Another special feature are templates, which describe how certain structured data should be displayed within the wiki. Many Wikis have a plugin interface that allows to add new functionality to the software.

Data within a wiki is not strictly formatted and reading semantic information from wiki articles requires a lot of effort and strict conventions. The article in itself is just a test document. Only by context, semantical networks and a special markup language, semantical information can be derived.

Wiki markup

Wiki markup describes a kind of lightweight markup languages that are mostly used in Wiki-contexts. The purpose of this text markup is to give semantical as well as stylistic formatting to plain text data. As Wikis are rendered as HTML-documents, this wiki-markup is usually a subset of HTML-markup features, although in a different, more simplistic written syntax. [2]

As *MediaWiki* is very widespread as wiki-software so is its markup language Wikitext[4]. Many wikis have a dialect of this language, although differences can range from small to very big. Special and important for MediaWiki is its rich template engine, a feature, not supported in most wikis. MediaWiki-markup has been criticized for being hard to parse by formal syntax parsers.

Wiki markup usually has three kinds of formatting markups:
Inline modifiers is formatting markups that is within the line of a document, for example for highlighting texts.
Block modifiers define formatting for the complete line. For example images or list items are often defined per line.
Multiline syntaxes describe more complex structures like tables with each line defining a new table row. A few examples of each are attached in Appendix 3.

Other than the legacy wiki markups inherited by the developed engines there are wiki

4 WikiText Markup of MediaWiki: http://en.wikipedia.org/wiki/Wikitext

markups independent of any particular wiki-engine. These languages have been developed and are used across a wide range of projects. Using the same syntax allows for quick data conversion and therefore addresses the same problems as this paper.

Two very widespread markups that spawned from the Ruby- and Python-Community are *markdown*[5] and *textile*[6]. Markdown was developed to be easily written, readable and is loosely based on eMail formatting conventions. Textile is more complex than markdown, supports a wider array of features, but is also a bit more technical to write. Markdown can be understood as the lowest common denominator while textile is a good choice if markdown is not good enough. Both have a wide support across multiple scripting languages and are available as implemented libraries for different platforms.

git

After managing the development of the Linux Kernel for over a decade and the experiences of different source code management systems in that time, Linux Torwalds decided to develop his own decentralized Version control system. In April 2005, *git*[7] was announced and developed by Junio Hamano and Linus Torvalds. By 2011, git has become one of the major source code management systems. Its strength being its decentralization, which allows cloning complete repositories and working on the projects locally.[4] Branching and merging received a new meaning, a much quicker workflow and allowed for non-linear project development. The repository history is cryptographically authenticated and not at last in hosting the Linux-Kernel git proofed its reliability and very good support for large projects.[5] A community grew, that not only continued its development, but also build third-party tools and libraries to connect git to other programs and use it in other programming languages.

Since git basically stores history of plain-text-files, the idea to use git as the database for Wiki-documents came up in several contexts. Projects emerged, that used this paradigm successfully (see gollum), while others had nice ideas on paper, but did not

5 Website of Markdown project: http://daringfireball.net/projects/markdown/

6 Website of Textile project: http://textile.thresholdstate.com/

7 Website of git: http://git-scm.com/

yet succeed to show results (see Levitation). Git's advantages as data storage for wikis are the native support of full history, commits and differences analysis between articles. In each commit, git stores the changes (diffs) done to the file, thereby making it easy to trace the history of each file, or in this case each article. These database of changes is compressed by git, to require least amount of storage, while still being quick to access. Lastly, all commits within git are identified with a cryptographic hash, which changes, if a commit is changed afterwards. This ensures a secure and untempered project history.

Ruby

Ruby[8] is a scripting language developed by Yukihiro Matsumoto in 1995. While it was designed with similar capabilities to Perl and Python, its main focus was to be human-oriented by offering a simple well-readable code style and very predictable programming. While other languages follow the approach that there should only be one right way to solute a problem, Ruby tries to offer a rich and sometimes redundant set of features to perform operations as graceful and close to the human conception as possible. [6]

Ruby is completely object-oriented without any native datatypes. Everything is an object and Ruby comes with a rich standard library.

At the time of this writing (August 2011) Ruby is currently available and used in two versions. Version 1.8.7 was released on July 1st 2008 and is the current stable branch and supported by all libraries and projects. Version 1.9.2 was released in August 2010 and introduced several new features and major changes, that made the new version incompatible to the existing 1.8.7. This requires scripts written for version 1.8.7 to be updated and this process is currently ongoing within the community. As a result it is easy to encounter libraries and tools, that do not work in Ruby 1.9.2. [7]

The Ruby community developed ways for easy distribution of scripts. *RubyGems*[9] is a well established deployment and packaging tool for Gems. Gems are packaged libraries of Ruby-scripts: either executables, libraries or third-party addons for other

8 Website of Ruby: http://www.ruby-lang.org/en/

9 Website of RubyGems: http://www.rubygems.org/

Ruby software. With RubyGems building and distribution becomes very comfortable.

For project building a tool called *rake*[10] emerged. It serves a similar purpose to *make* or *ant* and offers easy project building and helper tasks. Rake interprets task-definitions written in Ruby. Both RubyGems and rake will be used during the development of the project proposed in this paper.

Concept/Analysis

Goal

Data migration of documents edited and contained in a wiki environment is not a new topic. Software has been written for various Wiki-engines to convert data. Especially for database-based Wiki engines writing simple conversion tools to transfer data from one database into the next does not involve much complexity. However this very often includes many compromises and especially markup-conversion takes manual work to finish the process. Existing conversion tools have mostly been developed for widespread wiki-engines such as MediaWiki. Conversion systems for legacy or uncommon engines however are hard to come by and users who decided on their Wiki-software have very limited choices when they want to switch their wiki-engine. They have to write their own conversion tool, which can be a major effort or at least a time-consuming inconvenience depending on the programming experience of the user.

The project proposed in this paper tries to establish a small framework for wiki-migration into a git-based target wiki-engines. The software should provide a set of preset connectors for selected Wiki-engines and an extendable interface for configuring new connectors. For a full content migration markup conversion also needs to be performed and the software should provide a base implementation of a markup converter to convert the proprietary wiki syntax to more common markup languages, which are in wider use today.

The software should be a simple to use command-line tool that can be easily installed by a package management system and provide example configurations and

10 Website of rake project: http://rake.rubyforge.org/

documentation.

The project's name is caramelize calling for the metaphor of changing sugar to caramel by melting, thereby changing its consistency but still retaining its sweet properties.

Requirements

Required for achieving this goal was finding the individual components, which would have to interact together to fulfill the desired function.

