Code Review

Wikibase

Tobias Schlitt

1 Executive Summary

The quality of the Wikibase source code is generally better than in the average PHP project. Especially the derived projects Ask, DataValues and Diff consist of well crafted code.

The most pressing identified issues in the PHP code consist of a too high code complexity of multiple code entities. This leads to a decreased maintainability, because complex code is harder to understand by humans and code pieces cannot easily be replaced. As a result, the implementation of new features and the fixing of errors in the affected code take more time and are more error prone. In addition, implementation of automated tests is harder.

For these reasons, a re-factoring of the corresponding code artifacts is recommended. Approaches to achieve this were elaborated as a part of the code review and are discussed within this report.

The mentioned issues occur more frequently and to a larger degree in the Wikibase extension itself. They are almost absent in the derived extensions.

The overall project infrastructure of the Wikibase project is really good. The utilization of a mixture of unit, integration and front-end tests is commendable. This also applies to the development process, especially to the realized code review process. The source code is extraordinarily well documented. With regard to the test-mixture those observations are true, for the PHP as well as the JavaScript part of the project. Currently the JavaScript implementations only utilize Unit-Testing.

Many of the identified JavaScript problems arise from the usage of jQuery-UI. A lot of functionality needs to be recreated, which can already be found in other modern frameworks. For the needs of the Wikibase project, AngularJS[1] is a good fit.

1 http://angularjs.org

• Table of Contents

2 Introduction

The basis for this report was a code review of the Wikibase[2] extension, the code basis behind the Wikidata[3] project, commissioned by Wikimedia Deutschland - Gesellschaft zur Förderung Freien Wissens e.V. The review was mainly conducted by Qafoo employee Tobias Schlitt, supported by Jakob Westhoff for JavaScript expertise.

In this chapter, the project under review and the environment are introduced. The next chapter summarizes the results of the review without going into further detail yet. After that, the approaches to fix some of the most pressing identified issues are elaborated, which have been worked out by the reviewer together with the Wikibase development team. Finally, the findings of this report are concluded with a set of actionable items.

The subsequent appendix reflects the original review notes.

2.1 Environment

Subject of this code review is the Wikibase extension for the MediaWiki[4] software, which is the code basis for the Wikidata[5] project that is a sister project of Wikipedia[6]. Since Wikibase spun off the extensions Ask, DataValues and Diff, which are still used in the Wikibase project, these extensions were also included into the review.

The following Git revisions of the projects were included in the review (linking to Gerrit for browsing the revisions on the web):

- Wikibase: 86556a7eedd115c3781e05c3ce013b6ea14d2b6c (April 5th 2013)
- Ask: fcac41b029f2716b41340ebf123ee38ac102ca3c (April 7th 2013)
- DataValues: 5a9f4e26ca8cecbff138831c7fe2dde98d764abb (April 10th 2013)
- Diff: fe3441d7481f47981d2ed8c4bc28af3bbdfccec6 (April 3rd)

The scope of the review was to analyze the maintainability, extensibility and complexity of the project code as well as to investigate the test base. Furthermore, the review did not focus on pure analysis and the creation of this report, but on knowledge transfer to the team and elaboration of solution

2 https://www.mediawiki.org/wiki/Extension:Wikibase
3 https://meta.wikimedia.org/wiki/Wikidata
4 https://www.mediawiki.org/wiki/MediaWiki
5 https://www.wikidata.org
6 https://en.wikipedia.org/wiki/Main_Page

approaches.

The main focus of the review was put on PHP but a small fraction of the review time was spent on the JavaScript code base. As a result, the JavaScript remarks in this report are more superficial and there is no dedicated section about solution approaches.

2.2 Project Overview

The Wikibase team utilizes Scrum as its development mode, together with different processes and tools which are roughly described in this section.

The team realizes a code review process to ensure code quality and transfer knowledge among the team members. Every change set needs to pass this process before being merged into the Git master branch. There are no dedicated feature branches in Git. A reviewer merges the change set into master if the review was successful. Gerrit[7] is used as a tool for code review, mainly because it is also used for MediaWiki development itself and is provided by the Wikimedia Foundation (WMF) infrastructure team. Gerrit also triggers unit test execution for each change set in Jenkins.

The team claims to have typically small change sets, which are each connected to one or more issues in their tracker. As an issue tracker, Bugzilla[8] is in place.

Jenkins[9] is in place as a continuous integration (CI) system for Wikibase. The Jenkins server automatically integrates the project against the MediaWiki master branch. The CI process includes, beside unit test execution, Selenium[10] front end tests and JavaScript unit tests, but no software metrics. The Selenium tests are formulated using PageObjects[11] in Ruby. For JavaScript, the QUnit[12] framework is utilized. In order to integrate the QUnit results into Jenkins, the test suite is executed through Selenium.

For PHP unit and integration tests, the PHPUnit[13] framework is used. The team does not obey to the Test Driven Development approach, tests are mostly written after the production code. In some cases, there are no tests developed at all.

3 Analysis

This chapter summarizes the findings of the code review and concentrates on the most essential issues discovered.

7 https://code.google.com/p/gerrit/
8 http://www.bugzilla.org/
9 http://jenkins-ci.org/
10 http://docs.seleniumhq.org/
11 https://code.google.com/p/selenium/wiki/PageObjects
12 http://qunitjs.com/
13 http://qunitjs.com/

3.1 PHP

The PHP code of the Wikibase extension and its related extensions were performed in greater detail than the JavaScript review. In addition, a large portion of the review time was spent to elaborate on potential solutions for the identified issues which are presented in the next chapter.

3.1.1 Modus Procedendi

The PHP analysis was performed with calculated software metrics as starting point. On that basis, the Wikibase source code was investigated manually, following the code flow and hints by the Wikibase development team. After an overview investigation, the team was presented with the intermediate results and the subsequent strategy for the review was discussed: Some additional code pieces were investigated, but most of the time was spent on elaborating on solution approaches.

The initial review was guided by the following software metrics: The Executable Lines of Code (ELOC) measure was used to identify the classes with the largest amount of code, which are suspected to contain large portions of the application logic. Using the NPath Complexity[14] metric, the most complex methods were identified for investigation. Finally, to identify the classes with the highest impact on the project, the CodeRank[15] metric was used. All metrics were calculated using PDepend[16] with its default settings.

For each metric, the top three code entities were analyzed for the Wikibase extension source code. For the smaller extensions Ask, DataValues and Diff, smaller fractions of the top lists were investigated.

The resulting initial analysis was sent to the customer in raw format as a basis for discussing the further proceeding. Besides smaller investigation request, the focus was put on elaboration of solution approaches, which are described in chapter 4.

3.1.2 Summary

The subjective impression of the Wikibase code is better than the average PHP code base. Especially the derived extensions Ask, Diff and DataValues consist of well-structured object-oriented code with very few issues. The Wikibase extension itself appears to suffer especially from the restrictions imposed by MediaWiki extension points. Overall, the code appears to be really well documented.

The review of software metrics showed that more than 50 classes in the Wikibase extension consist of more that 100 ELOC. The classes

14 https://dl.acm.org/citation.cfm?id=42379
15 https://dl.acm.org/citation.cfm?id=1129468
16 http://pdepend.org/

`Wikibase\RepoHooks`, `Wikibase\DispatchChanges` and `ChangeHandler` are examples for this evidence. Such high numbers of code lines are generally harder to maintain as smaller classes, even though much code does not directly imply complexity.

However, a number of methods with severe NPath complexity measures exists. Most significantly, the method `Wikibase\Api\EditEntity::modifyEntity()` shows a complexity value of 17,336,096. Even the complexity of the top 20 methods decreases rapidly, all of these hold an NPath value above 1,000. It is highly advised to re-factor the classes with such a high complexity value in order to reduce the change risk and simplify automated testing.

It should be noted that, due to its sensitivity, the NPath complexity metric also turned up some false positives, especially in the DataValues extension. For example, the constructors of the `DataValues\TimeValue` and `DataValues\GeoCoordinateValue` classes appear with high metric values, while they only consist of easy to understand parameter validation code. There is no need to re-factor these occurrences.

The CodeRank metric identified the classes `Wikibase\Api\ApiWikibase`, `SpecialWikibasePage` and `ViewEntityAction` as the most essential ones. It is important to keep these classes stable in the short term, because there is a high impact for them to break the complete project. Extensive testing is also highly recommended for these classes. In the long run, it is recommended to re- factor the code in order to resolve the impact of the named base classes.

Manual investigation starting at the mentioned code entities lead to the identification of three essential problems that occur frequently across the code base:

- static scoping and missing object life cycle control
- violation of the Single Responsibility principle and
- usage of inheritance for code re-use.

The subsequent review of the Ask, DataValues and Diff extensions shows that their code is generally in a better state, but still occasionally suffers from the named issues.

In the following, you will find an elaboration on each of these issues. In addition, misc smaller issues that occurred during the code review and are worth mentioning in this report will be discussed.

3.1.3 Static Scoping & Object Life Cycle

The Wikibase code contains 160 static methods. Aside from alternative constructors like `newFromArray()`, a large number of methods contain logic and some deal as the main entrance point for the application code. The hook methods - which are registered at the MediaWiki application in order to trigger the execution of extension code as certain events occur - are among the latter

ones (e.g. `Wikibase\RepoHooks`). Manual investigation showed that there is quite some logic realized inside those. This code makes, due to its nature, again use of static dependencies in order to retrieve needed objects and to dispatch tasks.

Static code is generally discouraged for multiple reasons: Statically scoped variables, as well as the use of global variables, can easily lead to side effects in the applications which cannot be easily detected. Furthermore, logic realized in static methods cannot be replaced fine-grained in the application, which is an essential feature of object oriented programming compared to other paradigms. These are also the reasons why code that makes use of static entities is also very hard to test in a sensible way because dependencies cannot be replaced.

A similar issue arises from the other extension points provided by MediaWiki and used by the examined extensions: API extensions and special pages are registered through a string class name. In order to execute the corresponding code, MediaWiki instantiates these classes in a unified way. This way, the extensions do not have control over the life cycle[17] of these objects, preventing them to injection additional dependencies, which leaves the static access of utilized code as the only alternative.

One occasion of a factory-like structure that prevents control over the object life cycle for the developer came to attention in the Wikibase source code: The `ValueParsers\ValueParserFactory` class in the DataValues extension utilizes class names to register value parsers. It is highly recommended to refactor such occurrences to make use of a proper abstract factory[18] approach.

Missing facilities to inject custom dependencies into the executed code is assumed to be a main reason for other problems discussed in this report.

3.1.4 Single Responsibility Principle Violations

Large numbers of executable code and highly complex methods are often a sign for the missing separation of concerns or violations of the Single Responsibility Principle (SRP).

