

STATE OF PRACTICE FOR MEDICAL IMAGING SOFTWARE

ASSESSING THE CURRENT STATE OF THE PRACTICE FOR MEDICAL IMAGING
SOFTWARE

By
AO DONG.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the degree
Master of Engineering in Computing and Software

McMaster University

© Copyright by Ao Dong, August 2021

MASTER OF ENGINEERING (2021)
(Computing and Software)

McMaster University
Hamilton, Ontario

TITLE: Assessing the Current State of the Practice for Medical Imaging Software

AUTHOR: Ao Dong

SUPERVISOR: Dr. Spencer Smith

NUMBER OF PAGES: viii, ??

Abstract

Abstract here

Acknowledgments

acknowledgements here

Contents

Abstract	iii
Acknowledgments	iv
1 Introduction	2
1.1 Motivation	2
1.2 Purpose	2
1.3 Scope	2
2 Background	3
2.1 Software Categories	3
2.1.1 Open Source Software	4
2.1.2 Freeware	4
2.1.3 Commercial Software	5
2.1.4 Scientific Computing Software	5

2.2	Software Quality Definitions	6
2.3	Analytic Hierarchy Process	7
3	Methods	10
3.1	Domain Selection	10
3.2	Software Product Selection	12
3.2.1	Identify Software Candidates	12
3.2.2	Filter the Software List	12
3.3	Grading Software	13
3.3.1	Grading Template	14
3.3.2	Empirical Measurements	15
3.3.3	Technical Details	16
3.4	Interview Methods	17
3.4.1	Interviewee Selection	17
3.4.2	Interview Question Selection	17
3.4.3	Interview Process	18
3.5	An Example of Applying the Method	18
4	Measurement Results	21
4.1	Overall Scores	23
4.2	Installability	24

4.3	Correctness & Verifiability	26
4.4	Surface Reliability	28
4.5	Surface Robustness	29
4.6	Surface Usability	31
4.7	Maintainability	32
4.8	Reusability	34
4.9	Surface Understandability	35
4.10	Visibility/Transparency	36
5	Interviews with Developers	38
5.1	Summary of Answers	38
5.2	Discussions	38
6	Threat to Validity	39
7	Recommendations	40
8	Conclusions	41
	Bibliography	42
A	Full Grading Template	48
B	Summary of Measurements	49

C Interview Answers	50
D Ethics Approval	51

Chapter 1

Introduction

introduction

scientific computing (SC), also known as scientific computation or computational science

1.1 Motivation

1.2 Purpose

1.3 Scope

Chapter 2

Background

This chapter introduces several different categories of software, based on which we designed the processes to select the software domain and software candidates in Chapter 3. It also covers the software quality definitions used in the grading template in Appendix A and an overview of the Analytic Hierarchy Process (AHP), a tool we used to compare and grade software products.

2.1 Software Categories

We usually target specific software categories to narrow down the scopes when selecting software domains and software packages. In this section, we discuss three common software categories that are mentioned in Section 3.2, and also SC software. Section 3.1 and Section 3.2 present more details about why we prefer some of these categories.

2.1.1 Open Source Software

For Open Source software (OSS), its source code is openly accessible, and users have the right to study, change and distribute it under a license granted by the copyright holder. For many OSS projects, the development process relies on the collaboration of different contributors worldwide [6]. Accessible source code usually exposes more “secrets” of a software project, such as the underlying logic of software functions, how developers achieve their works, and the flaws and potential risks in the final product. Thus, it brings much more convenience to the researchers analyzing the qualities of the project.

2.1.2 Freeware

Freeware is software that can be used free of charge. Unlike with OSS, the authors of freeware typically do not allow users to access or modify the source code of the software [23]. The term *freeware* should not be confused with *free software*, which is similar to OSS but with a few differences. To the end-users, the differences between freeware and OSS often do not bother them. The fact that these products are free of charge is likely to make them popular with many users. However, software developers, end-users who wish to modify the source code, and researchers looking for inner characteristics may find the inaccessible source code to be a problem.