To focus on the work on the data migration and transformation process. This transformation process should work on small to medium sized wikis with an article length of up to 1MB. The software is not thought as a way to process Wikipedia-dumps which are magnitudes larger than the proposed sizes. It is expected to retain all versions of each document in the wiki and make it as browsable as possible in the new engine. Since git will be used as underlaying infrastructure, its rich features of commit and diff-comparisons should be available.

The programing platform chosen for the project is Ruby and should use preferably only Ruby libraries to build upon. The software should build as a Gem and be deployed within the RubyGems package system.
By doing so, the software should be able to run on any Ruby environment. In this regard it is important to pay attention to the different Ruby-versions 1.8.7 and Ruby 1.9.2, which are not fully compatibility towards each other. Libraries and implementation should be chosen and done in a way to work with both versions.

The software itself should support selected Wikis out of the box. The focus hereby is not to support the most-widespread, but to give an example implementation for users to extend. Aim is to have a system that is modular and can be extended by external configuration files. This way it should be simple and give a base framework for users to adapt to their wikis.

Since the application's purpose is to migrate and archive data, it should not change the existing stored data, but copy them without change. For markups this will mean, that the target wiki will not display the markup correct, as the syntax may be different. For this another step in the migration is necessary which transforms the given wiki markup to a new markup supported by the target system. This markup converter should also be an example implementation from which even inexperienced can create a customized converter for their wiki markups. If possible this markup parsing and transformation should be supported by a high-level library, which wraps complexity as good as possible, to give the user easy to use interfaces. In order not to change history, the markup-converted page revisions should be appended to the history as new page revisions.

Previous implementations

Before outlining the specific analysis it is a good way to have a look at previous similar implementations.

Various MediaWiki-Converters

As MediaWiki is by far the most widespread Wiki engine, several solutions have been developed to export data from MediaWiki. For sole export MediaWiki supports XML-dumps of all data. Other wiki systems have much simpler exports to plain text documents or most commonly HTML. For MediaWiki more diverse solutions developed.

WikiScraper[11] is a single Perl-script that reads a MediaWiki-XML-dump, iterates over all articles, writes each into a file within an initialized source code repository. It is a prime example for a simple data migration, without many transformation of the original data. It establishes the same dataflow that is exhibited basically in all migration scripts, as the basic premise is always to iterate over all revisions and move them into the new storage environment.

Git-MediaWiki[12] is a project that does not aim at solely migrating MediaWiki's database into a git-repository, but tries to create a bridge between a live MediaWiki and a git-repository. The software is written in Perl and recognizes changes in database and synchronizes them with the git repository. On the other hand, when

11 Code repository of WikiScraper project: https://github.com/chronomex/wikiscraper

12 Code repository of Git-Mediawiki: https://github.com/Bibzball/Git-Mediawiki/wiki

pages are changed in the repository, the change is transmitted to the MediaWiki-database. This way Git-MediaWiki serves as an arbiter and offers an interesting new interface and workflow for collaborators in the wiki. While regular users can still use the normal MediaWiki-interface, customized solutions build around the git-repository are possible for power-users.

Python-Wiki-Converter[13] is a script to convert JSPWiki-markup to DokuWiki markup. It does not feature data migration; it is only about markup conversion using Regular Expressions.

Media2iki[14] is a tool for converting MediaWiki data to ikiWiki, a wiki-engine build on the premise of data storage within a version control system. This transfers all wiki contents. Markup transformation is not necessary, as ikiWiki has WikiText support by plugins.

Levitation

In the wake of the relevance discussions within the German Wikipedia in 2009, Tim Weber proposed the idea of an Omnipedia.[8] The proposal described a git-based branchable Wikipedia fork which would allow users to host their own forks of the Wikipedia database and allow tracked global editing by the means of git. Using git's tools for committing, cherry-picking and merging the aim was to build the infrastructure for multiple instances of encyclopedias with different contextual and ideological focus. Based on this idea a prototype for a bridge from Wikipedia to git-repositories was developed. *Levitation*[15] was written in python started by Tim Weber. This was a proof of concept prototype and successfully imported MediaWiki-XML-dumps into git. For this the XML-file was read within Levitation and passed to git's fast-import[16] function, which allows for quick batch committing of large amounts of structured data.

13 Code repository of python-wiki-converter: http://code.google.com/p/python-wiki-converter/

14 Code repository of media2iki: https://github.com/mithro/media2iki

15 Code repository of levitation project: https://github.com/scy/levitation

16 Manual page of git fast-import: http://ftp.kernel.org/pub/software/scm/git/docs/git-fast-import.html

The prototype works and allows for the conversion of MediaWiki-exports, however the primary goal of converting Wikipedia-exports hit technical limits. Due to the structure of the XML-file, Levitation required lots of article sorting. Data in the XML are grouped by each article and before committing this to git, it required resorting to be ordered by date. As git calculates and ultimately stores the diffs of each article version, the sheer mass of data piled up in memory exceeded the capabilities of a regular home computer. It is an issue of memory management as the conversion of 4GB large xml dumps required to much memory for levitation and git. It thereby became the bottleneck to the idea of regular Wikipedia dumps.[17]

The source code is still available and works, but the project itself has not been continued since 2009.

Supported Wiki-Software

The general issue addressed in this paper is not particular to a specific set of Wiki engines. The only criteria is the size of the community that maintains a project and keeps supporting and developing it, even after the software is discontinued. While the list sure is not complete, WikiMatrix.org lists 135 wikis. To this number comes an unknown number of outdated and legacy engines. To set a scope for this paper, two available engines who both have a certain user group, but lack third-party tools to deal with data archiving and migration.

WikkaWiki

WikkaWiki[18] is a light-weight PHP-based Wiki-engine licensed under the GNU General Public License V2. WikkaWiki is a fork of the Wakka-Wiki-engine whose development stopped in 2003. Jason Tourtelotte established continued development of the engine with WikkaWiki and published the first release in May 2004.

WikkaWiki uses its own markup that borrows some similarities with the wide-spread

17 Summary of memory problems encountered in levitation: https://raw.github.com/scy/levitation/5a4f60e8423284df578c89d68443e32d97a61894/INTERNALS

18 Website of WikkaWiki: http://www.wikkawiki.org

MediaWiki-syntax, but still has many differences regarding solutions for tables, images and even headlines. The syntax formatter of the wiki-engine has its wiki-markup hardcoded, which made it very hard to allow for alternative syntax parsing within the engine. This was frequently requested in the past years, but was apparently of no priority to the developers.

Despite being light-weight WikkaWiki features many properties that are not too common among wikis. It has a wide range of plugins for mind-mapping and code-syntax-highlighting, that are impossible to transfer to other wiki-systems, that do not have such features.