A prototypical example for this is the `modifyEntity()` method of the `Wikibase\Api\EditEntity` class. The method consists of over 250 lines of code and has the highest complexity throughout the project. In addition to that, it also dispatches some work to other methods on the same or parent classes. The method performs at least the following tasks:

- validation of input data
- dispatching of several actions depending on the input data

17 http://qafoo.com/blog/020_object_lifecycle_control.html
18 https://en.wikipedia.org/wiki/Abstract_factory_pattern

- realization of each of these possible actions
- tracking of execution status information
- collection and preparation of output data and
- error handling.

As a result of this complexity and the amount of performed tasks, it is hardly possible for a human being to fully understand the method and almost impossible to create extensive automated tests. Both factors together result in an immense risk that any change in this code results in an error.

While this is the most complex piece of code in the project, there are several other methods that suffer from similar issues on a different level. Another illustrative example is the `SpecialWikibaseQueryPage` class, which provides the basis for some MediaWiki extensions that display a special page to query information from the Wikibase extension. The class defines two template methods for actually performing a query (`getResult()`) and for formatting a result row (`formatRow()`). This already indicates, that at least two responsibilities are mixed in the class:

- business logic and
- output formatting.

Injection of dependencies would solve the SRP violation here, but is hardly possible due to the code structure at the moment. These circumstances interfere with a correct separation of concerns, since there is no easy way to access other objects in order to dispatch execution to them.

3.1.5 Usage of inheritance for code re-use

Many of the inheritance structures in the project seem to be created with the main purpose of re-using code. For example the abstract `Wikibase\Api\ApiWikibase` class extends the MediaWiki base class for API extensions for the purpose of providing helper methods for further derived classes. One derived class is the `Wikibase\Api\ModifyEntity` class. This class implements the main execution method for API extensions, but only to leave another Template Method[19] abstract for further derived classes. In addition, it provides some more protected helper methods for these derived classes. One of these child classes is the `Wikibase\Api\EditEntity` class, which was already subject of discussion due to the high complexity of its `modifyEntity()` method, which is the template method left for implementation by the parent class.

The fundamental issue that arises here is, that the logic implemented in protected methods of the base classes cannot be replaced fine-grained for any of the child classes. If one of these classes requires an adjustment, this cannot be realized by injecting a different dependency into the using code, but only by

19 https://en.wikipedia.org/wiki/Template_method_pattern

touching existing code, which bears the risk of impacting the sibling classes.

This also has a high impact on automated test strategies, since the logic that is encapsulated in protected methods cannot be replaced to isolate the code under test. Instead, all code executed from within a protected method must be treated and tested as if it was in-lined. This implies another raise of complexity for the `modifyEntity()` method.

Furthermore, if inheritance is used for the sole purpose of code re-use, the polymorphic relation between sub-class and parent is not guaranteed. This can easily result in the need to violate the Liskov Substitution Principle[20], which holds a high risk of leading to hard to debug errors.

Even if code re-use by inheritance was often taught best practice in the past, it is discouraged nowadays due to exactly these reasons. Instead, aggregation and delegation are the preferred ways for code re-use. They ensure proper decomposition and allow the fine-grained replacement of delegated logic, while only requiring slightly more code.

It is assumed that the methodology of using inheritance for code re-use is mostly a result of the missing dependency injection facilities behind the MediaWiki extension points. Under such circumstances, moving re-usable code into a common base class appears to be one of very few possibilities.

3.1.6 Tests

A very rough analysis of the Wikibase tests revealed that the test base is quite impressive and of better than average quality. It should be considered to use PHPUnits' mock API[21] or a dedicated mock library like Phake[22] for newly created tests, instead of the currently used hand-crafted mock objects.

There is also some complexity in the test cases which should be considered an indicator for too high complexity in the corresponding production code, which might require re-factoring.

3.1.7 Miscellaneous

The Wikibase project uses namespaces inconsistently since these were originally discouraged by the upstream MediaWiki team. When namespaces were introduced to Wikibase, only a single one (`Wikibase`) was used for the full project. At a latter stage, the team started to split the code base into sub packages using nested namespaces. There is no consequent mapping of namespaces and class names to source code files, yet. It is recommended to introduce consequent namespace scheme. The application of the PSR-0[23] standard to the code base is recommended to enhance the overview for

20 https://en.wikipedia.org/wiki/Liskov_substitution_principle
21 http://www.phpunit.de/manual/3.8/en/test-doubles.html
22 https://github.com/mlively/Phake
23 https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md

developers and simplify autoloading of classes.

There is no consequent use of abstract classes and interfaces throughout the different extensions. It seems that mostly technical reasons were applied to the usage of both. For example, the DataValues extension provides a large number of interfaces for the purpose of defining how a certain set of classes looks like (e.g. `ValueFormatters\ValueFormatter`). In addition, there often exists an abstract base class to realize specific base functionality for the same set of classes that the interface applies to (e.g. `ValueFormatterBase`). It is recommended to revise this usage on basis of the semantical meaning of the constructs as e.g. discussed in our blog article on that matter[24].

Several occurrences of the ternary operator were found during the review. While this operator should generally be avoided for code readability reasons, it might be considered acceptable by the team to use it for the purpose of assigning default values. However, the ternary operator should be avoided for any other purpose. Especially its usage inside conditions, as shown in the following example from `Wikibase\Api\EditEntity`, should be highly discouraged.

```
if ( isset( $data[$props] ) && ( is_object( $page ) ? $page->getId() !==
$data[$props] : true ) )
```

There still exist very few realizations of the Singleton[25] pattern in the code, most probably to allow access to an instance of the corresponding class from the static code already discussed. One example is the `Wikibase\ChangeHandler`. The usage of Singleton produces irreplaceable, static dependencies. It is therefore recommended to get rid of such implementations. It is also recommended to avoid the usage of the Singleton access provided by classes of the MediaWiki framework in order to get rid of the static dependencies implied by the usage. This can greatly enhance testability of the extension code. A possible solution approach to realize this is mentioned in chapter 4.1.

In the DataValues extension, the usage of a result object occurs for the sake of error handling (e.g. `ValueParsers\GeoCoordinateParser`). There is no obvious reason to favor a result object over the use of exceptions in case of an error. It is therefore recommended to switch to using exceptions there, which would remove several conditions from the code resulting in an enhanced readability if the assumption is correct.

The usage of the `assert()` function in production code is generally debatable. The PHP manual actually discourages this.

3.2 JavaScript

Since the Wikibase team asked for advice if a switch to AngularJS[26] is suitable, this topic is picked up where feasible. In the following, the results of the JavaScript review are summarized, after the chosen modality for the review has

24 http://qafoo.com/blog/026_abstract_classes_vs_interfaces.html
25 https://en.wikipedia.org/wiki/Singleton_pattern
26 http://angularjs.org/

been described.

3.2.1 Modus Procedendi

The JavaScript review was bootstrapped by the results of plato[27]. On that basis, the JavaScript code was explored by spot-checks guided by the experience of the code reviewer. In addition to that, the Wikibase team provided a list of questions and created a sketch of an object-model diagram in order to support the reviewer while the review was already in process. This material became available after half the review was conducted. As a follow-up to the actual review, the executive reviewer was available for questions.

3.2.2 Results

In general, the JavaScript code is in a good state. As with the PHP code, occasional violations of the Single Responsibility Principle are one of the most pressing issues. Too much logic is bundled in single modules.

For example, the `wbclient.linkitem.js` plugin and the `jquery.ui.suggester.js` should be split up into multiple entities to enhance readability, maintainability and code re-use. Along the same line, a more strict separation of logic and view should be considered. Where currently there is a lot of inline DOM manipulation performed within the widget logic, a simple template system like Handlebars[28] is recommended for use. Alternatively, the introduction of AngularJS could be considered, which integrates well with jQuery and could deal as template system for the transition phase.

The management of all modules in a global `extension` namespace is discouraged. Instead, a dependency injection mechanism should be implemented. If a switch to AngularJS will take place, the DI mechanism of that framework can and should be used.

Only very few coding style issues can be found throughout the code, which is the result of individual team members making use of JSLint or JSHint. It is recommended to commit a central configuration file for these tools to provide a common basis for all developers and integrate such checks into the continuous integration environment.

As reported by the team, performance issues occur due to the number of re-draws triggered by individual widgets. As a potential solution, the delaying of drawing has been discussed. Such an approach can be handled using events and promises, while each widget can attach the required drawing actions and a collective re-drawing can occur when the corresponding event reaches the top level of the DOM. Separation of read and write operations can enhance the performance in addition.

Unit testing is currently realized using QUnit. Because heavy workarounds have

27 https://github.com/jsoverson/plato
28 http://handlebarsjs.org

to be performed in order to integrate the QUnit results into continuous integration the use of an alternative test runner like js-test-driver[29] or karma-runner[30] (formerly Testacular) can improve the process. These testing tools do not only provide easier integration with many continuous integration solutions, but also superior features like code coverage information and more.

4 Solution Approaches

Due to the interactive nature of the processed PHP code review, solution approaches for several of the identified issues, as well as for several questions asked by the Wikibase team, were discussed and elaborated. The following sections contain a summary of each extensively discussed topic.

Please note that the presented approaches are still in draft stage and do not function as production ready solutions.

4.1 Dependency Injection

The need for a proper dependency injection mechanism was identified as the most pressing issue during the code review. Discussions with the Wikibase team confirmed and underlined this impression. A dependency injection mechanism should help to decompose complex code into multiple classes to delegate parts of the necessary logic to.

This section elaborates on the need for such a mechanism, summarizes the requirements and the proposed solution approach.

4.1.1 Background

Essentially three different approaches for extending MediaWiki are used in the Wikibase extension and its derived extension projects:

- Hooks
- API Modules
- Special Pages

In order to register code points for these extensions, MediaWiki expects either a PHP callback with a certain parameter signature (Hooks) or a string class name. Both approaches prevent the proper usage of dependency injection for different reasons.

In case of class name registration, the extension developer has no control over the object life cycle of the class provided as the extension code point. MediaWiki creates an instance of the class in a unified way, with a unified set of constructor arguments. There is no facility for the developer to provide

29 https://code.google.com/p/js-test-driver/
30 http://karma-runner.github.io/0.8/index.html

additional arguments for the construction process.

The situation is different for the callback mechanism. In this case, the creation of a custom object could be performed by the extension developer and a callback to a public method on that object could be registered. However, one important constraint of MediaWiki extensions which are to be used in the Wikipedia is a very low configuration footprint in terms of memory usage and code execution time. For this reason, and most probably due to established habits, static method calls are registered as callbacks.

Therefore, no dependency injection is performed for these extension points, which serve almost exclusively as entry points for the extension code. It is suspected that this is one of the main reasons behind the found Single Responsibility Principle violations and static code structures, which lead to in-flexible and hard to test code.

4.1.2 Requirements

The desired approach should provide a simple dependency injection mechanism, which takes over the composition of actual objects to be used at application runtime. In addition to that, a bridge is needed to make this approach usable behind the static extension entry points provided by the MediaWiki software. The amount of static code is to be kept as small as possible.

