A Hands-on OpenStack Code Refactoring Experience Report

Gábor Antal[1], Alex Szarka[1], and Péter Hegedűs[2]

[1] Department of Software Engineering, University of Szeged, Hungary
{antal, szarka}@inf.u-szeged.hu
[2] MTA-SZTE Research Group on Artificial Intelligence, Szeged, Hungary
hpeter@inf.u-szeged.hu

Abstract. Nowadays, almost everyone uses some kind of cloud infrastructure. As clouds gaining more and more attention, it is now even more important to have stable and reliable cloud systems. Along with stability and reliability comes source code maintainability. Unfortunately, maintainability has no exact definition, there are several definitions both from users' and developers' perspective. In this paper, we analyzed two projects of OpenStack, the world's leading open-source cloud system, using QualityGate, a static software analyzer which can help to determine the maintainability of software. During the analysis we found quality issues that could be fixed by refactoring the code. We have created 47 patches in this two OpenStack projects. We have also analyzed our patches with QualityGate to see whether they increase the maintainability of the system. We found that a single refactoring has a barely noticeable effect on the maintainability of the software, what is more, it can even decrease maintainability. But if we do refactorings regularly, their cumulative effect will probably increase the quality in the mid and long-term. We also experienced that our refactoring commits were very appreciated by the open-source community.

Keywords: OpenStack, refactoring, quality assurance, static analysis, code quality

1 Introduction

Restructuring the source code of an object-oriented program without changing its observed external behavior is called refactoring. Since Opdyke's PhD dissertation [28] – where the term was introduced – and Fowler's book [16] – where refactoring is used on "bad-smells" – many researchers studied refactoring as a technique to improve the maintainability of a software system. However, there are still only few empirical evidences that would objectively prove the the effects of refactoring on code maintainability.

In an earlier work [17] we analyzed the effect of code refactorings by automatically extracting refactorings using the RefFinder [21] tool from various small and mid-sized open-source projects. As a means of measuring maintainability, we used static source code metrics. We found that refactorings reduce complexity and coupling, however, code clones were affected negatively and commenting

did not show any clear connection with refactoring activities. The disadvantage of this approach was the high false positive rate of RefFinder and the lack of real, certainly valid set of refactorings to analyze.

During an in-vivo industrial experiment [38], we showed that bulk-fixing coding issues found by various code linters can lead to a quality increase. The drawback of this approach was that developers tend to perform the easiest code changes, which typically were not even classical Fowler refactorings.

In this paper, we present a large-scale, open-source study on the OpenStack cloud infrastructure [33], where we contributed a large number of refactoring code modifications to the upstream code bases. For quality estimation we used the QualityGate maintainability assesment framework [6] and the SonarQube[3] technical debt measurement model [22]. we calculated the system-level maintainability of the OpenStack modules before and after the refactoring patch is being applied on the code base. This way we could analyze the direct impact of our code rafactoring activity on the OpenStack modules.

We found that often such small, local refactoring changes will not have any traceable effect on the overall level of maintainability of a large OpenStack module. This was particularly true for the Nova module, as it is very large compared to Watcher for example. In case of Watcher, the smaller module we analyzed, we found that several refactoring changes caused a quality increase and a technical debt decrease. Nonetheless, in general we can say that in most of the cases where we were able to measure the maintainability changes, refactorings had a positive effect on them. It was a slightly easier to show a decrease in the technical debt measures in SonarQube.

The rest of the paper is organized as follows. In Section 2, we list the related literature and compare our work with them. In Section 3, we introduce OpenStack, the contribution workflow, and what are the difficulties we had to face with, in order to contribute to the code. Section 5 describes how the patches were created and how we analyzed our merged patches. In Section 6, we present the results of the analysis. We list the possible threats of the analysis in Section 7 and conclude the paper in Section 8.

2 Related Work

Mens et al. published a survey to provide an extensive overview of existing research in the area of software refactoring [25]. Unfortunately, it is still unclear how specific quality factors are affected by the various refactoring activities.

Many researchers studied refactoring but only a few papers analyze the impact of refactoring on software quality. Sahraoui et al. [34] use quality estimation models to study whether some object-oriented metrics can be used for detecting code parts where a refactoring can be applied to improve the quality of a software system. They do not validate their findings within an large-scale experiment. Stroulia and Kapoor [37] investigate how size and coupling metrics behave after refactoring. They show that size and coupling metrics of a system decrease after refactoring; however, they only validate this in an academic environment.

[3] `https://www.sonarqube.org/`

Bois et al. [11] propose refactoring guidelines for enhancing cohesion and coupling metrics and obtain promising results by applying them on an open-source project. Simon et al. [35] follow a similar strategy, they use a couple of metrics to visualize classes and methods which help the developers to identify the candidates for refactoring. Demeyer [10] shows that refactoring can have a beneficial impact on software performance (e.g. compilers can optimize better on polymorphism than on simple if-else statements). Bois and Mens [12] develop a framework for analyzing the effects of refactoring on internal quality metrics, but again, they do not provide an experimental validation in an industrial environment.

There are abundance of studies [9, 26, 27, 36] on OpenStack, the cloud infrastructure we used in our study. However, they do not target the analysis of code refactoring effects on software quality. Baset et al. [9], for examle, analyze the evolution of OpenStack from testing and performance improvement point of view. Slipetskyy [36] analyze OpenStack from a security perspective. Musavi et al. analyze the API failures in the OpenStack ecosystem [26,27]. Advani et al. [5] extract refactoring trends from open-source systems in general.