All authors and revisions of a page are stored in a MySQL database.

Even though WikkaWiki is still being developed with one major release a year, its community is too small to provide well-establish and stable means of exporting the wiki's data. Combining these mentioned factors, these made a perfect candidate.

Redmine wiki

Redmine[19] is a issue-tracking and project management software programmed in Ruby-On-Rails. It is licensed under the GNU General Public License V2. Development began four years ago and has continued steadily ever since.

Among its basic features for bug-tracking, communication, time-tracking and source-code-management is a feature for per-project wikis. Each project has its own wiki namespace.

Almost all of Redmine's user input fields are enhanced with Textile as its default syntax for text formatting. This includes the bug reports and of course the wiki pages as well.

As Redmine is based on the Ruby-On-Rails framework it supports all databases the original framework supports in version 2.3 supports.

Gollum Wiki

While there are a few solutions for storing wiki-data in git-repositories, Gollum is by

19 Website of Redmine: http://www.redmine.org

far the most widespread. Gollum is actively developed by github who use it for project wikis on the development platform by the same name. The library was made Open Source and is available under a MIT License. Strictly speaking, gollum is not just a library, but also features a small web-server for Wiki-hosting and is extendable by many third-party plugins. Gollum wiki's can be included in Rails-applications. This offers a large array of possibilities to continue processing and using the stored data. Its web server makes it very easy to open archived wiki data for review. Once installed as a gem, gollum requires no additional project folder than the wiki's git repository itself.

Gollum features support for many markup languages most notably markdown, textile, creole and Wikitext.

ikiWiki

Described as wiki compiler, ikiWiki[20] has a similar, but more complex setup then gollum. Wiki documents are stored as source data in a Version control system such as Subversion or git and ikiWiki compiles HTML documents from the sources. These are made available by means of a web server.

IkiWiki is written in Perl and is available in many Linux package management systems for quick and easy installation. Out of the box it supports markdown markup, but by the means of plugins support for textile or WikiText can be added.

Markup parsing library

Requirements

As WikkaWiki has a unique formatting markup, an important part of this project is the parsing and eventual transformation of different syntax dialects to a specified target syntax. The markup conversion implemented in the project should serve as an example implementation which users can customize to meet their needs for markups not natively supported. Since this syntax transformation is not a trivial task it should be evaluated, what libraries are available that can help in this process.

20 Website of ikiWiki: http://ikiwiki.info/

Libraries should be written in native Ruby and be Open Source. Preferably they are available as Gems in the RubyGems package system.

The libraries should help writing syntax parsing grammar. In case of full Expression Grammars, the libraries can generate a tree structure of the documents, on which the syntax conversion can be performed. Abstraction and complexity reduction in the library is welcome and ease-of-use is more important than the amount of features. After all the library should be the base for syntax-converters written by users who never had experience with compiler-building. The library should help with this task and should not be in the way of finding a simpler solution, if they don't prove viable.

Existing solutions

An early part of the analysis and preparation for the project was research into already existing markup and syntax parsing solutions. Based on the requirements described above the goal was to see which libraries are good candidates for the project and to see where the advantages and disadvantages are.

racc

Racc[21] is a Ruby implementation of the Unix tools *yacc*. It serves for lexical analysis and syntax parsing. Racc - like yacc - describes deterministic Context-Free Grammar by their syntax definition.

`class Calcparser`
`rule`
`  target: exp { print val[0] }`
 	
`  exp: exp '+' exp`
`    | exp '*' exp`
`    | '(' exp ')'`
`    | NUMBER`
`end`
	1 + (3 * 37)

Table 1: Example definition using racc

While there is lots of yacc technical documentation, there is little documentation to get started specifically for racc. Its purpose is to give a good yacc-like interface within Ruby. This abstraction is not very verbose, yet powerful. Racc is low-level and is a complex endeavor. As such it is a powerful tool for power users comfortable and

21 Code repository of racc: https://github.com/tenderlove/racc

experienced with yacc. For inexperienced users it is far from suitable for an application like the one described in this paper.

treetop

Next to the re-implementations of Unix syntax parsing tools are high-level approaches that wrap the complexity in to well-defined grammar definitions. *Treetop*[22] establishes its own definition format and offers function-based programmable grammar definitions. These are written and interpreted in Ruby. It is a very rich tool and offers good debugging capabilities. It is licensed with an Open Source MIT license.

```
grammar Arithmetic
  rule additive
    multitive ( '+' multitive )*
  end

  rule multitive
    primary ( [*/%] primary )*
  end

  rule primary
    '(' additive ')' / number
  end

  rule number
    '-'? [1-9] [0-9]*
  end
end
```

47 * (1 + 1)

Table 2: Example definition using treetop[23]

Treetop requires to have a good understanding of Parser Expression Grammars. It is a powerful tool to work with and it simplifies things for large parsing solutions. However it does not help beginners of the subject to easily get into parsing grammar. The learning curve is rather steep and without prior experience it takes some time to understand its working and implement a full syntax coverage. There is good documentation and links to reference implementations of some known programming syntaxes, yet the required effort of writing a full parser with treetop is too large to recommend it for use in this project. It is supposed to be used in the configuration file for the wiki-export, this kind of effort can not be expected by the target group of users

22 Website of treetop: http://treetop.rubyforge.org/

23 Example from treetop introduction: http://treetop.rubyforge.org/

for the software.

parslet

Parslet[24] is another Ruby library for writing high-level Parsing Expression Grammars. These definitions are written in pure Ruby and given to a parser who creates a document tree based on the defined grammar rules. Compared to treetop, its parsing definitions are shorter, but also more technical to read. Where treetop outsourced the definitions into its own grammar-definition-files, parslet keeps the definitions within the ruby-code. However with little effort this could also be outsourced using native Ruby means.

What makes parslet a very interesting solution is, that it already has build-in support for tree-transformation. The data tree of a parsed document can be transformed into other tree structures.

This is a short example of a grammar definition.

```
class MiniP < Parslet::Parser
  # Single character rules
  rule(:lparen)     { str('(') >> space? }
  rule(:rparen)     { str(')') >> space? }
  rule(:comma)      { str(',') >> space? }

  rule(:space)      { match('\s').repeat(1) }
  rule(:space?)     { space.maybe }

  # Things
  rule(:integer)    {
    match('[0-9]').repeat(1).as(:int) >> space? }
  rule(:identifier) { match['a-z'].repeat(1) }
  rule(:operator)   { match('[+]') >> space? }

  # Grammar parts
  rule(:sum)        { integer.as(:left) >>
operator.as(:op) >> expression.as(:right) }
  rule(:arglist)    { expression >> (comma >>
expression).repeat }
  rule(:funcall)    { identifier.as(:funcall) >>
lparen >> arglist.as(:arglist) >> rparen }

  rule(:expression) { funcall | sum | integer }
  root :expression
end
```

puts(1 + 2)

Table 3: Example definition for a parslet parser[25]

24 Website of parslet: http://kschiess.github.com/parslet/

25 Example from parslet introduction: http://kschiess.github.com/parslet/get-started.html

Parslet suffers from a very similar problem as Treetop did. It is a very powerful tool for people who already know what they are doing. Existing syntax parsing definitions may be easy to read and quick to understand, but they are very tough to create and this makes it also not viable for use in the proposed project.