Furthermore, the designed approach must ensure that the memory and code execution footprint of the Wikibase extension is kept low, since the extension is used on the Wikibase server. A lazy initialization approach is therefore recommended.

Discussions with the team showed that a recent issue in the development of Wikibase is the usage of several instances of the same class in parallel with different configurations. The dependency injection mechanism should support this requirement.

4.1.3 Dependency Management

The usage of an existing dependency injection solution is possible and viable. Potential candidates could be the Symfony2 Service Container[31], as an example for a feature-rich solution, Pimple[32], as an example for a more light-weight approach, or any of the other open source PHP implementations. However, it is questionable whether the Wikibase community would be satisfied with such a solution and how it would be adopted. For this reason, a simple, home-made approach was suggested as an alternative. The final decision is left to the Wikibase team.

The proposed approach utilizes a simple `DependencyManager` class that manages

31 http://symfony.com/doc/current/book/service_container.html
32 http://pimple.sensiolabs.org/

dependency resolution in combination with implementations of an abstract `DependencyBuilder` base class. Each of the latter ones realizes the creation of a part of the object graph similar to the Builder pattern[33].

Illustration 1: DependencyManager illustrates a `DependencyManager` with some example builder objects. Each builder is assigned to a specific key by the manager, allowing the user to access the root object of the graph that is created by the builder. If requested, the manager will instruct the corresponding builder to create the object. The manager will then return the created object so that its creation is transparent to the user.

For example, if the a `DependencyManager` as configured in Illustration 1: DependencyManager is asked to retrieve the object with the key “`mail`”, it will ask the registered `DatabaseBuilder` to create that object and return it.

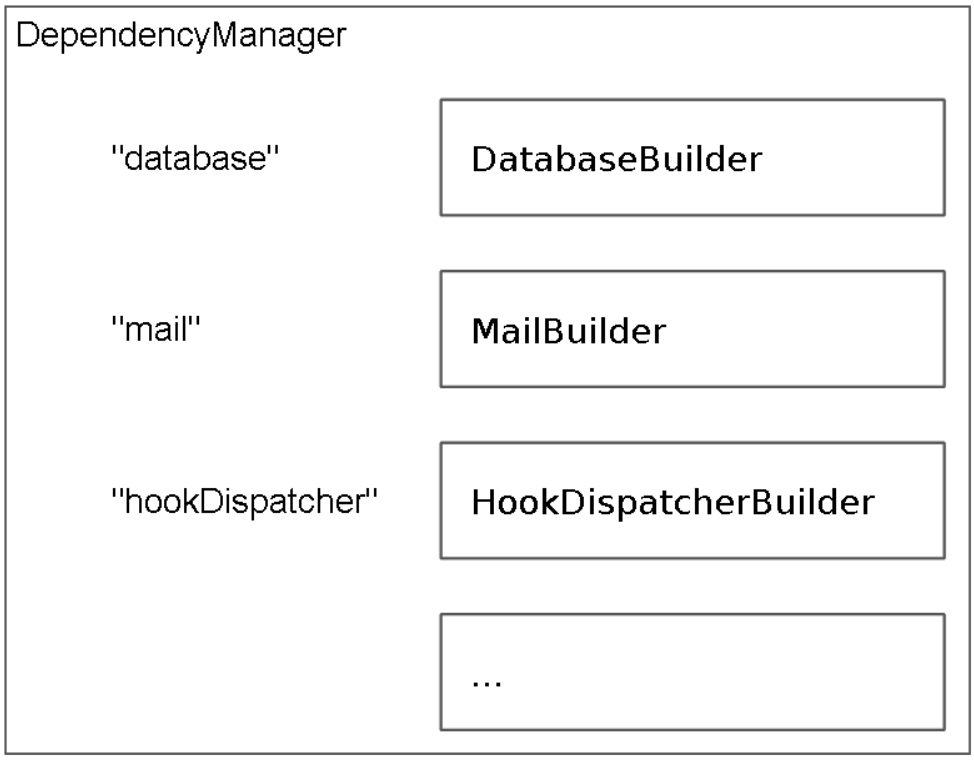

Illustration 1: DependencyManager

The following example code outlines the `DependencyManager` class. The `registerBuilder()` method is used during configuration to register builders for all partial object graphs that can be created at run time. The method `getObject()` triggers the described creation process and returns the created object for a given key.

```
class DependencyManager
{
    public function registerBuilder($objectKey, DependencyBuilder $builder)
    {
        // ... store in lookup ...
```

33 https://en.wikipedia.org/wiki/Builder_pattern

```
    }
    public function getObject($objectKey)
    {
        // ... sanity checks ...
        return $this->builderMap[$objectKey]->buildObject($this);
    }
}
```

A `DependencyBuilder` base class could look as follows:

```
abstract class DependencyBuilder
{
    public abstract function buildObject(DependencyManager $dependencyManager);

}
```

Only the `buildObject()` method is required, which is triggered by the `DependencyManager`. Note that the manager is given to that method in order to allow it to depend on other partial object graphs. That way, a mailer object can retrieve a logger as a dependency. Now the object graph can be created in a fine-grained way and dependencies can easily be exchanged by the registration of different builder objects during configuration.

The sketch for an example builder could look like this:

```
class MailBuilder extends DependencyBuilder
{
    public function __construct($hostname, $port)
    {
        // ...
    }
    public function buildObject(DependencyManager $dependencyManager)
    {
        return new Mail(
            new SmtpTransport($this->hostname, $this->port),
            $dependencyManager->get('templateEngine')
        );
    }
}
```

This particular builder receives configuration parameters through its constructor. The corresponding settings are used in the `buildObject()` method to create an instance of the `SmtpTransport` object. Since no other objects rely on this one, there is no need to outsource it into its own builder. If this should be the case, a new builder could be implemented for the mailer. The case is different for the template engine in this example: Since this one is needed by other objects, it is contained in a dedicated builder, which is requested from the `DependencyManager` to resolve the dependency.

4.1.4 Extension Encapsulation

In addition to the pure dependency injection concept, a solution is required to break out of the static context of the extension points provided by MediaWiki. To achieve this, a workaround is needed which allows to access the configured

instance of the `DependencyManager` in the static context.

On basis of experience from other projects, two mechanisms of self-control and guidance for new developers have been introduced into the solution proposal.

First, the static access method, which is required to bridge the dynamic and static context, should be encapsulated into a dedicated class called e.g. `ExtensionAccess`. This way, it can be clearly documented what the purpose and limits of this mechanism are. Furthermore, it eases the possibility of e.g. a PHP_CodeSniffer[34] rule to restrict the usage of this mechanism to a small number of well specified cases – i.e. the extension points of MediaWiki.

```
class ExtensionAccess
{
    private static $registry;
    public static function setRegistry(ExtensionPointRegistry $registry)
    {
        self::$registry = $registry;
    }
    public static function getRegistry()
    {
        return self::$registry;
    }
}
```

Second, in order to restrict the set of objects directly available to a specific extension point, a class named `ExtensionPointRegistry` is introduced. This registry class exposes a method for each accessible object and hides the `DependencyManager` to enforce a clean usage of the dependency injection mechanism.

```
class ExtensionPointRegistry
{
    private $dependencyManager;
    public function __construct(DependencyManager $dependencyManager)
    {
        // ...
    }
    public function getHookDispatcher()
    {
        return $this->dependencyManager->get('hookDispatcher');
    }
    // ... more entry points from MW extensions ...
}
```

This introduces a second level of indirection in the process of retrieving an object to dispatch the actual logic to, it is highly recommended to keep this mechanism. Its purpose is to avoid one of the most common mistakes when using a dependency injection mechanism: Retrieving the dependency container and pulling required objects from it. The only object allowed to do so should be the `ExtensionPointRegistry`.

34 http://pear.php.net/package/PHP_CodeSniffer

4.1.5 Usage

The described dependency injection mechanism occurs in two different places: 1st an instance of the `DependencyManager` needs to be configured. This configuration includes the creation of the required builder objects and potential injection of configuration variables:

```
$dependencyManager = new DependencyManager();

// Retrieves configuration from config object
$dependencyrMmanager->registerBuilder('database', new DatabaseBuilder());

// Give configuration through constructor
$dependencyManager->registerBuilder(
    'mail',
    new MailBuilder('smtp.example.com', 25)
);
```

The example code shows two variants of how configuration could take place: The registered `DatabaseBuilder` is expected to access a configuration object obtained from the `DependencyManager`. This centralizes the place for configuration to a central management object which could for example parse a configuration file. Such a solution would imply a dedicated builder for each object instance that requires a dedicated configuration.

The second builder registration, for the `MailBuilder`, illustrates how configuration settings are delivered directly to the builder during its construction. This implies mixing the concerns of configuration and object registration, which is typically not desired, but might be feasible here for performance reasons. This variant can also be used with the current way of accessing configuration from a global variable. This is not recommended for the previously shown approach.

The 2nd point of usage occurs at program run time in the extension points this mechanism is created for. However, the `DependencyManager` will never be used directly there, but (as discussed) only through the `ExtensionPointRegistry`, which is obtained via the `ExtensionAccess` mechanism:

```
class RepoHooks
{
    public static function onBeforePageDisplay( \OutputPage &$out, \Skin &$skin
) {
        // Obtain hook dispatcher
        $hookDispatcher = ExtensionAccess::getRegistry()->getHookDispatcher();
        // Dispatch hook to a dedicated object
        return $hookDispatcher->dispatchHook(
            'onBeforePageDisplay',
            $out,
            $skin
        );
    }
    // ...
}
```

4.1.6 Conclusion

The sketched approach fulfills all presented requirements: The dependency injection mechanism is kept simple and easy to understand. The builder infrastructure supports lazy initialization with varying degrees and requires only a minimal processing overhead during configuration. Through the static bridge, it is possible to use the dependency injection mechanism from within the MediaWiki extension points without instantiation of the actual object graph during configuration. Two security mechanisms have been put in place as examples how to prevent abuse of the dependency injection mechanism for pulling arbitrary dependencies.