2.1.3 Commercial Software

“Commercial software is software developed by a business as part of its business” [9]. Typically speaking, the users are required to pay to access all of the features of commercial software, excluding access to the source code. However, some commercial software is also free of charge [9]. Based on our experience, most commercial software products are not OSS.

For some specific software, the backgrounds of commercial software developers often differ from the ones of non-commercial OSS. In such a case, the former is usually the product of software engineers, and the latter is likely to have developers who work in the domain and are also end-users of the products. One example is software in Scientific Computing (SC), since the developers need to utilize their domain-specific during the development process [31].

2.1.4 Scientific Computing Software

Software development in SC depends on the knowledge of three areas - the inside of a specific engineering or science domain, the ability to mathematically build models and apply algorithms, and to implement theoretical models and algorithms with computational tools. SC software is built with mathematical and computational tools to serve the purpose of solving scientific problems in a domain [21]. However, the majority of scientists developing their software are self-taught programmers [31], so there may be a bigger Room for

software quality improvement in SC domains.

2.2 Software Quality Definitions

The definitions of software qualities are from Smith et al. [28]. The order of the qualities follows the grading template in Appendix A.

- **Installability** The effort required for the installation, uninstallation, or reinstallation of a software or product in a specified environment.
- **Correctness & Verifiability** A program is correct if it behaves according to its stated. Verifiability is the extent to which a set of tests can be written and executed, to demonstrate that the delivered system meets the specification.
- **Reliability** The probability of failure-free operation of a computer program in a specified environment for a specified time, i.e. the average time interval between two failures also known as the mean time to failure (MTTF).
- **Robustness** Software possesses the characteristic of robustness if it behaves “reasonably” in two situations: i) when it encounters circumstances not anticipated in the requirements specification, and ii) when the assumptions in its requirements specification are violated.
- **Usability** The extent to which a product can be used by specified users to achieve

specified goals with effectiveness, efficiency, and satisfaction in a specified context of use.

- **Maintainability** The effort with which a software system or component can be modified to i) correct faults; ii) improve performance or other attributes; iii) satisfy new requirements.
- **Reusability** The extent to which a software component can be used with or without adaptation in a problem solution other than the one for which it was originally developed.
- **Understandability** (To be completed)
- **Visibility/Transparency** The extent to which all of the steps of a software development process and the current status of it are conveyed clearly.

2.3 Analytic Hierarchy Process

To generate grading scores for a group of software packages, we use the AHP to pairwise compare them. Thomas L. Saaty developed this tool, and people widely used it to make and analyze multiple criteria decisions [30]. The AHP organizes multiple criteria factors in a hierarchical structure and pairwise compares the alternatives to calculate relative ratios [26].

For a project with m criteria, we can use a $m \times m$ matrix A to record the relative importance between factors. By pairwise compare criterion i and criterion j , the value of A_{ij} is decided as follows, and the value of A_{ji} is $1/A_{ij}$ [26],

- $A_{ij} = 1$ if criterion i and criterion j are equally important;
- $A_{ij} = 9$ if criterion i is extremely more important than criterion j ;
- A_{ij} equals to an integer value between 1 and 9 according the the relative importance of criterion i and criterion j .

The above process assumes that criterion i is not less important than criterion j , otherwise, we need to reverse i and j and determine A_{ji} first, then $A_{ij} = 1/A_{ji}$.

The priority vecotr w can be calculated by solving the following equation [26],

$$Aw = \lambda_{max}w, \quad (2.1)$$

where λ_{max} is the maximal eigenvalue of A .

In this project, w is approximated with the approach classic *mean of normalized values* [14],

$$w_i = \frac{1}{m} \sum_{j=1}^m \frac{A_{ij}}{\sum_{k=1}^m A_{kj}} \quad (2.2)$$

Suppose there are n alternatives, for criterion $i = 1, 2, \dots, m$, we can create an $n \times n$ matrix B_i to record the relative preferences between these choices. The way of generating

B_i is similar to the one for A . However, unlike comparing the importance between criteria, we pairwise decide how much we favor one alternative over the other. We use the same method to calculate the local priority vector for each B_i .