Regular Expressions

Regular Expressions are a formal language for string and pattern matching.

Compared to high-level solutions like treetop and parlset, Regular Expressions are rather rough, very basic and low-level. Regular Expression follow a different approach in that they do not attempt to parse a document into a graph tree. Regular Expressions give the means to parse the document for defined patterns and it is up to the programmer to do what he wants from here. With effort you can build document trees, but it is not their purpose.

`\((\d*) ([\+|\-|\*|\/]) (\d)\) ([\+|\-|\*|\/]) (\d*)`	(46 + 6) * 64

Table 4: Example Regular Expression for parsing a formula of predefined format

In the context of this paper, Regular Expressions can be used to match markup patterns, deconstruct their properties and substitute them with target markup. The process itself is not complex, however constructing correct pattern expressions for each markup modifier can still be tricky as issues like property order and white-line matching can make patterns difficult.

Another difficulty is that different dialects of Regular Expressions exists, which ware not fully compatible. Also each Regular Expressions engine is slightly different in interpreting these dialects and may not implement the full set of features. For example native support for Regular Expressions in Ruby 1.8.7 omits look-behind and only has been added in version 1.9.

Programming framework

Requirements

While a simple executable script-file may be enough for small functional scripts, this project would require more structure as several classes for different data types and different Wiki-connectors are needed. So having a framework that gives a basic file structure and means of building the project is very convenient and gives many advantages.

Packaging, deployment and installation of the software should be as easy as possible. If possible using a pre-existing packaging and distribution system. A framework should support this and help with otherwise tedious tasks commonly required in software development.

Having the possibility to include a test-framework is optional, however of advantage in case testing becomes relevant during the development or afterwards.

As there exist two Ruby version at the moment (Version 1.8.7 and Version 1.9.2) the framework should be compatible to both versions.

The executable would allow for several command-options to create a template configuration file and to execute a given configuration.

Existing frameworks

Writing scripts and software for command-line usage is a very common use case for the Ruby scripting language. In 2003 development on RubyGems began, creating a package management system for Ruby. Since then RubyGems allows easy installation, update and removal of installed packages. This would be the required distribution-system mentioned above and since 2003 has become standard within the community.

Packages are called *Gems* and include meta descriptions as well as dependency definitions. Dependencies are managed by RubyGems. Gems can be libraries, plugins for third-party frameworks and executables, the latter being the use case for this project.

For the purpose of creating Gems several frameworks emerged that help to setup the basic project structure and utilities to build and deploy the Gem.

The file structure within Gems follows several conventions. See table 4 on page 24 The bin/ directory contains the executable binary, the lib-directory contains all

classes, test/ or spec/ contains all testing, depending on the test-framework that is used. As this structure is independent of the framework used, it becomes very easy to switch between frameworks if necessary.
Three candidates were evaluated for this project. All use rake as build-tool and follow the conventional file-structure.

Bones

Bones[26] is a tool to create light-weight skeletons for Gem-projects. This skeleton serves as a starting point for your own project. The generated skeletons can be customized, incase you create new projects regularly. It includes several rake tasks for building and deployment as well as maintenance features such as todo-lists and debugging help.

On the evaluation of the library an compatibility issue came up, that aborted all rake tasks on Ruby version 1.8.7. This was due to an unresolved dependency of Bones with rake, which caused problems with the mentioned Ruby-version. The issues was reported, but was still open at the time of this writing. Without support for Ruby 1.8.7 this framework was no longer an option, despite it is initial good impressions.

Jeweler

The oldest, most widespread, but also largest gem framework is *Jeweler*[27].
Jeweler has wide array of features and rich rich rake helpers for building, testing and maintenance. It also has generator rdoc API documentation.

Jeweler allows for many options in creating projects and hooks for additional frameworks such as testing frameworks like rspec or shoulda. The user is endorsed to follow a very strict project structure. Git is an integral part of the building process as all files, versions and change logs are tracked by it. As github is a very important project hosting site for the Ruby community as it is regarded as the platform of choice for development and deployment. Jeweler has build-in support for many features using github's open API.

26 Website of Bones: https://github.com/TwP/bones

27 Website of Jeweler: https://technicalpickles.github.com/jeweler

Jeweler is a fully integrated gem building solution and much more than just a simple framework or a simple building tool. It is a great tool for large scale gem projects, with community and a large set of convenience helpers. However for novice users this amount of features can be overwhelming and in this regard Jeweler targets for the experienced user.

bundler

Gembundler[28] or short Bundler is a command-line software written in Ruby to manage Gem-dependencies in Ruby-projects. Bundler creates a definition file in which the required libraries are referenced. Bundler can check against this dependencies, resolves them and installs all necessary libraries. This way we have a clean definition of dependencies and a comfortable way of installing them.

Bundler also includes utilities for building gem-projects. This includes a project template and rake-tasks for building, installing and deploying Gems.

Bundler was the most light-weight of all of the tested frameworks and performed immediately without problems. Another big advantage is, that by using bundler for dependency resolution you do not need another library for project building as bundler is used for dependency management anyway. For these reasons Bundler was the framework of choice for this library. Not because it is as large as possible, but because it is as slim as can be.

28 Website of GemBundler: http://gembundler.com/

Implementation

Project setup

After evaluating different gem frameworks Bundler was the framework of choice dealing with building and deployment to RubyGems.org.

Bundler assumes the project is under git-version-control, by which it determines which files belong to the project and what is packaged into Gem-packages. This package is hosted at RubyGems and this way available to all RubyGems users. The software is open source on MIT License. The source code is hosted at github and can be customized or forked. It requires rake for building and builds on both Ruby 1.8.7 and Ruby 1.9.2. Other dependencies as well as meta information are defined in the "caramelize.gemspec" within the project source. Based on this gemspec, Bundler packages the project into a gem.

Project structure

No matter the gem framework, gem projects have by convention the same directory structure and require few mandatory files.

An important part of most Ruby-projects is setting up a full test suite. Several behavior-driven testing frameworks have been developed that help in this process. However this is not considered and is outside of the scope of this project.