The following graphic visualizes roughly the code flow when an extension point is triggered by MediaWiki:

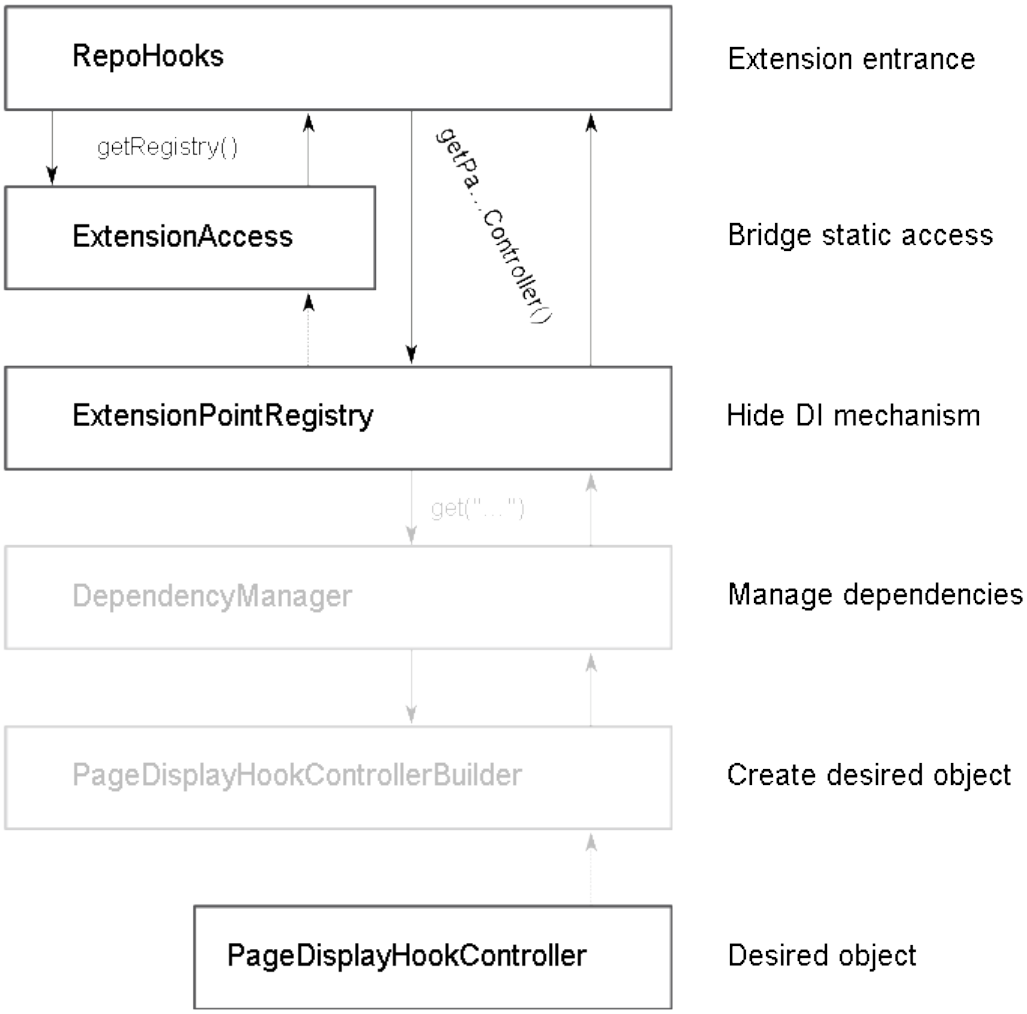

Illustration 2: Dependency injection flow

4.1.7 Remarks

The presented solution approach is not a completely elaborate solution but

should only deal as an inspiration for the further work of the Wikibase team. Especially the class naming must carefully be rethought for intuition. Furthermore, the following remarks should be considered for a final solution:

There is currently no way in the example to share objects between instantiation requests, i.e. each object is created from scratch through `DependencyManager` and the corresponding builder whenever it is requested. A generic decorator for builders could be used to achieve sharing or the sharing mechanism can be implemented into the manager.

If the degree of lazy initialization supported by this mechanism does not fulfill the performance requirements in terms of memory footprint, a solution might be to allow factories to access the dependency injection mechanism. However, this requires additional strict rules for usage of the `DependencyManager` in order to prevent its injection into logic objects and arbitrary pulling of dependencies.

Another possible performance optimization might be to not configure the dependency mechanism on each boot of MediaWiki, but to delay the configuration by outsourcing it into a dedicated file. This file could be imported (`include`/`require`) by a global function when the Wikibase extension is actually activated.

It is highly discouraged to realize these two performance optimizations without evidence of an actual performance issue!

4.2 Controllers

The previously described dependency injection mechanism and its bridge into a static context allow the Wikibase extension code to work with dependency injection behind the static MediaWiki extension points. It is however not recommended to implement logic directly inside the extension code points by pulling several objects from the `DependencyManager`.

Instead, there should be a dedicated controller class or a method on such for each extension point. These controllers encapsulate the glue logic which dispatches the typical web application tasks:

- parsing and validation of input parameters
- triggering of business logic and
- output rendering.

4.3 Parsers & Serializers

By request of the Wikibase team, the parsing and serializing infrastructure of the DataValues component was analyzed.

The serializing infrastructure of the DataValues component uses a valid approach, while there is potential for optimization in respect to testability and

extensibility. The currently implemented parsing infrastructure does not take care of nested value structures, which is foreseen to become a problem in a near future.

While the solutions sketched in this chapter are specifically designed for the DataValues extension, the shown principles can and should be applied to other occurrences of parsing and data serialization.

4.3.1 Analysis

The DataValues component provides functionality to serialize a given value object graph into a plain structure that can be stored ore transferred easily. This structure may consist of scalar values and arrays, but must not contain objects or other complex data types. Furthermore, the component provides parsers for the reverse direction of creating a value object structure from a given plain representation.

The serialization process is bound directly to the value objects by the `DataValues\DataValue` interface, which requires a method `toArray()` to be implemented by each value object. This method returns an array that contains a key “`type`”, which is assigned to a type identifier for the value, and a key “`value`”, which holds the plain representation of the value. In order to transform a nested value, e.g. `DataValues\MultilingualTextValue`, to its plain structure, the `toArray()` code of the containing object must call the `toArray()` method of the nested object.

While this approach works well for now, it might result in code duplication when further serialization formats are required to be implemented. For these, additional methods need to be implemented on all value objects, which basically contain the same logic. Furthermore, for testing the current approach with nested values, there is the need to either mock nested value objects or to create complex data fixtures. Both approaches are sub-optimal.

The parsing of value objects from a plain structure is modeled through dedicated parser classes. The `ValueParsers\ValueParserFactory` is the central point for registration of parsers. While the implementation of this class inhibits issues with object life cycle control (as mentioned in chapter 3.1.3), it provides a mechanism for third parties to register custom parsers.

Each registered parser must implement the `ValueParsers\ValueParser` interface and is responsible for parsing exactly one type of value. Parsing of nested values is not supported by the infrastructure directly because it was not needed, yet. Due to the missing dependency injection facility for parsers, it is not even possible to delegate parsing of nested values to a dedicated parser object by now. Therefore, parsers would need to take care of parsing nested values themselves, which would lead to hardly testable code and potentially to code duplication.

4.3.2 Parser

In order to support the parsing of nested values properly, the following approach for a central parser dispatching point is suggested. The sketched solution resolves the issue of missing object life cycle control for parser developers, too.

As the central point for maintaining parser instances, the original `ParserFactory` is to be replaced by a `ParserDispatcher`, which has the purpose of dispatching parsing of a certain value type to the corresponding parser. In order to avoid all parser instances to be created at configuration time, the `ParserDispatcher` does not aggregate parsers directly, but accepts slim factory objects, which are modeled after the abstract factory[35] pattern. This way, lazy initialization is supported for parsers.

```
class ParserDispatcher
{
    public function __construct(array $parserFactories)
    {
        // ... validate & store ...
    }
    public function parseValue($inputValue)
    {
        foreach ($this->parserFactories as $factory) {
            if ($factory->canParse($inputValue)) {
                $parser = $factory->createParser();
                return $parser->parse($inputValue, $this);
            }
        }
        throw new \RuntimeException('No parser found for input value.');
    }
}
```

In order to parse a value, the `parseValue()` method is utilized. It iterates the available parser factories until one is found that can parse the given `$inputValue`. The determination of which parser factory is to be taken is delegated to the factories themselves, which allows more powerful matching algorithms to be implemented. In addition, it eases testing of the matching algorithm.

Once triggered, the parser receives the dispatcher again, in order to dispatch nested value parsing to it. The following code example shows such a parser:

```
class NestedValueParser extends Parser
{
    public function parse($value, ParserDispatcher $dispatcher)
    {
        // ... parse $value into $result

        $result->someNestedValue =
$dispatcher->parseValue($value['someNestedValue']);
        $result->otherNestedValue =
```

35 https://en.wikipedia.org/wiki/Abstract_factory_pattern

```
$dispatcher->parseValue($value['otherNestedValue']);

        return $result;
    }
}
```

As a result, the parsing of a single value is completely encapsulated in a single parser, which does not need to know about the structure or nature of nested values. In addition to that, 3rd party developers can overwrite the parsing of specific values easily, for example in order to have instances of derived value classes created instead of the default ones. Also, the testing of parsers is eased.

4.3.3 Serializer

While the current serialization approach works, it is recommended to decouple the serialization logic from the value objects in order to separate responsibilities clearly. However, this change is not crucial. The structure for such a serialization infrastructure works very similar to the Parser infrastructure presented in the previous section. Therefore, a detailed elaboration is skipped here.

5 Action Items

Together with the Wikibase development team, three action items have been identified for near-time realization:

1. Implement the dependency injection mechanism following the sketched solution described in chapter 4.1.
2. Refactor the `EditEntity` API module on that basis, slowly decomposing its functionality and verifying the re-usability of components.
3. Refactor the hook implemented by Wikibase using the dependency injection mechanism.

After that, the team can consider further steps or migrate to a soft refactoring of the missing code entities during everyday work. Furthermore, it should be ensured that code entities with a high CodeRank are well-tested and kept stable. For this proceeding, the enhanced CodeRank mode of Pdepend should be used. At this point, a reconsideration by Qafoo is recommended to verify that the taken steps resulted in the desired effects.

6 Appendix

The following appendix contains the raw notes of the code review as requested by the customer. Please note that solution approaches are not elaborated

explicitly again in this section, because there is already an extensive documentation found in chapter 4. In addition, there was quite some oral and chat communication taking place during the review, which is – due to its nature – not documented here.

6.1 PHP

The following notes relate to PHP code only. They are structured by analyzed extension and class / method.

6.1.1 Wikibase

The Wikibase extension was the main focus of the review. Therefore, the biggest amount of time was spent reviewing its code. Software metrics were used to bootstrap the manual review. In following, the review notes of this initial analysis are presented. Please note that these notes do not include detailed elaborations on the solution approaches worked out together with the Wikibase team, since these are already presented in depth in chapter 4.

Wikibase\RepoHooks

Wikibase/repo/Wikibase.hooks.php

This class contains the hooks for extending MediaWiki. The callbacks are realized using static methods.

```
public static function onPageTabs( \SkinTemplate &$sktemplate, array &$links )
{
    wfProfileIn( __METHOD__ );

    $title = $sktemplate->getTitle();
    $request = $sktemplate->getRequest();

    if
( EntityContentFactory::singleton()->isEntityContentModel( $title->getContentMo
del() ) ) {
        unset( $links['views']['edit'] );

        if ( $title->quickUserCan( 'edit', $sktemplate->getUser() ) ) {
            $old = !$sktemplate->isRevisionCurrent()
                && !$request->getCheck( 'diff' );
            $restore = $request->getCheck( 'restore' );

            if ( $old || $restore ) {
                // insert restore tab into views array, at the second position

                $revid = $restore ? $request->getText( 'restore' ) :
$sktemplate->getRevisionId();
```

```
                $head = array_slice( $links['views'], 0, 1);
                $tail = array_slice( $links['views'], 1 );
                $neck['restore'] = array(
                    // … code including method calls …
                );

                $links['views'] = array_merge( $head, $neck, $tail );
            }
        }
    }

    wfProfileOut( __METHOD__ );
    return true;
}
```

As can be seen in this exemplary method from the class, there is quite some logic included in the static method context. Because there is no easy way to break out of the static context, the code needs to use more static accesses and complexity is concentrated in the method.