In our paper we performed a large number of refactorings and observed their effect on the quality of the OpenStack open-source cloud infrastructure modules. These refactorings fixed different kinds of coding issues and introduced improved code structures, thus so we could investigate the system states before and after applying different types of patches. Our work was carried out in an *in vivo* open-source environment (OpenStack has a huge contributor base and is very active), which is an important difference compared to previous studies.

3 OpenStack

3.1 Overview of OpenStack

OpenStack is an open-source cloud computing software platform [13, 15, 29, 33], mainly developed in Python. It is mostly used to provide an infrastructure-as-a-service platform, it helps their users to create private and/or public clouds. In this model, the service provider (who runs OpenStack) provides virtual machines and other resources to their customers (who pays for the resources monthly or on-demand). OpenStack is modularly developed, which means you can customize your cloud infrastructure well. You can deploy only the modules you really need, e.g. if you need a load balancer to your cloud system, you can install the specific module (Octavia) but your cloud can fully operate without it.

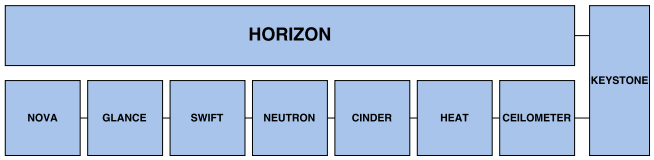

Fig. 1. Schematic structure of OpenStack

There are several core services which is often used together, including Nova (managing computing resources), Keystone (authentication and authorization), Neutron (networking), Glance (image services for virtual machine images), Horizon (dashboard). A schematic figure of the modules can be found in Figure 1. All of the modules have their own developers, core members (who are experts of the module, and who can directly push modifications into the codebase). These teams are usually not distinct, one contributor often develops to several modules. You can find the list of OpenStack modules in [4].

OpenStack was started in 2010 as a collaboration project between Rackspace and NASA. As the project became more and more popular, other companies have joined and started working on the core modules and also on their own drivers in OpenStack, for example there are several hardware hypervisors that OpenStack can use (e.g. HyperV, VMware, Xen, Libvirt).

Nowadays, the project is handled by the OpenStack Foundation [29], which is a non-profit organization. Their goal is to provide an independent and strong home for OpenStack projects, provide resources and an entire ecosystem to build community, and help collaboration between open-source technologies. OpenStack Foundation consists of more than 500 companies (e.g. Intel, Canonical [14,32]), 82000 community members, from more than 180 countries worldwide.

The full codebase of all OpenStack modules is now consists of more than 1 million lines of code. OpenStack projects often use a six-month release cycle [30, 31] with several (typically 3) development milestones. This agile model lets the contributors to quickly land new features to OpenStack.

3.2 Contribution

As OpenStack is an open-source software, everyone can contribute to it. The developers are completely decentralized around the world, they have different coding habits. In order to contribute, some contribution guidelines [3] must be followed. The schematic view of the code developing process can be seen in Figure 2.

The pre-requisite is to get familiar with Git, as the OpenStack community uses it as their version control system, and GitHub to host their codebase. First of all, if you want to contribute, you need to set up your local environment where you can develop, execute tests, and run the components of the OpenStack for your features (or bug fixes) on a live system. After that, you have to clone the repository you want to work on. Of course, then you have to make the modification you want to do. Then, you have to commit the changes to your local Git repository. After that, you have to send the modifications to a code review system. Code review is a manual process when people verify each others code by hand. This process needs to be highly automated in a worldwide open-source system.

To achieve this, the OpenStack projects use the Gerrit code review system from Google. Based on Git version controlling, Gerrit is a powerful code review system, which manages the modifications that are intended to be merged to a Git repository (to OpenStack's GitHub repository in our case). Gerrit is a great

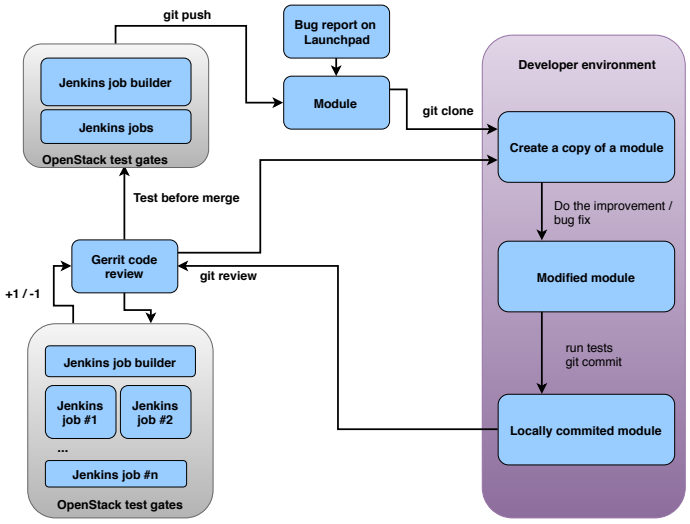

Fig. 2. OpenStack contributon flow

platform where developers can share their opinion on a modification, giving feedback to the authors how they can improve their patches, what are the additional tasks with a patch. They can rate patches $+1$ or -1 based on whether they found the patch mature enough to get merged or not.