File/directory	Description
bin/<command name>	Executable shell-script starting the application process.
lib/	Model, Controller and View classes of the actual application.
test/	Functional and unit tests.
<project name>.gemspec	Gem definition file, giving meta description and dependencies.
Gemfile	Defines project dependencies, automatically created by Bundler
Rakefile	Defines customized rake tasks.
LICENSE	License of the project
README	Readme describing use and building the project.

Table 5: Directory convention for ruby gem project

Dependencies

Although dependency management is all dealt by Bundler and can be managed very easily, The amount of required libraries was kept to a minimum. Each chosen library may however come with their own gem-dependencies. During the development no dependency conflicts arose as the project is too basic for that.

Required for running caramelize as a gem are “mysql2” as a MySQL-client connector to open connections to databases of configure Wikis.

“cmdparse” is a small framework for a command-line interpreter. It helps to create a command-structure and options-management within the application, so that parsing command-line options can be parsed in a sophisticated way.

Lastly “gollum” is required, as it is chosen as the target Wiki software to which the data are migrated. Gollum depends on grit, which is a ruby-based adapter to command-line git. Through grit, gollum manages its data within the Wiki's repository.

For building the project by source code, Bundler and rake are required. Bundler will also resolve the dependencies mentioned above.

Class structure

Structurally, caramelize is roughly based on the MVC-pattern, although, it does not actually have a View, as the only visual output is directly to the command-line. For the full class hierarchy, see Illustration 1.

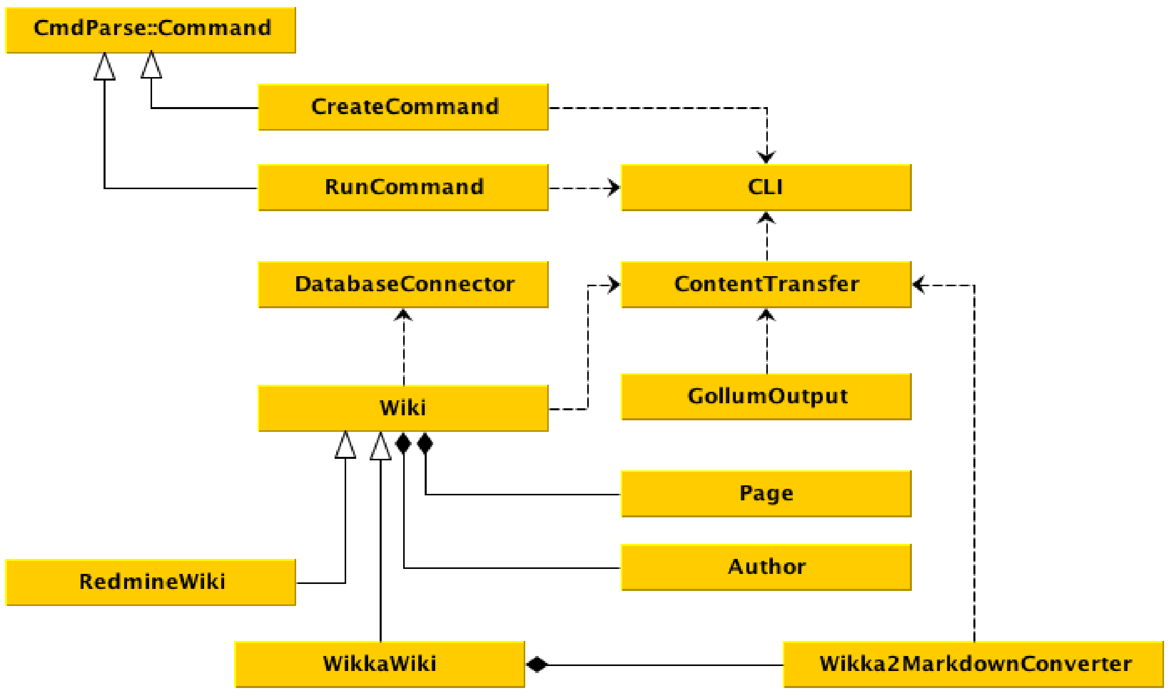

Illustration 1: Basic class diagram of caramelize

The CommandParser-class inherits from the CommandParser of the cmdparse-library and serves as controller. This class parses the commando passed from the command-line and calls the appropriate dispatches with the passed options. In this case there are two main commands for creating a new configuration file and for migrating the data by loading configuration. Data migration is done by the ContentTransferer-class, which controls the general dataflow. About this in a later chapter.

The Models of the MVC-pattern are the storage objects for the data transfer. This being the abstract classes for Wiki, Pages and Authors. A Wiki defines a Wiki-connector that contains lists of all revisions and authors, as well as a few convenience

actions. One page is one revision of an article bearing a title, modification date, Author and change note.
The management of Authors is optional. Gollum attaches the name of an author to the created commit. So if the Wiki-engine stores the author-name together with the revision, this can be used and does not require reading a second list of all authors to get the page-author-association.

Markup conversion is performed by the Wikka2MarkdownConverter, which is a Mixin for the WikkaWiki-connector implementation. Mixins are loose functions, that are added to classes during runtime, many classes can thereby include the same function definition, which reduces redundant code. This Converter includes basically just one function for converting the markup in the given String. About the conversion procedure later. The Markdown-Converter basic interface could be implemented by similar classes.

Connectors

Ruby has no interfaces, however for many models the Adapter-pattern is used. Wiki is abstract class and is extended by the RedmineWiki and WikkaWiki implementation. These describe the basic input connectors. While caramelize supports various input wiki-engines, it exports to one output wiki, namely gollum. Therefore gollum has an own wrapper interface independent of the Wiki-class adapter-pattern.

The adapters for Redmine and WikkaWiki can be regarded as presets, which contain everything to connect to any of those two Wikis. The user only needs to change the database connection logins in the configuration file. For the database connection the mysql2-gem is used.

Since caramelize is to support not just the two predefined wiki-engines, it includes an interface that allows the user to create his own wiki-adapters outside of the main application. Since the configuration file is also Ruby code that is loaded by the application on runtime, own connector-classes can be defined in the configuration. An example implementation of the WikkaWiki-adapter is included in the default configuration as a starting point for users.

Among the options defined within the Wiki-class is the wiki-engine's markup, used to determine whether a markup conversion is required or if gollum already handles the markup.

Data flow

The wiki-adapters return a wiki-object based on a strict interface. At this point it does not matter what the original wiki-engine was, the data are migrated with the same algorithm with little variation depending on some of the options set in the wiki-model. This dataflow is performed in the ContentTransferer-class, see Appendix 1. Since the application follows the MVC-pattern, this class mostly executes methods in the respective models and controls the dataflow.