In this project, the 9 software qualities mentioned above are the criteria ($m = 9$), while 29 software packages ($n = 29$) are compared. The software are evaluated with the grading template in Appendix A and a subjective score is given for each quality. For a pair of qualities or software, i and j , such that i is not less significant than j , the pairwise comparison result of i versus j is converted from $\min((score_i - score_j) + 1, 9)$.

Chapter 3

Methods

We designed a general process for evaluating the state of the practice of domain-specific software, that we instantiate to SC software for specific scientific domains.

Our method involves: 1) choosing a software domain (Section 3.1); 2) collecting and filtering software packages (Section 3.2); 3) grading the selected software (Section 3.3); 4) interviewing development teams for further information (Section 3.4).

Details of how we applied the method on the MI domain are expanded upon in Section 3.5.

3.1 Domain Selection

Our methods are generic, but our scope only included scientific domains due to the objective of our research, and thus we have only applied this method to scientific domains.

When choosing a candidate domain, we prefer a domain with a large number of active OSS. The reason is that we aim to finalize a list of 30 software packages [29] in the domain after screening out the ones not meeting our requirements, such as the ones without recent updates or specific functions. Thus the quantity of final-decision packages may not meet our initial expectation if we do not have enough software candidates. Another reason is that our grading method works much better for OSS. Moreover, we need the domain to have an active community developing and using SC software, making it easier to conduct interviews with developers.

With an adequate number of software packages in a selected domain, we may still find the software products with various purposes and fall into different sub-categories. So we should ask one question to ourselves - do we prefer a group of software all providing similar functions and features, or do we aim to cross-compare several sub-sections within the same domain? With that answered, we can better determine a favored domain.

Another aspect to consider is the team carrying out the research, specifically the domain experts - if there is any - in the team. In our team, researchers in the software engineering field usually lead the projects, and experts working in scientific domains support them. Having domain experts in the group provides significant benefits in selecting software packages and designing interview questions.

3.2 Software Product Selection

The process of selecting software packages contains two steps: i) identify software candidates in the chosen domain, ii) filter the list according to needs [29].

3.2.1 Identify Software Candidates

We can find candidate software in publications of the domain. Another source is to search various websites, such as GitHub, swMATH and the Google search results for software recommendation articles. Meanwhile, we should also include the suggested ones from domain experts [29].

3.2.2 Filter the Software List

The goal is to build a software list with a length of about 30 [29].

The only “mandatory” requirement is that the software must be OSS, as defined in Section 2.1.1. Because to evaluate all aspects of some software qualities, the source code should be accessible.

The other factors to filter the list can be optional, and we should consider them according to the number of software candidates and the objectives of the research project.

One of the factors is the functions and purposes of the software. For example, we can choose a group of software with similar functions, so that the cross-comparison is between

each individual of them. On the other hand, if our objective is to compare sub-categories in the domain, we should select candidates from each of the categories.

The empirical measurement tools listed in Section 3.3.2 have limitations that we can only apply them on projects using Git as the version control tool, so we prefer software with Git. Some manual steps in empirical measurement depend on a few metrics of GitHub, which makes projects held on GitHub more favored [29].

Some of the OSS projects may experience a lack of recent maintenance. So we can eliminate packages without recent updates, unless they are still popular and highly recommended by the domain users [29].

Whether we set what standards to filter the packages, with domain experts in the team, we should value their opinions. For example, if a software package is not OSS and has no updates for a long while, the domain experts may still identify it as a popular and widely used product. In this case, perhaps it is valuable to keep this software on the list.

3.3 Grading Software

We grade the selected software using a template (Section 3.3.1) and a specific empirical method (Section 3.3.2). Some technical details for the measurements are in Section 3.3.3.

3.3.1 Grading Template

The full grading template can be found in Appendix A. The template contains 101 questions that we use for grading software products.