At OpenStack, developers can get their modifications merged only via Gerrit. In Gerrit, every project has core members who are experts in that specific project. At least two of the core members need to approve the code before it can be merged. Core members can approve a patch by rating it $+2$ or disapprove by rating it -2. It basically means that a patch is probably not needed at that time. This process helps both developers and maintainers to have a good quality code [20, 24].

In addition to the manual approval, there are automated Jenkins[4] tests which can also approve or disapprove the code. When a Jenkins job fails, the author has to fix the code before it can be merged. These Jenkins jobs run the tests of the projects in various environments (e.g. with specific version of Python), with various settings enabled, on various hardware platforms. There are some really specific Jenkins jobs which only run on a specific code change (e.g. on specific, hardware-related driver changes). The jobs are managed by the Jenkins job builder (which is also an OpenStack project[5]). The Jenkins job builder is triggered when someone uploads a new set of modifications to the project or updates an already existing one.

When both Jenkins and the core members of the project approve a patchset (a set of modifications), the changes are getting tested once again, before being

[4] Jenkins is an open-source continuous integration system. More details can be found at `https://jenkins.io/`

[5] `https://github.com/openstack-infra/jenkins-job-builder`

merged to a specific branch (mostly to the main branch) of an OpenStack project. If the patch still seems to be correct, an automated system merges it.

3.3 Quality assurance

Quality assurance is one of the most important tasks in software engineering. However, it is not trivial even for a company, and also not a trivial task when the developers are all around the world, represent different companies, or they are just individuals who want to work on an open-source software system. Thus, a well-defined quality assurance model is necessary in order to have a good quality, manageable project. At OpenStack, there are 3 main aspects of quality assurance.

Manual code review. People can verify each others planned modifications. More details on how it works can be found in Section 3.2.

High test coverage. Having high test coverage is also a really important, continuous job in an international open-source project like OpenStack. Since everyone can contribute, it is cardinal to know whether a modification has unwanted side effects or not. To find out, we need to run the test suite provided by the project. If there is something we did but not intended to do, we can investigate the source of the problem. The OpenStack modules have quite good code coverage: Nova has a coverage of 87%, Watcher has a 86% coverage, Glance has a coverage of 72%, and Neutron has a 85% coverage.

To keep the coverage high, most of the time when a new feature is implemented, the patchset have to contain test cases for the new features. This is also a validation point when a bug fixing patch is getting merged. In general, a bug fixing patch should also contain test cases where the fixed bug would have occurred. At OpenStack, there are 3 levels of testing.

The first one is unit testing, which is needed for almost every case. Second level is the integration tests. This is only a common test level which is necessary when someone create modifications between 2 or more components. The last one is the live tests that are only required when one creates a modification, which can only be tested on a live and running OpenStack instance (tempest[6] tests). The used test coverage tool measures only the unit test coverage, and since these are functions that can be tested only with integration test, they are not taken into account while measuring coverage.

All of the tests are run with the *tox* testing framework. This helps to automate and unify testing procedure across modules so that all the modules can run the tests in the same way. Tox builds virtual, separated environments where the tests can be run. The ability to change between Python versions is also an advantage of tox, as OpenStack mostly supports Python 2.7 and 3.4 in parallel.

The pep8 Jenkins job. There are so called Jenkins test gates that test the modification from quality point of view. These jobs execute the pep8[7] tool to check if the code complies with the coding standards. In addition, these jobs run the style guide checker tool as well, which is created by the OpenStack

[6] `https://github.com/openstack/tempest`

[7] https://www.python.org/dev/peps/pep-0008/

community and called *hacking* rules. Hacking is a set of *flake8* plugins that test and enforce rules defined by [1].

There are global *hacking* rules, which must be conformed in each and every module. Additionally, every module can create its own *hacking* rules, which must be conformed only by that module. Local rules are often stricter than the global ones and usually contain rules that only make sense in the defining module.

4 Tools to measure code quality

4.1 QualityGate

For the definition of software quality we refer to the ISO/IEC 9126 standard [18] and its successor the ISO/IEC 25000 [19], which defines six high-level characteristics that determine the product quality of software: *functionality, reliability, usability, efficiency, portability*, and *maintainability*. There is another related standard, the ISO/IEC 14764 that describes the quality of the maintenance process (for example, maintenance planning, execution and control, evaluation). The proposed metrics address process quality, while we wanted to study product metrics particularly as our analyzed projects already had a well-defined release and maintenance cycle [31].

Due to its direct impact on development costs [8], and being in close relation with the source code, maintainability is one of the most important quality characteristics.

To calculate the absolute maintainability values for every revision of the system we used the Columbus Quality Model, a probabilistic software quality model that is based on the quality characteristics defined by the ISO/IEC 25000 [19] standard. The computation of the high-level quality characteristics is based on a directed acyclic graph (see Figure 3) whose nodes correspond to quality properties that can either be internal (low-level) or external (high-level). Internal quality properties characterize the software product from an internal (developer) view and are usually calculated by using source code metrics. External quality properties characterize the software product from an external (end user) view and are usually aggregated somehow by using internal and other external quality properties. The nodes representing internal quality properties are called *sensor nodes* as they measure internal quality directly (light yellow nodes in Figure 3). The other nodes are called *aggregate nodes* as they acquire their measures through aggregation. In addition to the aggregate nodes defined by the standard (i.e. Testability, Analyzability, Changeability, Stability, Reusability, Modifiability) we also introduce new ones (i.e. Complexity, Fault proneness, Comprehensibility, Documentation).