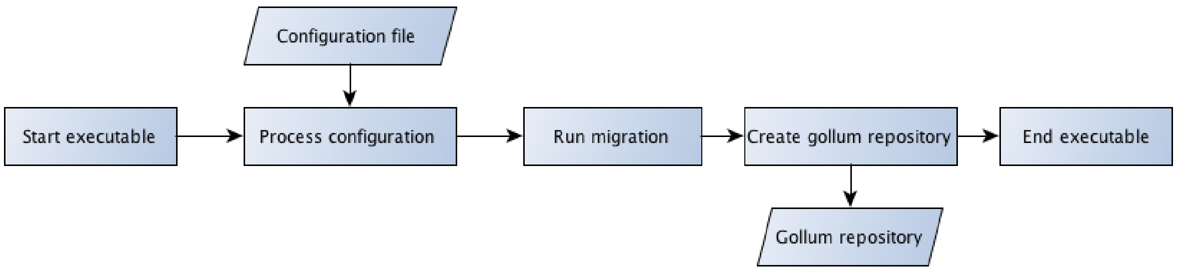

Illustration 2: Flow diagram of caramelize executable

The Wiki connectors return a Wiki-object that contains a collection of all Page revisions and all authors. An instance of the GollumOutputWiki-class initializes the git-repository, that is going to store the document data. Once this is done, the collection of revisions of the input wiki is passed to the output wiki class, that iterates over each page revision and converts each into a git commit. This is automatically performed by gollum, as caramelize only passes the contents and meta data for the actual commit.

The input wiki stores the information which markup the original data used. If it is a markup already supported by gollum, no further markup transformation is required and the migration is already finished.

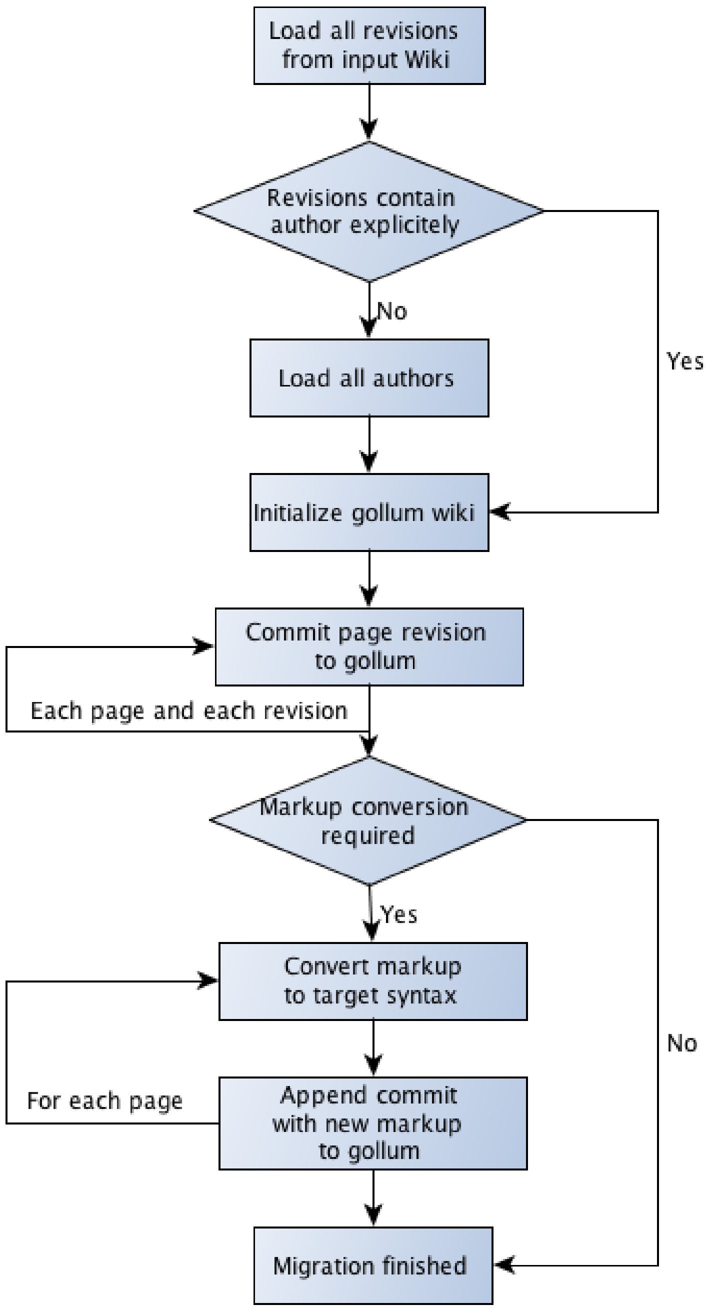
Load all revisions from input Wiki
Revisions contain author explicitely
No
Yes
Load all authors
Initialize gollum wiki
Commit page revision to gollum
Each page and each revision
Markup conversion required
Yes
No
Convert markup to target syntax
For each page
Append commit with new markup to gollum
Migration finished

Illustration 3: Dataflow within the ContentTranserer class

If the markup is not supported, caramelize has to rewrite the markup to markdown syntax to work within gollum. This markup transformation is performed by the Wikka2MarkdownConverter class in the case of the WikkaWiki connector. Markup converters can also be customized in the configuration files. Based on the requirement that the wiki content migration should also be possible for data migration, it is important not to change the page revision posthumously. In case of markup conversion, the conversion is performed creating a new page revision, that is committed to the gollum-wiki-repository. This also has the advantage of avoiding error-handling for broken markup-syntax, which can occur in many saved revisions. By appending the new markup as commits, the user can also see in the changes performed by the commit in git, how the syntax conversion worked and what exactly has been changed. It is thereby part of the active history of the wiki's content data.

Syntax Parsing

The original approach for the markup conversion was to find a library that allows easily and clearly define Parser Expression Grammar to interpret and transform the page document of each article. As described in the chapter Concept and Analysis, research was conducted on available libraries that could be used for this purpose. All three – racc, parlset and treetop – follow the same basic premise of formal language theory and deliver good interfaces to formulate rule patterns. Yacc describes Context-Free Grammar, whilte parslet and treetop formulate Parser Expression Grammar. Without prior experience with this paradigm, however they give little incentive. They include their basic documentation – some more than others – but they don't break with the established paradigm in a way, that may be less powerful in execution, but more rewarding and easier to get into by inexperienced programmers. Based on requirements a solution was needed that would allow novice users to customize their parser-converter. This could not be done with libraries that take 3 weeks and a steep learning curve.