The hook mechanism should receive an abstraction layer to decouple the logic from the static context of MediaWiki extension points. See chapter 3.1.3 for details.

Wikibase\DispatchChanges

Wikibase/lib/maintenance/dispatchChanges.php

The `DispatchChanges` class implements a maintenance script which populates changes in Wikibase to different Wikipedia instances. Again, there is quite some complex logic involved in the MediaWiki extension point method `execute()`, for example control structures are nested until level 4, while there occur multiple of these control structures in a sequence. The result is a degree of complexity that can not easily be grasped, which results in decreased maintainability. Furthermore, the method dispatches additional logic to protected methods on the same class, with the result of really hard to test code. It is recommended to refactor this class.

Wikibase\ChangeHandler

Wikibase/client/includes/ChangeHandler.php

Beside a larger amount of code which should be refactored into multiple classes in order to reduce complexity, this class implements a Singleton[36] pattern, which is discouraged due to its nature of populating static dependencies. It is supposed that this Singleton is required in order to make an instance of the class available to MediaWiki extension point implementations, which also reside in a static context.
It is recommended to get rid of such constructs. Further details on this issue can be found in chapters 3.1.3 and 4.1.

36 https://en.wikipedia.org/wiki/Singleton_pattern

It has also been noticed that there is quite some logic realized in the constructor of the class, mainly in order to obtain default instances of otherwise injected dependencies, when the Singleton is used.
There is one case, where an object property is set to null: if no dependency was injected.

```
if ( $this->mirrorUpdater !== null && ( $change instanceof EntityChange ) ) {
    // keep local mirror up to date
    $this->mirrorUpdater->handleChange( $change );
}
```

This results in additional necessary checks for this special case around the class, such as

```
if ( $this->mirrorUpdater !== null && ( $change instanceof EntityChange ) ) {
    // keep local mirror up to date
    $this->mirrorUpdater->handleChange( $change );
}
```

For such situations, the usage of a Null Object[37] is recommended to avoid additional complexity for handling special cases.

Wikibase\Api\EditEntity::modifyEntity()

Wikibase/repo/includes/api/EditEntity.php

This method shows the highest NPath complexity in the project, which is 17,336,096. While there exist code entities with way higher complexities in other projects, the shown one already indicates unmaintainable code. It is highly recommended to refactor the method.

The `Wikibase\EditEntity` class implements the MediaWiki API facilities for updating a Wikibase entity. For this purpose it derives from `Wikibase\Api\ModifyEntity`, which provides re-usable code to it through protected methods and a Template Method[38] pattern. The re-use of code through inheritance is generally discouraged in favor or delegation to other objects. For an elaboration in this, please refer to chapter 3.1.5. Refactoring is recommended.

The `modifyEntity()` method itself mixes multiple different responsibilities, like for example:

- Input parameter parsing
- Data validation and sanitation
- Logic dispatching for various input entity types
- Collection of status information

It is highly recommended to refactor the code of this method into multiple

37 https://en.wikipedia.org/wiki/Null_Object_pattern
38 https://en.wikipedia.org/wiki/Template_method_pattern

classes in order to decrease complexity and that way improve extensibility and maintainability.

The overall structure of the class consists of a `foreach` loop, which contains a huge `switch-case` structure to realize logic for different input entity types.

```
foreach ( $data as $props => $list ) {
    switch ($props) {
        case 'labels':
            // … ~20 lines of update logic …
        break;
        case 'descriptions':
            // … ~20 lines of update logic …
        break;
        case 'aliases':
            // … >50 lines of update logic …
        break;
        // … more cases …
    }
}
```

At a first glance, it appears to be a good approach to extract each piece of entity type dependent logic into its dedicated “action” class.

Beside complexity which is generated through disobeying the Single Responsibility Principle, there are smaller coding issues in the method:

The method receives an object parameter by reference:

```
protected function modifyEntity( EntityContent &$entityContent, /*...*/ )
```

Since PHP 5 it is generally not necessary to pass objects by reference in order to work on a reference instead of a copy of the object. Passing an object by reference only allows to overwrite the variable contents completely, which is highly discouraged since it can easily lead to errors that are extremely hard to debug. The removal of all occurrences of such constructs is therefore highly recommended.

While use of the ternary operator is generally discouraged since it decreases code readability, its usage inside of conditions should definitely be avoided. Constructs like

```
if ( isset( $data[$props] ) && ( is_object( $page ) ? $page->getId() !==
$data[$props] : true ) ) {
```

decrease maintainability of control structures significantly. Additionally, many uses of the ternary operator could be replaced by a simple method to choose a default value, resulting in calls like

```
$this->getValueWithDefault( $value, $defaultValue )
```

Several smaller code duplications could be found in the `EditEntity` class, for example the following code piece occurs several times, only with slightly different parameters:

```
$this->dieUsage( $this->msg( 'wikibase-api-illegal-field',
'lastrevid' )->text(), 'illegal-field' );
```

This could be easily moved into a dedicated method with a speaking name to avoid code duplication and increase readability through calls like:

```
$this->dieUsageIllegalField( 'lastrevid' );
```

Wikibase\EntityView::getHtmlForLanguageTerms()

Wikibase/repo/includes/EntityView.php

With an NPath complexity value of 625,004, this method ranks second place in the Wikibase project. The `Wikibase\EntityView` class provides basic code for creating entity views. So, again, there is a case of code re-use through inheritance, which is discouraged in favor of delegation. Chapter 3.1.5 contains a detailed elaboration on this issue.

The affected method `getHtmlForLanguageTerms()` renders HTML output for a Wikibase entity's terms. The method iterates terms by language and renders a list of terms in form of a table, which is responsibility of the view. While a simple template engine seems to be used through the `wfTemplate()` function, the main view logic is kept on the PHP side.

The complexity of the method mainly originates from the determination of default values, which could easily be replaced by a speaking method call like

```
$this->getValueWithDefault( $value, $defaultValue )
```

which would reduce the complexity of the method and increase readability. Furthermore, the rendering logic for the table could be extracted into dedicated classes or a more powerful template engine could be used. Even moving the code into a dedicated PHP file could help to divide the view logic more explicitly from the business logic.

The same recommendation basically applies to most other methods in the `Wikibase\EntityView` class.

Wikibase\MultiLangConstraintDetector::addConstraintChecks

Wikibase/repo/includes/MultiLangConstraintDetector.php

This method allows to extend the Wikibase functionality through the standard MediaWiki hook mechanism. While it is understandable, that standard hook mechanisms must be exposed by the Wikibase extension, it is recommended to abstract the hook mechanism into its own class in order to avoid code duplication and create a clear border to extension points.

This step could also increase readability of the code significantly, if each hook point provided by the Wikibase extension is put into a dedicated method with a speaking name. By now, there is hardly any documentation on the extension facilities provided by Wikibase and the code is inexplicit on what data is submitted to the hooks and what manipulations are expected. Optimally, a specific set of objects is submitted, on which hooks can work.

With an NPath complexity of 314,000, the method ranks place three in

Wikibase. It was not possible to fully understand the code of the method within a reasonable time, which is another strong indicator that the method needs refactoring. From what has been understood, the introduction of a `Constraint` base class to encapsulate checks and provide extension facilities to 3rd parties appears reasonable.

Discussion with the team showed that the affected class is considered for a complete re-building, which can be supported.

Wikibase\Api\ApiWikibase

Wikibase/repo/includes/api/ApiWikibase.php

Following the CodeRank metric, this class is the most essential one in the Wikibase project. This appears logical and was confirmed by the team, because it provides the base for all API modules provided by the Wikibase extension. It is highly recommended to have extensive tests for this class and to minimize changes to it. Errors occurring in this class have a very high impact on the complete project and therefore the full application.

The purpose of the `Wikibase\Api\ApiWikibase` class is to provide re-usable pieces of code to the deriving MediaWiki API modules. So, this is another occurrence of code re-use by inheritance, which is discouraged. Please refer to chapter 3.1.5 for details on this issue.

It is highly recommended to resolve the high impact of this class by changing the way API modules are handled.

There is code in this class to store information about operation results and to display Wikibase entities. As it seems, the latter functionality was already partly extracted into a `Wikibase\EntitySerializer` class. A refactoring in this direction is recommended, but the currently taken approach for this can be optimized. Further elaboration on this can be found in chapter 4.3.

For refactoring such code, the following procedure is recommended:

- Create (integration) tests for the module the functionality resides in
- Create new API
- Implement tests for new API
- Move the code into the new API (guided by tests)
- Replace the source in its old place with calls to the new API

Wikibase\SpecialWikibasePage

Wikibase/lib/includes/specials/SpecialWikibasePage.php

Ranking second place regarding the CodeRank metric in the Wikibase project, this class again provides the basis for implementations of MediaWiki extension points: The so-called “Special Pages”. Again, this class provides code for re-use for derived classes, which is discussed extensively in chapter 3.1.5. The issues applying here are almost identical to the ones discussed for the previous section.

Investigation of this class lead to one of its derivatives, the `Wikibase\SpecialWikibaseQueryPage` class, which provides specialized functionality for query pages. The class shows a prototypical violation of the Single Responsibility Principle by taking care of parsing input parameters as well as performing the actual query and rendering the results.
It is recommended to refactor these concerns into dedicated classes and make use of delegation instead of providing the functionality by inheritance.

ViewEntityAction

Wikibase/repo/includes/actions/ViewEntityAction.php

Ranking third place in the CodeRank metric for Wikibase, the issues in this class are basically the same as described for `Wikibase\Api\ApiWikibase` and `Wikibase\SpecialWikibasePage`. The recommendations provided there basically apply here, too, regarding code re-use by inheritance (chapter 3.1.5) and missing dependency injection facilities (chapter 4.1).

6.1.2 DataValues

The DataValues extension is the largest extension of the extensions derived from Wikibase. Since the review focus was clearly on the Wikibase extension and there exist two more derived extensions, the DataValues extension was only reviewed very shortly.

ValueParsers\GeoCoordinateParser

DataValues/ValueParsers/includes/parsers/GeoCoordinateParser.php

173 lines of executable code appear to be a bit too much, which already gives a hint to the slight violation of the Single Responsibility Principle in this class. However, the metric indicates a far better state than the Wikibase extension itself.