For the Python version of the model we used for the analysis of OpenStack before and after the refactorings, the following source code metrics apply:

- *LLOC (Logical Lines Of Code)* – the LLOC metric is the number of non-comment and non-empty lines of code.
- *NOA (Number Of Ancestors)* – NOA is the number of classes that a given class directly or indirectly inherits from.

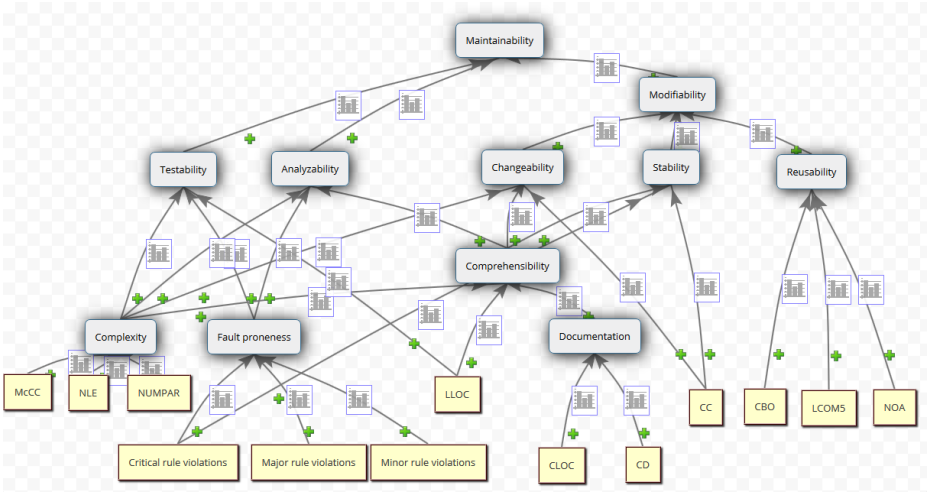

Fig. 3. Attribute Dependency Graph of Columbus Quality Model

- *NLE (Nesting Level Else-if)* – NLE for a method is the maximum of the control structure depth. Only `if`, `for`, and `while` instructions are taken into account and in the `if-elif-else` constructs only the `if` instruction is considered.
- *CBO (Coupling Between Object classes)* – a class is coupled to another if the class uses any method or attribute of the other class or directly inherits from it. CBO is the number of coupled classes.
- *CC (Clone Coverage)* – clone coverage is a real value between 0 and 1 that expresses what amount of the item is covered by code duplication.
- *LCOM5 (Lack of Cohesion On Methods)* – the value of the metric is the number of local methods that share at least one common attribute, call an abstract method or invoke each other, not counting constructors, getters, or setters.
- *NUMPAR (NUMber of PARameters)* – the number of parameters of the methods.
- *McCC (McCabe's Cyclomatic Complexity)* – the number of decisions within the specified method plus 1, where each if, for, while counts once and each try block with N catch counts N+1.
- *CLOC (Comment Lines of Code)* – for a method/class/package, CLOC is the number of comment and documentation code lines of the method/class/-package, excluding its anonymous and local classes/nested, anonymous, and local classes/subpackages respectively.
- *CD (Comment Density)* – ratio of the comment lines of the method/class/-package (CLOC) to the sum of its comment (CLOC) and logical lines of code (LLOC).
- *Critical rule violations* – Critical issues found by the Pylint tool[8] in the code that can cause bugs and unintended behaviour.

[8] `https://www.pylint.org/`

- *Major rule violations* – Major issues found by the Pylint tool in the code that can cause e.g. performance issues.
- *Minor rule violations* – Minor issues found by the Pylint tool in the code that e.g. decrease the code readability.

The edges of the graph represent dependencies between an internal and an external or two external properties. The aim is to evaluate all the external quality properties by performing an aggregation along the edges of the graph, called Attribute Dependency Graph (ADG). We calculate a so called *goodness value* (from the [0,1] interval) for each node in the ADG that expresses how good or bad (1 is the best) the system is regarding that quality attribute. The probabilistic statistical aggregation algorithm uses a so-called benchmark as the basis of the qualification, which is a source code metric repository database with 100 open source and industrial software systems.

For further details about Columbus Quality Model [7], see work by Bakota et al. They also showed that there was a converse exponential relationship between the maintainability of a software system and the overall cost of development [8], supported by an empirical validation. The QualityGate SourceAudit tool by Bakota et al. [6] is based on this quality model.

4.2 SonarQube

SonarQube is an open-source static analyzer for continuous inspection of code quality. It is written in Java and it can work with more than 20 programming languages, including Python. It has a user-friendly web front-end where one can read the analysis reports and inspect the source code. SonarQube is able to detect various code smells, violation of coding standards, code duplications, test coverage, and security issues. SonarQube also measures some software metrics, and helps the developers with advices. On the web front-end, developers can see several figures, tables that helps the understanding of the system.