Prototypes were developed to test practicability of different libraries. These prototypes involved parsing of example markup to construct simple base implementations for inline modifiers and line modifiers. Both parslet and treetop leave a positive impression on the overall range of features, however both target a different user group than caramelize does. Both are full grammar-parsers that read and analyze full documents to generate a tree graph. The learning curve for both libraries was very steep and the prototypes did not show the desired progress. These libraries are certainly of great help for developers who are experienced with parser expression grammars and parser construction. They will find very clean interfaces to map abstract grammars into well-readable program code.

However the libraries do not help beginners to learn the premise. They give the tools, but to learn to use them is - despite all documentation and examples given - very hard for inexperienced programmers. Since caramelize has a target user group that may be novice to advanced scripting, they can not be expected to have the experience with syntax parsing to use these libraries and neither can they be expected to learn it for using caramelize.

Parsing Grammar

To correctly parse given strings, a specific and exact set of rules is required. These grammar rules specify the correct definition and interpretation of semantical and formatting properties within the document. The purpose is to analyze and translate source text into structured data.

Yacc and racc feature a deterministic *Context-Free Grammar* short CFG, which is a concept that has been used since the 70s. Parslet and treetop follow a newer paradigm of creating *Parsing Expression Grammar* short PEG. This has been developed by 2004 to be closer to human pattern recognition. [3][9]

These concepts are very powerful, but also more memory-consuming. Comparing them with Regular Expression fails as both have different functions and serve as tools for different means.

Alternate approach

Originally the plan was to include one reference implementation of the WikkaWiki markup, which could then be taken as a starting-point for the user's own wiki-syntax-parser. Given the complexity of the subject it can be debated how helpful this would have been. Especially since the parser-creation includes a lot of trial and error and debugging self-written parsers would turn out hard and frustrating for the user very quickly.

The turnout of this evaluations required to go back to the requirements and to find a more basic approach, which reduces complexity and is more practicable for inexperienced users. The alternative solution chosen were Regular Expressions, to seek and replace existing markup with the new markup.

```
#h1
document.gsub!(/(======)(.*?)(======)/ ) {|s| '# ' + $2 }
#h2
document.gsub!(/(=====)(.*?)(=====)/) {|s| '## ' + $2 }
#h3
document.gsub!(/(====)(.*?)(====)/) {|s| '### ' + $2 }
#h4
document.gsub!(/(===)(.*?)(===)/) {|s| '#### ' + $2 }

#bold
document.gsub!(/(\*\*)(.*?)(\*\*)/) {|s| '**' + $2 + '**' }
#italic
document.gsub!(/(\/\/)(.*?)(\/\/)/) {|s| '_' + $2 + '_' }
#underline
document.gsub!(/(__)(.*?)(__)/) {|s| '<u>' + $2 + '</u>'}

#list
document.gsub!(/(.*?)(\n\t-)(.*?)/) {|s| $1 + '\n' + $3 }
```

Code 1: Selected string substitutions of markup using Regular Expressions

The markdown-converter class analyzes the given document string. Various regular expressions find inline and line modifiers and substitute them to markdown markup.

Command structure & usage

Caramelize is conceived solely as a tool for the command-line. It is installed as a Gem using RubyGems.

```
$ gem install caramelize
```

Once installed, the user navigates to a folder of his choice, where he wants to create the new Gollum-Wiki-repository. He executes 'caramelize create' to create a new template configuration file, which he can edit to his needs. He can either use preset Wiki-connectors in which case he only needs to define his database authentication data. If his wiki is not supported, he can write a customized wiki-connector.

Once finished, the migration can be started with the command “caramelize migrate”. The execution will begin, read the data from the defined wiki and transfer it to the newly created gollum-wiki. Once finished, the user can change directory into the new repository – its default path is 'Wiki.git' and execute 'gollum' to create a local server to browse the data.

```
caramelize <command> [options]
\________/ \_______/ \_______/
     |         |         |
     |         |         \- Depending on the chosen Command different
     |         |            options can be activated. Among them:
     |         |            -v, --verbose: Verbosity to display detailed
     |         |            progress output
     |         |            -q, --quiet: Quiet to suppress any output
     |         |            --config: Optional, to pass the path to a
     |         |            specific configuration file. If none is
     |         |            given, it defaults to preset locations.
     |         |
     |         \- Command to execute in the software.
     |            - create: Will create a template configuration file
     |            - run: Will execute the content conversion based
     |            on a given configuration
     |            - help: Will display the options and description of
     |            this software
     |            - version: Displays the version and debug information
     |
     \- Caramelize software
```

Text 2: caramelize commands and options

```
$ caramelize create [--config=config.rb]
```

Creates a template configuration file. This includes documentation on how to use the preset Wiki-connectors and how to write addition customized connectors.

```
$ caramelize run [--config=config.rb]
```

Will execute a wiki migration based on a found configuration file. If no configuration path is given the script looks in predefined locations for possible configuration files.

```
$ caramelize help
```

Returns help information about the commands and options.

```
$ caramelize version
```

Returns version and release information.

Result

Project evaluation

The first version was released in August 2011. This version contained the basic functionality and the features described in this paper. The software is released as Open Source with MIT License.

The software development of the actual software described in this paper went quite straight-forward without major problems. However this only because of the library evaluation described in this paper. In case of the project framework for building the actual gem library, this avoided many problems that could have come later concerning compatibility with Ruby versions and project deployment.
In case of the markup parsing library building several prototypes helped to evaluate each library and to develop a feel for each differences, strengths and in the end their short-comings compared to my requirements. These libraries included a learning curve concerning the theoretical approach and the practical implementation of Parser Expression Grammar. Ultimately this proofed impractical for this project, as it consumed resources for this implementation, while not fulfilling a major requirement, that of being easy to adopt and to modify. In this regard it was important to take a step back, review the original requirements and decide on a simpler, but effective solution.

The project itself has not a definite end and can be continued in the future, should the need for more functionality arise. It works, but it is open for extension.

Development potential

While all planed features for the first version are in place, there is a list of features, that were considered nice-to-have and did not make it into the first build. During the development some new ideas came up while others were not possible to do for the first release. They can be considered as reference for future development.

Very obvious to extend is the Wiki-connectors interface by creating additional connectors for more Wiki engines. Any customized wiki configuration could be turned into a preset Wiki-connector very easily, so this is also a point where interested third-party developers could join with ease. Similarly, the existing syntax converter from WikkaWiki to Markdown could be taken as a base for other markup converters. Specially conversion to textile could become very handy to convert more markup information than the common markups markdown supports.