The class represents a parser for different representations of geographical coordinates. It realizes the different parsing algorithms each in a dedicated protected method, which are dispatched from a central protected method:

```
protected function getParsedCoordinate( $notationType, $coordinate ) {
    $coordinate = $this->resolveDirection( $coordinate );

    switch ( $notationType ) {
        case self::TYPE_FLOAT:
            return (float)$coordinate;
        case self::TYPE_DD:
            return $this->parseDDCoordinate( $coordinate );
        case self::TYPE_DM:
            return $this->parseDMCoordinate( $coordinate );
        case self::TYPE_DMS:
            return $this->parseDMSCoordinate( $coordinate );
        default:
            // @codeCoverageIgnoreStart
```

```
            throw new InvalidArgumentException( 'Invalid coordinate type
specified' );
            // @codeCoverageIgnoreEnd
    }
}
```

This code complicates extension of the parser infrastructure by new representations for geographical coordinates. Furthermore, it increases the difficulty to unit test this class. It is therefore recommended to refactor each of the parsing algorithms into a dedicated class and delegate parsing to these. This approach also splits up responsibilities nicely. A similar approach as described in chapter 4.3 can be used.

It is unclear, why a result object is used instead of throwing an exception, in case the provided coordinate representation cannot be parsed. Using exceptions could simplify the code in this class and in the using code, which raises the readability.

ValueParsers\ApiParseValue

DataValues/ValueParsers/includes/api/ApiParseValue.php

With 91 executable lines of code, this class is ranked second in the DataValues module. This can already be considered a good code size. The inspected class provides a MediaWiki API module for the value parsers shipped with the extension.
The class already utilizes an approach slightly similar to a dependency injection mechanism, which is good. However, if the solution elaborated in chapter 4.1 is implemented, it could be used here, too.

ValueValidators\DimensionValidator

DataValues/ValueValidators/includes/validators/DimensionValidator.php

With 86 executable lines of code, this class appears quite balanced.

The class derives from `ValueValidators\ValueValidatorObject` for the purpose of code re-use, which is discouraged (see chapter 3.1.5). As a result, the storing of error messages in `$this` appears confusing when analyzing the code. Instead of collecting error messages using functionality of the base class and leaving the determination of a result object to it, this functionality could better be put into a dedicated class.

The option handling could potentially be extracted into its own class, being responsibly only for validation and sanitation of options for the `ValueValidators\DimensionValidator`. This would result in slightly better testability.

The `doValidation()` method could be split up into small private methods with speaking names to enhance code readability.

DataTypes\DataTypeFactory::newType()

DataValues/DataTypes/includes/DataTypeFactory.php

With an NPath complexity value of 1,620, this is the most complex method of the extension. While manual investigation showed that the method is really somewhat complex, it is harmless compared to the findings in the Wikibase extension.

Analysis revealed that this class actually performs some kind of dependency injection approach. It should be replaced by a dedicated approach for that purpose, for example as suggested in chapter 4.1.

DataValues\TimeValue::__construct()

DataValues/DataValues/includes/values/TimeValue.php

With an NPath complexity of 1,296, this class ranked second in DataValues. Manual inspection indicated that the code is actually not complex to read for a human being, since it only originates from parameter validation with early return by an exception on error:

```
public function __construct( $time, $timezone, $before, $after, $precision,
$calendarModel ) {
    if ( !is_string( $time ) ) {
        throw new InvalidArgumentException( '$time needs to be a string' );
    }

    if ( !is_integer( $timezone ) ) {
        throw new InvalidArgumentException( '$timezone needs to be an
integer' );
    }

    if ( $timezone < -12 * 3600 || $timezone > 14 * 3600 ) {
        throw new OutOfBoundsException( '$timezone out of allowed bounds' );
    }

    // … more validation …

    // Can haz scalar type hints plox? ^^

    $this->time = $time;
    $this->timezone = $timezone;
    $this->before = $before;
    $this->after = $after;
    $this->precision = $precision;
    $this->calendarModel = $calendarModel;
}
```

As can be seen, this code is perfectly valid and easy to understand.

DataValues\GeoCoordinateValue::__construct()

DataValues/ValueParsers/includes/parsers/GeoCoordinateParser.php

The complexity of 288 shown in this method is perfectly valid for validation code that returns early with an exception. See previous section for details.

DataValues\DataValueObject

DataValues/DataValues/includes/DataValueObject.php

This file provides default implementations of required interface methods for data value objects. The code has very low complexity and seems to be valid. Extensive tests and only necessary changes are recommended, since this class was identified to be most essential to the component, using the CodeRank metric.

ValueParsers\StringValueParser

DataValues/ValueParsers/includes/parsers/StringValueParser.php

Similar observations as in the previous section apply to this class. However, this class uses the Template Method[39] pattern, which indicates code re-use by inheritance. Still, since the degree is really low, it is not an issue in this specific case.

ValueValidators\ValueValidatorObject

DataValues/ValueValidators/includes/ValueValidatorObject.php

While similar observations as for the previous two sections apply, this class makes more extensive use of code re-use by inheritance. For example, the `valueIsAllowed()` method provides an implementation of white/black listing that can be triggered by derived classes, if wanted. A better solution would be to provide this functionality by a dedicated validator that can be aggregated by others, if such functionality is required.

General

The use of interfaces and abstract classes is a purely technical one. While this is alright and widely used, the reading of the following blog post about semantics of these two concepts is recommended:
http://qafoo.com/blog/026_abstract_classes_vs_interfaces.html

The `ValueParsers\ValueParserFactory` class creates objects from registered classes, which prevents object life cycle control by implementers of parsers. An abstract factory should rather be used. Refer to chapter 3.1.3 for further details on this issue.

6.1.3 Diff

As for the DataValues extension, the Diff extension was only analyzed roughly. Due to the extension size, the rule of investigating the top three classes for

39 http://en.wikipedia.org/wiki/Template_method_pattern

each of the selected metrics was softened to only analyze a smaller number.

The executable lines of code statistics did not reveal any outliers or extraordinary high numbers.

Diff\MapDiffer::doDiff()

Diff/includes/differ/MapDiffer.php

This is the class with the highest complexity in the Diff component. The degree of complexity is acceptable. However, extensive unit testing is recommended to ensure proper functionality of the diff algorithm.

Diff\ListDiffer

Diff/includes/differ/ListDiffer.php

The two different modes this class can work in could better be realized by delegating the `diffArrays()` method to an aggregated strategy object to enhance extensibility.

6.1.4 Ask

As for the DataValues and Diff extensions, the Ask extension was not in the focus of the review and was therefore only analyzed roughly. Due to the extension size, the normal approach of investigating the extension by metrics was skipped. No conspicuous metric values were detected.

Ask\Language\Description\DescriptionCollection

Ask/includes/Ask/Language/Description/DescriptionCollection.php

This class makes use of the `assert()` function in order to assert value plausibility. While this is not a general issue, incautious usage of `assert()` might lead to unexpected issues, if assertions are not properly configured in production systems. It is therefore questionable, if `assert()` should be used at all. However, there is no call for action.

Ask\Language\Description\ValueDescription

Ask/includes/Ask/Language/Description/ValueDescription.php

This class represents a node in the abstract syntax tree of descriptions. However, instead of using one node class per operation type, constants are used to express the operation represented by the node. This might lead to extra complexity when processing the abstract syntax tree of expressions in other components, which use the Ask component to represent expressions. Further investigation turned out, that there is no issue with representing extended expressions through a custom node type.

6.1.5 Tests

The test base of the Wikibase project consists of a mixture of integration and unit tests on basis of PHPUnit. As part of the code review, the two test cases with the highest ELOC metric were analyzed. The quality of the test cases appears to be satisfying. None of the critics collected in this section therefore requires urgent action. It is not recommended to change existing test cases at all, but to only apply the recommendations to newly created tests.

Wikibase\Test\ChangeHandlerTest

Wikibase/client/tests/phpunit/includes/ChangeHandlerTest.php

It appears that the test uses hand crafted “mock” objects in the form of in-memory emulation of the original applications. While this is a valid approach, such generic mock objects can become quite complex. Errors might easily result in falsified test results. It is recommended to use the PHPUnit mock API[40] or another mock framework for PHPUnit (e.g. Phake[41]) instead, where feasibly. This will also save time during test development and make the introduction into tests easier for new developers.

The usage of custom assertion methods like `assertChangeEquals()` is a valid approach and we recommend it in favor of multiple assertions in one test method. The requirement for creating large fixtures as done in e.g. `makeTestChanges()` should be considered an indicator for too high complexity of the code under test. The required data structure itself appears to be too complex.
The `testMergeChanges()` method tests success and failure cases in one go. It is recommended to split these cases into several test cases to make the tests more explicit and therefore easier to read and to maintain. For the error test case, the PHPUnit construct `setExpectedException()` could be considered.

The `testSingleton()` method actually tests two behavioral aspects at once and should therefore be split up.

Non-speaking names like `testHandleChanges()` should be avoided. The name of a test case should include the tested method and the tested aspect. Since the test asserts that MediaWiki hooks are actually called, a name like `testHandleChangesCallsHooks()` might be better suited.

The named test is also overusing the `@dataProvider` feature of PHPUnit. The test needs to use `func_get_args()` in order to detect how the method under test is to be called. For readability and maintainability it is recommended to use a dedicated test for the possible data sets instead.
The backup and reset of global variables like `$wgHooks` could better be handled in the `setUp()` and `tearDown()` methods to avoid cluttering of the tests themselves.

40 http://www.phpunit.de/manual/3.8/en/test-doubles.html
41 https://github.com/mlively/Phake

Wikibase\Test\EditEntityActionTest

Wikibase/repo/tests/phpunit/includes/actions/EditEntityActionTest.php

The `testActionForPage()` method appears to test test code.

The methods `adjustRevisionParam()` and `tryUndoAction()` are both too complex. It is important to avoid complex code in tests even more than it is in production code, in order to keep tests reliable and maintainable. The complexity of these methods is an indicator for too high complexity in the corresponding production code, too, and might be a hint for a missing abstraction.

6.2 JavaScript

The following notes relate to the JavaScript code of the Wikibase extension. They are structured by issue type.

6.2.1 Syntax and Coding Style Considerations

The code is mostly free of linting errors regarding Jslint[42]/JSHint[43]. The team confirmed that individual members perform occasional lint checks. However, there is no central JSLint/JSHint configuration available for the project to give all team members and the community a guideline and allow integration into the CI environment. It is therefore recommended to create such a configuration and commit it into the project source code. Furthermore, only linted code should be committed. There should be no exceptions from this rule for specific files. Most JSLint/JSHint checks are very sane and provide protection against a lot of JavaScript quirks. As the usage of a linting tool is to be introduced, JSHint is favored over JSLint as it has a more active community, better configurability and a more robust integration into CI systems.