It also integrates a high-level quality indicator called "Technical debt". Techinal debt shows how many person days of effort would be needed to fix all of the found problems. The technical debt module in SonarQube is based on the SQALE model [22]. We used SonarQube for an initial investigation of the source code in order to find bad smells, which we can refactor.

We have created an own SonarQube server[9], which analyzes some of the core modules to help others in finding issues for refactoring. Along with this we have added a quality improvement plan [2]. Since then, our quality improvement plan is available in the official contribution guide [3].

5 Methodology

A schematic overview of our process can be seen in Figure 4. First of all, we selected two of the OpenStack modules where we wanted to work on and create refactorings. We wanted to do refactoring on an mature, bigger module, and also on a smaller, younger one to see if that matters in the case of refactoring. We selected Nova and Watcher to work on. Nova is the oldest project in OpenStack

[9] `http://openqa.sed.hu`

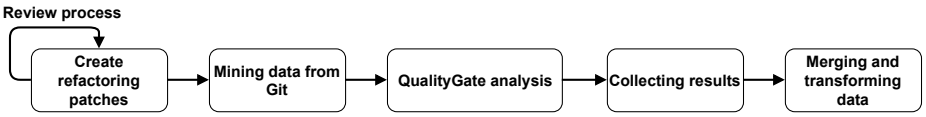

Fig. 4. Overview of the process

that is responsible for the resource management. Watcher is a younger module, which is responsible for resource optimization in multi-tenant OpenStack clouds. Nova has 362480 lines of code, Watcher has 43866 lines of code.

We used three sources of oracles to decide where to refactor: static analyzers (results of both SonarQube and QualityGate), issue tracking system (maybe there are already detected bad smells), and manual code investigation (we searched for bad smells, code snippets which can be simplified).

We have managed to merge 47 patches in the two modules: 22 patches were merged into Nova, 25 patches were merged to Watcher, all of them contain refactoring to a specific issues, like removing code duplication, replacing orders when catching exceptions, using more specific assert methods in tests, fixing indentation, reducing code complexity. All details of the changes can be found in our online appendix.[10]

Sometimes our proposed patches got merged quickly, and sometimes we had to modify them several times before core members approved them. After our patches got merged, we did some data mining from GitHub.

It is obvious that if we want to examine the effect of a refactoring, we have to analyze both the version of the source code before refactoring and after refactoring. To achieve this, we collected the hashes (the unique identifier of a commit in Git) of our patches and we also collected the hash of the commit before the refactoring along with other data (date, commit message). The data mining script is also available in our online appendix.[10]

Once we finished mining, we did the analysis with QualityGate (Section 4.1) based on the list collected from the GitHub repository. We analyzed the patches pairwise, we used the commit before the refactoring as an initial state, and then we analyzed the commit after the refactoring. Analyzing all of the commits which were on the list took hours.

After finishing the analysis, we had to collect all the data from QualityGate. QualityGate stores its qualification data in a database, which we had to query in order to get the raw data. Then, the collected raw data had to be transformed into a more readable form where we can investigate all of the quality indicators defined in Section 4.1 and compute the effect of a refactoring. Furthermore, we needed some other data from the mined Git repository, so we combined all of the collected data into a big table. All of the combined data is available in our online appendix.[10]

[10] `http://www.inf.u-szeged.hu/~antal/pub/2018-iccsa-openstack-refactoring/`

6 Results

6.1 Overall change of maintainability

The effect of our submitted refactorings on the code quality can be seen in Figure 5 and Figure 6. In these figures, blue lines represent the succeeding commits (before and after a refactoring), and between the pairs we used dashed gray lines. The y-axis shows the value of *Maintainability* as shown in Figure 3, the x-axis shows the number of analyzed versions.