While excluded configuration and connector definitions are already possible, its handling is not yet perfect and can become rather complex and with lots of error-potential for novice users. As soon as customized connectors are required, the user needs to program Ruby. This can not be avoided, but the interfaces given may not help enough to avoid errors. In this regard caramelize could also improve on the debugging-possibilities of configuration files. Other Ruby-tools have a very clean and defined way of handling such so called recipes. Caramelize is not yet up to this standards.

The command structure of caramelize in the command-line is clean and simple, which is good to keep. However some more options for specialized workflows could work well. Practice will determine what additional options should become helpful.

As for the project in general. Something considered too late during its development and not integrated is a proper testing of classes and dataflow. Caramelize is not an overly complex project, so testing was not necessary, but would have been good practice as the Ruby community in general has a very good convention and strong tools for testing.

Summary

Data migration is always an important topic and finding ways to facilitate this tasks is a rich field for innovation using existing components and libraries. This project showed how with a few components an extendable data converter for Wiki engines can be written and what emphasize such a software requires. There are simpler ways in terms of basic migration scripts and there can be much more elaborate solutions for parsing and transformation of markup syntaxes. The important thing is that it is easy to understand for people who do not have build conversion scripts regularly and give them a tool to get started. Such data migrations occur seldom and it is something that is generally avoided. This software should invite to create ways to keep your data across platforms and to switch Wiki engines more often to adopt to changing requirements to the software. Software is supposed to be a tool and if it does not fulfill the needs, it should be possible to switch easily instead of being locked into by bad software design. The software proposed and implemented in this paper is one way to help in this regard.

Attachments

Appendix 1: ContentTransferer.rb

```
#Encoding: UTF-8

require 'gollum'
require 'grit'

module Caramelize
  autoload :Wiki, 'caramelize/wiki'
  autoload :WikkaWiki, 'caramelize/wikkawiki'
  autoload :RedmineWiki, 'caramelize/redmine_wiki'
  autoload :GollumOutput, 'caramelize/gollum_output'
  autoload :Wikka2MarkdownConverter, 'wikka2markdown_converter'
  autoload :Author, 'caramelize/author'
  autoload :Page, 'caramelize/page'

  # Controller for the content migration
  class ContentTransferer

    # Execute the content migration
    def self.execute original_wiki, options={}

      # read page revisions from wiki
      # store page revisions

      original_wiki.read_authors
      @revisions = original_wiki.read_pages

      # initiate new wiki
      output_wiki = GollumOutput.new('wiki.git')

      # commit page revisions to new wiki
      output_wiki.commit_history @revisions

      # if wiki needs to convert sytax, do so
      if original_wiki.convert_syntax?
        puts "latest revisions:"
        # take each latest revision
        for rev in original_wiki.latest_revisions
          puts "Updated syntax: #{rev.title} #{rev.time}"
          # parse markup & convert to new syntax
          body_new = original_wiki.convert2markdown rev.body
          unless body_new == rev.body
            rev.body = body_new
            rev.author_name = "Caramelize"
            rev.time = Time.now
            rev.author = nil

            # commit as latest page revision
            output_wiki.commit_revision rev
          end
        end
      end
    end
  end
end
```

Appendix 2: config.rb

```
require 'caramelize/wiki'
require 'caramelize/wikkawiki'
require 'caramelize/redmine_wiki'

# Within this method you can define your own Wiki-Connectors to Wikis not
supported by default in this software

# Note, if you want to activate this, you need to uncomment the line
below.
def customized_wiki

  # This example is a reimplementation of the WikkaWiki-Connector.
  # To connect to WikkaWiki, I suggest to use the predefined Connector
below.
  wiki = Caramelize::Wiki.new(:host => "localhost", :username =>
"user", :database => "database_name", :password => 'admin_gnihihihi',
:syntax => :wikka)
  wiki.instance_eval do
    def read_pages
      sql = "SELECT id, tag, body, time, latest, user, note FROM
wikka_pages ORDER BY time;"
      @revisions, @titles = [], []
      results = database.query(sql)
      results.each do |row|
        @titles << row["tag"]
        author = @authors[row["user"]]
        page = Page.new({:id => row["id"],
                            :title =>   row["tag"],
                            :body =>    row["body"],
                            :syntax =>  'wikka',
                            :latest =>  row["latest"] == "Y",
                            :time =>    row["time"],
                            :message => row["note"],
                            :author =>  author,
                            :author_name => row["user"]})
        @revisions << page
      end
      @titles.uniq!
      @revisions

    end
  end

  wiki
end


# if you want to use one of the preset Wiki-Connectors uncomment the
connector
# and edit the database logins accordingly.
def predefined_wiki

  # For connection to a WikkaWiki-Database use this Connector
  #return Caramelize::WikkaWiki.new(:host => "localhost", :username =>
"root", :database => "wikka")

  # For connection to a Redmine-Database use this Connector
  return Caramelize::RedmineWiki.new(:host => "localhost", :username =>
"root", :database => "redmine_development")
end
```

```
def input_wiki

  # comment and uncomment to easily switch between predefined and
costumized Wiki-connectors.
  #return customized_wiki

  return predefined_wiki

end
```

Appendix 3: Markup modifiers

Inline modifier:

```
Text with **strong** and _italic_ emphasis.
```

Text 3: Inline modifiers for strong and italic markup

Line modifier:

```
![Alternative text](/path/to/image.jpg)
```

Text 4: Image modifier

Multi-line modifier:

```
> Donec sit amet nisl. Aliquam semper ipsum sit amet velit.
> Suspendisse id sem consectetuer libero luctus adipiscing.
```

Text 5: Quotation block modifier

```
- Donec sit amet nisl. Aliquam semper ipsum sit amet velit.
- Suspendisse id sem consectetuer libero luctus adipiscing.
```

Text 6: List block modifier

Sources

1. Ward Cunningham: “What Is Wiki”, 1995, http://www.wiki.org/wiki.cgi?WhatIsWiki
2. Ward Cunningham: “The Wiki Way: Quick Collaboration on the Web”, Addison-Wesley Professional, 2001
3. Ronald Mak: “Writing Compilers and Interpreters”, Second Edition 1996
4. Linux Kernel Development team: “Git Frequently Asked Questions”, 2011, https://git.wiki.kernel.org/index.php/GitFaq
5. Official git website: “About git”, 2011, http://git-scm.com/about
6. Bruce Stewart: “An Interview with the Creator of Ruby”, 2001, http://linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html
7. Yukihiro Matsumoto: “Ruby in a nutshell”, O'Reilly Media, 2001
8. Tim Weber: “Jedem seine Wikipedia”, 2009, http://scytale.name/blog/2009/11/jedem-seine-wikipedia
9. Bryan Ford: “Parsing expression grammars: a recognition-based syntactic foundation”, ACM, 2004