There is good usage of `use strict` (strict mode) throughout the project. However, the file `client/resources/wbclient.linkItem.js` is currently the only one that does not use strict mode. A short investigation did not reveal a reason, for not having strict mode for this file. If there is a particular reason, the corresponding code should be refactored into a much smaller code file.

Each implemented jQuery functionality should reside in its own source file or plugin. This is mostly done quite well. However, one example exists, which should be cleaned up: `removeClassByRegex()` which is defined in `lib/resources/wikibase.utilities/wikibase.utilities.jQuery.js` should be moved to its own plugin.

In general, the renaming of variables passed to *module pattern* functions is considered bad practice for code readability reasons. Developers always need to scroll to the position of the invocation in order to take a look at what has been passed. Abbreviation counteracts readability, without providing much

42 http://www.jslint.com/
43 http://www.jshint.com/

benefit: IDEs will auto-complete the full qualified name to ease programming. A proper minification process will eliminate long names inside of module pattern functions. Therefore, there is no further size benefit for download by such premature minimization.

If there is still the need to shorten a submitted variable name, this aliasing should not occur through the function signature but rather in the first lines of the function code. As a result, it is clear from the signature which variables are expected and used.

6.2.2 Module Pattern Variations

Module Pattern is an encapsulation of different code entities into an anonymous self calling function, in order to isolate the module scope from the rest of the application. It is currently used throughout the application in the following way:

```
(function(gD1, aD, ...) {
  // Module code is put inside here
  window.someExportedThing = …
  // or
  globalDependency1.someExportedThing = …
})(globalDependency1, anotherDependency, ...);
```

This is a viable approach. However, the exported symbols are not directly visible. Nevertheless, this approach has the benefit of being able to export symbols to more than one namespace. In most situations this is however discouraged, as it does counteract readability.

An alternative can be:

```
var exportedNamespace.exportedSymbol = (function(globalDependency){
  // module internal code here

  return whateverShouldBeExported;
})(globalDependency)
```

Using this technique, only one symbol/namespace segment can be exported, which is what is desired most of the time. Furthermore, the public interface is easily visible and discoverable.

Another alternative, which is actually used inside the project, is:

```
var exportedNamesapce.exportedSubNamespace = new (function(globalDependency) {
  // module internal code

  this.someExportedFunctionOrObject = …
  this.anotherExportedFunctionOrObject = …
})(globalDependency);
```

This variant is even better suited, as the exported sub-namespace is quite easily visible. However, it is not directly visible which public API is exported. The usage of `new` in conjunction with a function expression may lead to irritation for inexperienced JavaScript developers.

The utilized module pattern should be unified and a solution which is most

appropriate for the team should be selected. With regards to how the application is currently structured the usage of the second pattern described here is recommended.

6.2.3 Event Handling

Registered events should always have an event namespace set

```
$element.on('click.namespace', …)
```

The namespaces allow to easily unregister all registered events on destruction of a widget. jQuery-UI actually automatically performs the unregistration on a call to `_destroy()`. For code inside a UI plugin, `this.eventNamespace` should be prepended to any event registration name. This property is automatically initialized during widget creation by prepending a dot to the widget name as well as appending a GUID. The most efficient way to support this is to store the value of the property to a variable local to the module pattern function during the `_create()` or `_init()` phase. This processing allows a minimizer to reduce the variable names length significantly, which is desired if many events are registered.

Always appending this created variable takes care of the following advantages: It ensures the usage of the correct and same namespace all the time. Furthermore, it ensures that jQuery-UI can automatically unregister the event if it is not needed any longer, i.e. once the widget is destroyed. Otherwise, event handling becomes slower and slower over time, as irrelevant events are looped over and are fired into the void.

The namespaces minimize event registration and unregistration collisions by different widgets and plugins. There is no need care of which other widget might have registered a certain event.

The usage of `this.eventNamespace` will most likely be deprecated in jQuery-UI version 2.0. The registration and unregistration of all events from within a widget should then be handled using `this._on(...)` and `this._off(...)` to allow for the following benefits: Automatic unregistration of events occurs on destruction without the need to manually provide event namespaces. `this` is automatically mapped to the widgets calling context without the need to utilize `$.proxy`. Events on "disabled" widgets will be suppressed (this can be disabled).

This functionality is already present in current jQuery-UI versions and should be used if these versions are ready to be utilized by the project. Jquery-UI versions are not directly bound to the used jQuery version. Therefore, it may be possible to utilize a quite current jQuery-UI implementation even though the project demands a lower jQuery library version.

6.2.4 Prototyping and Inheritance

Prototypes should not be defined as plain object literals:

```
var SomeClass = function() {...};
SomeClass.prototype = {
  …
}
```

This makes real prototypical inheritance below the first level impossible. The coding style then needs to be changed on inheritance, as otherwise the whole prototype would be overwritten, instead of only overwriting the inherited methods:

```
// Really minimalistic and basic inherits function

function inherits(base, child) {
  // Make sure base constructor is not called upon inheritance chain creation
  var ctor = function() {
    this.constructor = child;
  };
  ctor.prototype = base.prototype;

  // Create the proper prototypical inheritance chain, while still providing in

  // individual prototype state for extension
  var child.prototype = new ctor();
};

var A = function() {};
A.prototype = {
  foo: function() {}
};

// B is supposed to inherit from A and implement a new function bar
var B = function() {};
inherits(A, B);

// Same syntax as above
B.prototype = {
  bar: function() {
    // Do something, then call foo
    this.foo();
  }
}

// ^ Does not work, as we have overwritten the whole prototype,
// including the inheritance structure

// Something like this is needed:
B.prototype.bar = function() {...};

// Inconsistent coding styles with different inheritance levels
Codingstyle should be consistent throughout inheritance chains like this:
var SomeClass = function() {...};
SomeClass.prototype.someMethod = function() {...};
SomeClass.prototype.someOtherMethod = function() {...};
```

This approach works with real prototypical inheritance in every inheritance level. Currently only the `DataValues` extension suffers from this issue.

If literal object definition is wanted as the chosen coding style, `$.extend` should always be used to ensure no already existing properties are left behind. This is done in some parts of the DataValue extension as well.

By the time this report is written, the issue should already have been resolved.

There currently exist different forms of inheritance implementations. While the other extensions utilize different inheritance functions, DataValues has its own function which shows the following issues: The function can only be used for defining inheritance but not for defining a base class. Due to this insufficient base approach, wrappers like `ValueView.expert` exist, which are essentially only a small wrapper around inheritance again. These are less maintainable and hard to understand for non-functional developers. Therefore, a way should be created to allow defining all "classes" consistently across the project. This increases the readability as well as the maintainability.

From discussion with the Wikibase team it was decided to only use one complete wrapper around inheritance consistently throughout the project since compelling usage of a unified methodology is really important in this case.

Naming of constructor functions and corresponding static functions on a pseudo namespace should never only be distinguishable by lower and uppercase letters. Those kind of naming easily leads to confusion. Therefore jQuery-UI's own way, like `Widget` and `widget` is discouraged due to readability concerns.

6.2.5 Separation of concerns

While the code structure of Wikibase JavaScript code is generally quite good, there are some more complex widgets/plugins which could still be optimized by refactoring:

- wbclient.linkitem.js
- jquery.ui.suggester.js
- jquery.wikibase.claimlistview.js
- jquery.wikibase.claimview.js
- jquery.wikibase.edittoolbar.js
- jquery.wikibase.entityselector.js
- jquery.wikibase.entityselector.js
- jquery.wikibase.snakview/snakview.js
- jquery.wikibase.statementview.js
- wikibase.entity.js
- wikibase.repoApi.js

- jquery.ui.tagadata.js

Those could be split up into multiple different jQuery plugins to increase code maintainability a bit.

As an example for a plugin, wbclient.linkitem.js could be separated into several plugins, where each performs one of the following tasks:

- Check user login state and display appropriate dialog
- Wikibase specific dialog displaying and handling
- Retrieve linkable sites, i.e. all sites beside the current one
- Display and handle the site-link form
- Creation of the site-link table
- Linking of entities
- One time tooltips

In general, all functions that currently reside within this plugin could be split into multiple plugins for better readability, code re-use and maintainability.

As an example for a widget, `jquery.ui.suggester.js` could be re-factored as follows:

- Isolate the text field selection code (including feature detection and handling)
- Highlighting of matched characters in a list of links
- Calculation of the browser's scrollbar width

A lot of the functionality of the suggester actually belongs to the menu widget which is modified. Those functionality should be moved to a separate extension of the menu-widget, which is then used by the wb.suggester instead of the default menu widget.

The SnakView is a good example for well structured code.

6.2.6 Handling of Self Reference

The usage of `$.proxy` and closure based `var self = this` is inconsistent. Both ways of conserving the value of `this` for a callbacks are completely valid. It is important to not mix up those techniques, which can easily lead to confusion about the current state of `this`. As jQuery makes heavy use of `this` manipulations internally (in `$.each` and every plugin function for example) the usage of the `var self = this` method is encouraged over the use of `$.proxy`.

6.2.7 View and Logic Separation

Currently, a lot of *inline* DOM node manipulation using the jQuery-API is

performed. With this, widget logic and view logic is mixed up. To resolve this, the view logic should be separated from the widget using a JavaScript template language. A minimal, but very powerful template language is Handlebars[44]. This one should be sufficient for the Wikibase use case.

For the realization, it is recommended that each widget receives a set of required templates, which are used to create the corresponding DOM. The logic needed to handle special cases, like calling other jQuery plugins from within the newly created DOM nodes, should be implemented as template functions or blocks.

If the introduction of a template engine causes performance problems due to additional re-rendering cycles, a “dirty flag” might provide a solution. Alternatively, re-rendering could be preserved using a promise or an event bus system. The idea here is to delay rendering until an event has reached the top level of the DOM tree and to render all triggered changes at once.

If a migration to AngularJS[45] is performed, the toolkit can also be used for the templating task. It can easily be integrated with jQuery, while the integration with jQuery-UI widgets would take a little amount of work. The idea should be to create simple directives as wrappers for each jQuery-UI widget. The Angular-UI[46] thirdparty project could be used as a starting point to create such an integration. However, due to the current inconsistency and code quality of this project it should not be simply integrated, but only used as inspiration.

This way, AngularJS could be used as a "template" language during the first steps of the transition phase. The dependency injection system of AngularJS could be used to have a further decoupled application system. Later on, AngularJS *resource* and *http* abstractions can be used as well. Migrating tests over to AngularJS' karma-runner[47] (former testacular) eases the integration of testing with continuous integration and multiple different browser environments.

Once jQuery-UI widgets get more and more replaced by real directives (instead of wrappers), the advantages of AngularJS will become visible:

- No more "event-hell"
- Automatic data binding and view rendering
- Faster rendering due to the integrated rendering-queue in AngularJS
- Better separation of concerns due to ease of dependency injection
- Easier re-usability

Due to its decoupled nature and structure, AngularJS is very modular. Only the parts required for the current refactoring stage need to be loaded. This eases

44 http://handlebarsjs.org
45 http://angularjs.org
46 http://angular-ui.github.io
47 http://karma-runner.github.io

the introduction of another framework on top of the MediaWiki stack. Due to its tight integration with jQuery, the possibility exists to only use AngularJS on the repository side. The client side could still work with jQuery only, without sacrificing a lot of functionality. This makes the integration with the currently existing MediaWiki environment easier.