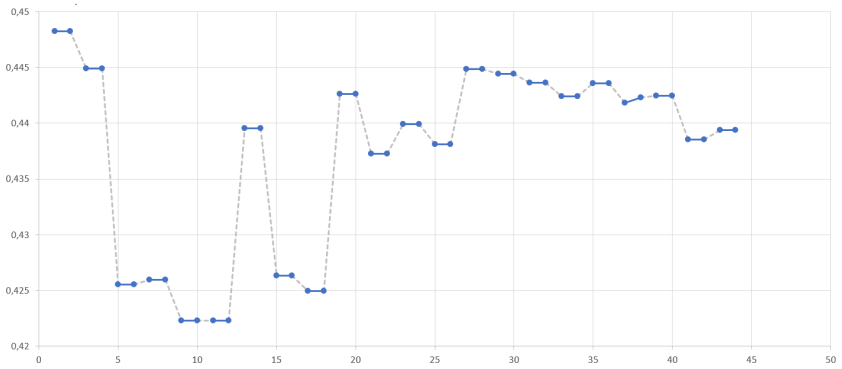

Fig. 5. Maintainability change of Nova

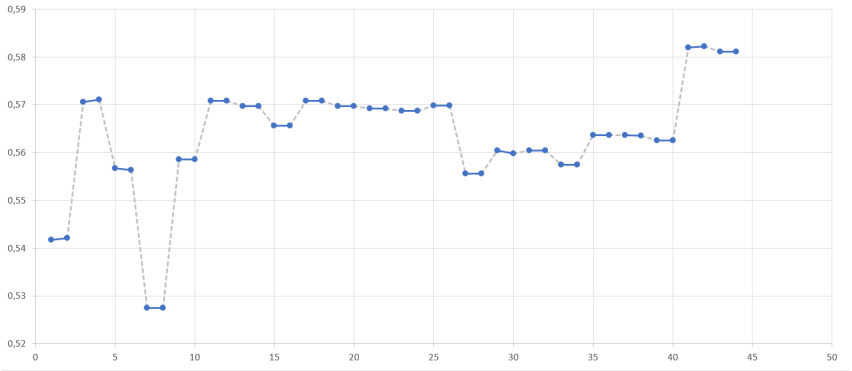

Fig. 6. Maintainability change of Watcher

As it can been seen, most of the time refactorings made undetectable changes in the quality of the whole system. We consider this to be normal as the systems had lots of lines of code, thus small changes have very small impact on the module as a whole. However, in some cases QualityGate showed a change in the software's quality. Out of the 47 patches, significant change could be detected only in 8 cases – 7 out of 25 in Watcher, which is the smaller project, and 1 out of 22 case in Nova, which is the bigger project. The detected changes and impact on *Maintainability* can be seen in Figure 7. The changes of aggregated quality

nodes and sensor nodes can be seen in Table 1 and 2, where negative numbers represents increased quality (the node had a better quality after refactoring; we subtracted the value of the quality indicator after the refactoring from the value before refactoring).

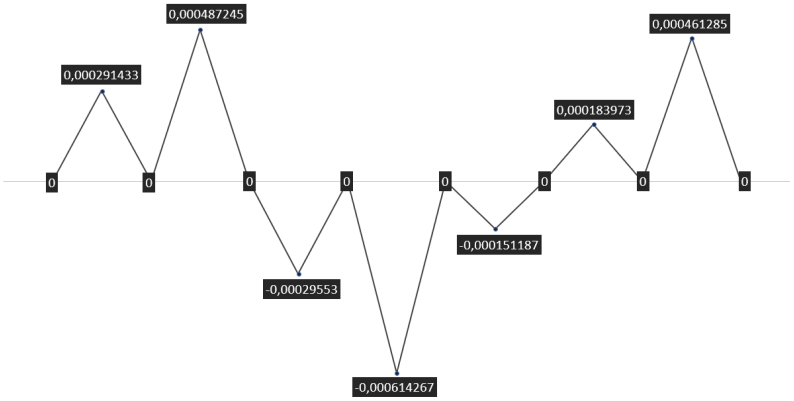

Fig. 7. Maintainability changes on patches which had any impact on quality

The significant patches caused increasing the software's *Maintainability* in 4 cases; decreasing the quality in 3 cases; and caused zero change in 1 case. The aggregate quality-indicator nodes *Analyzability*, *Stability* and *Testability* show increase in 5 out of 8 cases. It is also an interesting fact that the goodness of *Documentation* were decreased in all affected patches. All the analysis result data can be found in our online appendix.

Table 1. Changes in the aggregated quality nodes

(N)ova/(W)atcher	Analyzability	Changeability	CodeComplexity	Comprehensibility	Documentation	CodeFaultProneness	Critical rule violations	Major rule violations	Minor rule violations	Modifiability	Stability	Testability	Maintainability
N	-0.00049	-0.00043	-0.00060	-0.00063	0	0	0	0	0	-0.00028	-0.00025	-0.00070	-0.00046
W	-0.00015	0.00002	0.00049	-0.00048	0	-0.00085	-0.00143	0	0	-0.00019	-0.00055	-0.00067	-0.00029
W	-0.00062	-0.00046	-0.00111	-0.00014	0.00125	-0.00057	0	-0.00143	-0.00143	-0.00022	-0.00005	-0.00070	-0.00049
W	0.00104	0.00075	0.00321	-0.00006	0.00079	-0.00128	-0.00143	-0.00143	0	0	-0.00088	-0.00047	0.00030
W	-0.00033	0.00055	-0.00066	-0.00010	0.00069	-0.00015	0	0	-0.00143	0.00058	0.00095	-0.00038	0
W	0.00066	0	0	0	0	0.00341	0.00571	0	0	0.00053	0.00145	0.00069	0.00061
W	0.00019	0.00017	0.00033	0.00015	0	0	0	0	0	0.00010	0.00006	0.00017	0.00015
W	-0.00014	-0.00012	0	-0.00035	0	0	0	0	0	-0.00010	-0.00014	-0.00038	-0.00018

Table 2. Changes in the sensor nodes quality

	CC	CD	CLOC	LLOC	McCC	NLE	NUMPAR
Nova	0	0	0	-0.00143	0	-0.00143	0
Watcher	0	0	0	-0.00286	-0.00143	0	0.00429
Watcher	0	0	0.00286	0	-0.00143	-0.00143	0
Watcher	-0.00143	0.00143	0	-0.00714	-0.00143	0.00571	0.00571
Watcher	0.00286	0	0.00143	0	0	0	-0.00286
Watcher	0	0	0	0	0	0	0
Watcher	0	0	0	0	0	0	0.00143
Watcher	0	0	0	-0.00143	0	0	0

6.2 Quality degradation

The definition of refactoring implies that refactoring a code will increase the quality of the source code. Nonetheless, the *Maintainability* of the project decreased in 3 cases, which is almost 40% of all the cases. We investigated these cases to find out the reason of decrease.

First, we investigated the patch titled "`Reduced the complexity of the execute() method`".[11] In this patch, we split a large method that had a high cyclomatic complexity. We introduced 5 more methods, which impacted the quality; we improved the sensor nodes *McCC*, *LLOC*, *CC*, and improved the aggregated nodes *CodeFaultProneness*, *Stability*, *Testability*, *Critical rule violations*. Quality value of *Analyzability*, *Changeability*, and *NUMPAR* has decreased. These changes are consistent with the introduction of new methods. In overall, the number of methods has grown, which means that there are more methods in the code after the refactoring, there are more connections between them, so inspecting the connections between the code is harder than before.

The second patch that caused decreasing Maintainability is titled "Missing super() in API collection controllers". [12] In this case, we placed missing constructor calls to the base class. Before the refactoring, there were no connection between super and base class, which is obviously bad. After placing the super call, the connection between the base and child class is created which affects the quality in the wrong way by increasing coupling.

The third patch which decreased quality is titled "`Removed duplicated function prepare_service()`". [13] This patch contained merging of two methods since they were almost exactly the same. We needed only one copy of the merged method, thus the other occurrence was deleted. Since the file contained only the deleted method, we got rid of the file entirely. Deleting the file caused the decrease in the quality as the deleted file was simple, contained only one method, which was also a small one.

On average, one small refactoring did an improvement of 0.00000852 in *Maintainability* (measured in the scale of [0,1]), which might not look that much, but refactoring the project regularly can cause a big quality improvement in the long term.

7 Threats to Validity

There is no exact definition for the quality of the source code expressed by a number. The QualityGate and SonarQube tools and the underlying ColumbusQM and SQALE technical debt models are only two of several approaches, with their own advantages and drawbacks. However, both are well-founded, stable, widely adapted models relied upon by industrial and research communities as well. Although we treat this as an external threat, we argue that the bias in their measurement has a limited impact on the presented results.

[11] `https://github.com/openstack/watcher/commit/a2750c7`
[12] `https://github.com/openstack/watcher/commit/67455c6`
[13] `https://github.com/openstack/watcher/commit/f0b58f8`

The sample set of refactorings we examined is quite small. Only a small team worked on producing these patches, thus the generalization of the results is a real threat. However, the refactoring work was carried out during years, thus despite the small number of developers, quite a high amount of effort has been put into it. This mitigates the threat.

We contributed only to the fraction of the OpenStack modules from which analyzed 2 altogether. It might happen that different modules show different results as there are more than 35 modules, thus for the complete generalization of our study, we plan to extend the number of analyzed modules.

It might also happen that the patches we've created are not refactoring patches. This threat can be mitigated as we followed the guidelines of both Refactoring [16] and Clean Code [23]. Another possible threat is that our patches contain poor refactorings that are not really fits into the open-source world. This is mitigated by the fact that at least 2 core members approved our patches before merge. Typically there were at least 5-6 reviewers who reviewed our patches.

8 Conclusions and Future Work

In this study, we investigated 47 refactoring patches in two projects from the worlds leading open source cloud operating system, OpenStack. We analyzed Nova and Watcher modules to find some quality issues which can be fixed. On these problematic parts, we applied various refactoring techniques, e.g. splitting a large method into numerous, smaller methods. The applied refactorings were analyzed in order to investigate whether refactoring really improves code maintainability. We used QualityGate to do the analysis, then results were collected, transformed and analyzed. During the final state of our research, we found out that one particular refactoring is often immensely small (i.e. often affects only a few lines). Out of 47 patches, only 8 produced measurable change in the code quality. The average improvement of all the patches were really small, but it was clearly an improvement and not degradation. Hence, refactoring the code regularly can improve the maintainability of the code in the mid and long-term.

Besides the measurable metrics the human factor is also very important. We got many comments [14,15,16] to our proposed patches that refactoring is always welcome and it is a nice thing that someone pays attention to code quality. In the future, we would like to extend this research by analyzing more patches on many more projects to generalize our results.

Acknowledgment

The project has been supported by the UNKP-17-4 New National Excellence Program of the Ministry of Human Capacities, Hungary.

The authors would also like to express their gratitude to Gergely Ladányi and Béla Vancsics for providing technical support in QualityGate analysis and OpenStack contribution as well as to Balázs Gibizer from Ericsson Hungary for his valuable guidance in the OpenStack development.

[14] `https://review.openstack.org/349582/`
[15] `https://review.openstack.org/247560`
[16] `https://review.openstack.org/263343/`