6.2.8 jQuery Philosophy

jQuery and jQuery-UI's philosophy is to use a functional approach. Functions and events are the main tools for programming. There is no real object orientation. While this approach changed a little with jQuery-UI, functions are still the major implementation way. The basic idea is to create many different small plugins and widgets, similar to the philosophy of GNU tools.

Furthermore, the toolkits always work in a DOM-centric way. Data is always bound to DOM elements, for example widget instance data. Events are also mostly bound to DOM elements or to widget objects directly. Widgets and plugins are almost exclusively called on "existing" DOM nodes which have been selected using jQuery before. Therefore, the DOM structure needs to exist in advance most of the time in order to execute operations.

jQuerys main idea is to enhance an existing HTML/DOM structure with dynamic operations. In this regard, Angular is quite similar. However, it uses an approach which ensures the separation of Model/View/Controller (MVC). While jQuery often leads to a mixture of these concerns.

There exists a possibility to split up jQuery plugins in an object oriented way: Different concerns of a plugin can be split into different prototypes, which are then combined inside the jQuery plugin function itself. This is mostly not needed, as plugins should be small due to the frameworks philosophy.

It is harder to apply object oriented approaches to jQuery-UI widgets as those are defined in a stricter structure, using the widget factory. Nevertheless, it is possible if this is really desired, by creating separated prototypes for different concerns. Furthermore, a generic adapter is required which is capable of combining different prototypes into a widget-factory compatible structure again. Different concerns could be, for example:

- Event handling
- DOM manipulation or creation
- Option handling
- Public widget interface

6.2.9 Global Dependencies and Namespaces

Instead of putting all the different modules into global variables namespaced with `extension`, a central dependency injection mechanism should be

implemented to inject dependencies where they are required. If the switch to AngularJS is realized, the dependency injector provided by this framework is recommended to be used. Alternatively, if the switch to AngularJS is not performed, the wire.js[48] dependency injection container should be considered or a custom solution should be implemented, which could be inspired by AngularJS.

6.2.10 Performance Considerations

Throughout the code base, jQueries' `$.each()` function is used, mostly in rendering intensive loops. `for-in` and `for` loops are generally considered faster than a function call, especially if called many times. If profiling of the corresponding code structures reveals performance issues, switching to the loop variant should be considered. However, there should be no general switch of paradigms here because the `$.each()` variant is considered more readable in jQuery projects.

In general, rendering updates in loops should not work on the actual DOM tree directly, but should rather work on document fragments. Newly created elements are generally kept as a DOM fragment in jQuery, as long as they are not attached to the DOM, yet. If multiple operations are to be performed on already attached DOM nodes, they should first be detached using the `.detach()` function to avoid unnecessary re-flows.

Re-flows, i.e. re-layouting of the browser engine, are an expensive operation. Browsers try to group rendering of operations that manipulate the DOM by queuing them until either the next frame needs to be rendered or until some kind of geometry/layout information (e.g. dimensions, CSS properties, attributes, ...) is requested. For this reason, it is highly recommended to strictly separate read and write operations on the DOM.

The following example shows a violation of this principle:

```
var element = $("#someElement");
element.css("color", "red");
var background = element.css("background-color");
// parse background-color and for example add some color to it
var newBackground = parseAndAlterColorValue(background);
element.css("background-color", newBackground);
```

In contrast, the following snippet fixes the issue

```
var element = $("#someElement");
var background = element.css("background-color");
// parse background-color and for example add some color to it
var newBackground = parseAndAlterColorValue(background);
element.css("color", "red");
element.css("background-color", newBackground);
```

The second solution does not cause an unnecessary re-flow. Such operations

48 https://github.com/cujojs/wire

should of course generally be avoided in loops, as for each loop a re-flow would be triggered. Therefore, different loops should be created, one that does all the DOM read operations, before another one, which executes all the writing.

Transparency in combination with text is especially expensive to be rendered in terms of rendering time and number of re-flows. The same applies to fade-in and -out operations. If such animations are combined with DOM manipulations, extra re-flows might be the result. Using CSS animations for such purposes might be a better option, since modern browsers provide better hardware acceleration for those.

The Chrome Developer Toolbar[49] can help to detect unnecessary re-flows with profiling. Another view especially on this problem can be gained through Googles SpeedTracer[50].

6.2.11 More Consistent Usage of Promises

Usage of promises is a good way of handling asynchronous operations in JavaScript, which is already utilized most of the time throughout the Wikibase code. However, there is potential for improvement:

`$.when` may be used to create a logical “and” conjunction between multiple promises. For example like this:

```
var promiseA = someAsyncOperation();
var promiseB = someOtherAsyncOperation();

// New promise is resolved once A and B are resolved
var promiseC = $.when(promiseA, promiseB);
```

In addition, callbacks may return new promises to keep the callback nesting level low:

```
someAsyncOperation.then(function(resultA) {
  return doSomethingAsyncWith(resultA);
}).then(function(resultB) {
  return doSomethingElseAsyncWith(resultB);
}).then(function(resultC) {
  //...
});
```

Animation functions return promises as well. Actually, a promise can be registered on any jQuery-set. The promise will be resolved once the animation is complete or, if there is no animation, it will be resolved directly:

```
$("#someId").fadeOut(500).then(function() {
  // Do something
});

// No knowledge about running animations is required
// (promises can always be used)
```

49 https://developers.google.com/chrome-developer-tools/
50 https://developers.google.com/web-toolkit/speedtracer/

```
$("#someId").then(function() {
  // …
});
```

This can help to make the Wikibase JavaScript more readable and therefore better maintainable. For example in `wikibase.ui.PropertyEditTool.EditableValue.js`, the method `performApiAction()` could be cleaned up like this: First of all, the registering of further actions to the created promises should be moved to another function to create a structural separation between the code creating the promise and the one using it. Furthermore, according to the examples above the creation of a new `Deferred` object is not needed here. A simple `return` from inside the called `fadeout` functions callback in combination with returning its own return value as well would be fully sufficient. This works, as `fadeout` is an asynchronous operation itself and therefore already returns a promise, which can easily be reused in this situation.

Furthermore, some general best practices about promises should be applied here: A `Deferred` should never be passed around as an argument (like in `triggerApi()`). A `Deferred` should never leave its creation scope. Only the `promise` it provides should be returned. Otherwise the whole idea of `Deferred`s isolating the resolution scope of `promise`s is undermined.

In this special situation the `triggerApi()` should create a `Deferred`, handle its resolvement/failure and return the associated `promise`. The `_performApiAction()` should take this promise and register callbacks to it. As promises take away all problems with async race conditions, it is no problem to do this, even if the action is already completed upon callback registration. Using it exactly the opposite way, like done here, makes most of the promise features (explained above) go away and therefore counteracts the whole idea.

6.2.12 Avoiding "global" Data Stores

Currently, the `wikibase` object is a global storage for all fetched and retrieved entities. It is globally accessed by different widgets to render the corresponding data and information. The result is a static dependency, which should be eliminated by creating a `Store` interface that is capable of providing all the necessary information for each widget.

It is even recommended to split up the interface even further, to have different storages for different parts of the handled information. The created interface can be implemented against possibly different back ends, e.g. one that utilizes direct communication, on on basis of the already present `repoApi` and one that uses Local Storage for caching.

All implementations should provide the requested information using promises, since this allows for arbitrary operations to occur for fetching (synchronous as well as asynchronous). Such an API therefore provides the greatest possible flexibility. Wrapping literal values inside of promise can be realized using `$.when`.

A cache layer could, on this basis, easily be implemented using aggregation: A storage interface conforming cache layer, which simply aggregates another storage or maybe even multiple ones to combine data sources e.g. back end and Local Storage.

The actually used store is then injected into each component/widget which needs the provided information. Such injection should take place using a dependency injection as described in chapter 6.2.9. As an intermediate, simplified realization, a widget option could be used for injection.

The result is a much better separation of the dedicated concerns data fetching, caching and the actual usage of the data. A widget should never have to really request data and should not be responsible for the storage itself. Required data should just be available to the view layer through the described interface.

6.2.13 Unit Testing

The QUnit[51] framework, which is utilized by the Wikibase project, is an easy to use, minimalistic unit testing solution. However, it has some drawbacks, especially regarding the integration with CI systems: Currently a workaround using Selenium and parsing the created HTML/DOM is implemented.

Additionally, QUnit does not provide many different assertions, which could help during error isolation. Mostly, QUnit uses true, false or equals assertions. As a result, the developer needs to provide additional descriptive information for almost every assertion.

Furthermore, QUnit does not adhere to the idea of one assertion per test. As a result, many of the currently implemented tests are too broad and should be divided into different separated test cases. Even though this clutters the output generated by QUnit, it helps the readability of the tests. Separate test cases can provide a good introduction for new developers and the community.

Calling internal functions of the units under test to provide test data, which would otherwise be injected asynchronously, is discouraged. A better solution would be to mock the data provider itself (or the promise) using e.g. Sinon.js[52]. This allows to only test the public API while leaving the asynchronous operations out of scope. Sinon.js does help with the testing of `XMLHttpRequests` as well. Furthermore, it provides sophisticated tools to monitor, mock and test function calls themselves. Timed events like animations and `setTimeout()` callbacks can be tested synchronously as well.

It is recommended to switch to a more advanced test runner for the purpose of having a smoother CI integration. Js-test-driver[53] or karma-runner[54] (former testacular) could be good alternatives to the current QUnit/Selenium approach. These also allow you to run tests on multiple browsers in parallel, while

51 http://qunitjs.com/
52 http://sinonjs.org/
53 https://code.google.com/p/js-test-driver/
54 https://code.google.com/p/js-test-driver/

aggregating results in a `junit.xml` compatible log. Furthermore, code coverage analysis is possible with both systems. Both runners can use QUnit test cases mostly out of the box.

For the time being, while this switch is not executed, the usage of the chai.js[55] assertion framework in conjunction with the chai-jquery[56] plugin can help to write cleaner tests, due to more precise and powerful assertion capabilities. The chai.js framework is completely test-runner agnostic and can therefore be easily integrated into the current QUnit tests while still being used, once the migration to another framework is finished.

55 http://chaijs.com/
56 http://chaijs.com/plugins/chai-jquery

Qafoo GmbH
Bochumer Strasse 226
45886 Gelsenkirchen
http://qafoo.com
contact@qafoo.com

Geschäftsführer:

- Kore D. Nordmann
- Manuel Pichler
- Tobias Schlitt
- Jakob Westhoff

AG Gelsenkirchen HRB 10560
Steuernummer 319 / 5764 / 0827