References

1. OpenStack Docs: hacking: OpenStack Hacking Guideline Enforcement. https://docs.openstack.org/hacking/latest/user/hacking.html
2. User talk: OpenStack Quality Improvement Plan. https://wiki.openstack.org/wiki/User_talk:P%C3%A9ter_Heged%C5%B1s
3. How To Contribute – OpenStack. https://wiki.openstack.org/wiki/How_To_Contribute (2018), [Online; accessed 09-April-2018]
4. OpenStack Docs: Modules List. https://docs.openstack.org/puppet-openstack-guide/latest/contributor/module-list.html (2018), [Online; accessed 09-April-2018]
5. Advani, D., Hassoun, Y., Counsell, S.: Extracting refactoring trends from open-source software and a possible solution to the 'related refactoring' conundrum. In: Proceedings of the 2006 ACM Symposium on Applied Computing. pp. 1713–1720. SAC '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1141277.1141685, http://doi.acm.org/10.1145/1141277.1141685
6. Bakota, T., Hegedűs, P., Siket, I., Ladányi, G., Ferenc, R.: Qualitygate sourceaudit: a tool for assessing the technical quality of software. In: Proceedings of the CSMR-WCRE Software Evolution Week. pp. 440–445. IEEE Computer Society (2014)
7. Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., Gyimóthy, T.: A probabilistic software quality model. In: Proceedings of the 27th International Conference on Software Maintenance (ICSM). pp. 243–252. IEEE Computer Society (2011)
8. Bakota, T., Hegedűs, P., Ladányi, G., Körtvélyesi, P., Ferenc, R., Gyimóthy, T.: A cost model based on software maintainability. In: Proceedings of the 28th International Conference on Software Maintenance (ICSM). pp. 316–325. IEEE Computer Society (2012)
9. Baset, S.A., Tang, C., Tak, B.C., Wang, L.: Dissecting open source cloud evolution: An openstack case study. In: HotCloud (2013)
10. Demeyer, S.: Refactor conditionals into polymorphism: what's the performance cost of introducing virtual calls? In: Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05. pp. 627–630. IEEE (2005)
11. Du Bois, B., Demeyer, S., Verelst, J.: Refactoring-improving coupling and cohesion of existing code. In: Proceedings of the 11th Working Conference on Reverse Engineering. pp. 144–151. IEEE (2004)
12. Du Bois, B., Mens, T.: Describing the impact of refactoring on internal program quality. In: Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications. pp. 37–48 (2003)
13. Erl, T., Puttini, R., Mahmood, Z.: Cloud Computing: Concepts, Technology & Architecture. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edn. (2013)
14. Ubuntu Cloud: OpenStack Wins, Eucalyptus Loses. http://talkincloud.com/ubuntu-cloud-openstack-wins-eucalyptus-loses
15. Fitfield, T.: Introduction to OpenStack. Linux J. **2013**(235) (Nov 2013), http://dl.acm.org/citation.cfm?id=2555789.2555793
16. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc. (1999)
17. Hegedűs, P., Kádár, I., Ferenc, R., Gyimóthy, T.: Empirical evaluation of software maintainability based on a manually validated refactoring dataset. Information and Software Technology (Nov 2017). https://doi.org/10.1016/j.infsof.2017.11.012, http://www.sciencedirect.com/science/article/pii/S0950584916303561, accepted, to appear.

18. ISO/IEC: ISO/IEC 9126. Software Engineering – Product quality 6.5. ISO/IEC (2001)
19. ISO/IEC: ISO/IEC 25000:2005. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. ISO/IEC (2005)
20. Kemerer, C.F., Paulk, M.C.: The impact of design and code reviews on software quality: An empirical study based on psp data. IEEE transactions on software engineering **35**(4), 534–550 (2009)
21. Kim, M., Gee, M., Loh, A., Rachatasumrit, N.: Ref-Finder: a Refactoring Reconstruction Tool Based on Logic Query Templates. In: Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering (FSE'10). pp. 371–372 (2010)
22. Letouzey, J.L.: The SQALE Method for Evaluating Technical Debt. In: Third International Workshop on Managing Technical Debt (MTD). pp. 31–36. IEEE (Jun 2012). https://doi.org/10.1109/MTD.2012.6225997
23. Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Pearson Education (2009)
24. McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E.: An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering **21**(5), 2146–2189 (Oct 2016)
25. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Transactions on Software Engineering **30**(2), 126–139 (2004)
26. Mirkalaei, M., Pooya, S.: API Failures in Openstack Cloud Environments. Ph.D. thesis, École Polytechnique de Montréal (2017)
27. Musavi, P., Adams, B., Khomh, F.: Experience report: An empirical study of api failures in openstack cloud environments. In: Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on. pp. 424–434. IEEE (2016)
28. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois (1992)
29. The OpenStack Foundation. `https://www.openstack.org/foundation`
30. OpenStack Releases. `https://releases.openstack.org/`
31. OpenStack Release models. `https://releases.openstack.org/reference/release_models.html`
32. Ubuntu Cloud Infrastructure. `https://help.ubuntu.com/community/UbuntuCloudInfrastructure`
33. OpenStack. `https://en.wikipedia.org/wiki/OpenStack`
34. Sahraoui, H.A., Godin, R., Miceli, T.: Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In: Proceedings of International Conference on Software Maintenance. pp. 154–162. IEEE (2000)
35. Simon, F., Steinbruckner, F., Lewerentz, C.: Metrics based refactoring. In: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering. pp. 30–38. IEEE (2001)
36. Slipetskyy, R.: Security issues in OpenStack. Master's thesis, Institutt for telematikk (2011)
37. Stroulia, E., Kapoor, R.: Metrics of refactoring-based development: An experience report. In: OOIS 2001, pp. 113–122. Springer (2001)
38. Szőke, G., Antal, G., Nagy, C., Ferenc, R., Gyimóthy, T.: Empirical study on refactoring large-scale industrial systems and its effects on maintainability. Journal of Systems and Software **129**(C), 107–126 (Jul 2017). https://doi.org/10.1016/j.jss.2016.08.071, `http://www.sciencedirect.com/science/article/pii/S0164121216301558?via%3Dihub`