We use the first section of the template to collect some general information about the software, such as the name, purpose, platform, programming language, publications about the software, the first release and the most recent change date, website, and source code repository of the product, etc. Information in this section helps us understand the projects better and may be helpful for further analysis, but it does not directly affect the grading scores for the packages.

We designed the following 9 sections in the template for the 9 software qualities mentioned in Section 2.2. For each quality, we ask several questions and the typical answers are among the choices of “yes”, “no”, “n/a”, “unclear”, a number, a string, a date, a set of strings, etc. Each quality needs an overall score between 1 and 10 based on all the previous questions. For some qualities, the empirical measurement sections also affect this grading score.

All the last three sections on the template are about the empirical measurements. We use two command-line software tools *GitStats* and *scc* to extract information about the source code from the project repositories. For projects held on GitHub, we manually collect additional metrics, such as the stars of the GitHub repository, and the numbers of open and closed pull requests. Section 3.3.2 presents more details of how these empirical mea-

surements affect software quality grading scores.

3.3.2 Empirical Measurements

We use two command-line tools for the empirical measurements. One is *GitStats* that generates statistics for git repositories and display outputs in the format of web pages [8]; the other one is Sloc Cloc and Code (as known as *scc*) [4], aiming to count the lines of code, comments, etc.

Both tools can measure the number of text-based files in a git repository and lines of text in these files. Based on our experience, most text-based files in a software project repository contain programming source code, and developers use them to compile and build software products. A minority of these files are instructions and other documents. So we roughly regard the lines of text in text-based files as lines of programming code. The two tools usually generate similar but not identical results as for the above measurements. From our understanding, this minor difference is because of the different techniques that they use to detect if a file is a text-based or binary file.

Additionally, we also manually collect some information for projects held on GitHub, such as the numbers of stars, forks, people watching this repository, open pull request, and closed pull request.

These empirical measurements help us from two aspects. Firstly, with more statistical details, we can get an overview of a project faster and more accurately. For example, the

number of commits over the last 12 months shows how active this project has been during this period, and the number of stars and forks may reveal its popularity. Secondly, the results may affect our decisions regarding the grading scores for some software qualities. For example, if the percentage of comment lines is low, we might want to double-check the understandability of the code. On the other hand, if the ratio of open versus closed pull requests is high, perhaps we should pay more attention to the project's maintainability.

3.3.3 Technical Details

To test the software on a “clean” system, we created a new virtual machine (VM) for each software and only installed the necessary dependencies before measuring. We created All 29 VM on the same computer. We only start the measuring of the following software after finishing a current one, and after grading each software, we destroy the VM.

Generally speaking, we spend about two hours grading one software, unless there are technical issues and more time is needed to resolve them. In most of the situation, we finish all the measurements for one software on the same day.

3.4 Interview Methods

3.4.1 Interviewee Selection

For a software list with a length of roughly 30, we aim to interview about 10 development teams of these projects. Interviewing multiple individuals from each team should give us more comprehensive information. Still, if it is challenging to find various willing participants, a single engineer well-knowing the project can also be sufficient.

Ideally, we select projects after the grading measurements and prefer the projects with higher overall scores. However, if we do not find enough participants, we should also contact all teams on the list.

We try to find the contacts of the teams on the websites related to the software, such as the official web pages, repository websites, publication websites, and bio pages of the teams' institutions. Then, we send at most two emails asking for their support and participation before receiving any replies for each candidate.

3.4.2 Interview Question Selection

We have a list of 20 questions to guide our interviews with the development teams, which can be found in Appendix C.

Some questions are for the background of the software, the development teams, the interviewees, and how they organize the projects. We also ask about their understandings

of the users. Another part of the interview questions focuses on the significant difficulties the team experience currently or in the past, and the solutions they have found or will try. We also discuss the importance of documents to their projects and the current situations of these documents with interviewees. A few questions are about specific software qualities, such as *maintainability*, *understandability*, *usability*, and *reproducibility*.

The interviews are supposed to be semi-structured based on the question list, and we also ask follow-up questions when necessary. Based on our experience, the interviewees usually bring up some exciting ideas that we do not expect, and we can expand on these topics for a few more details.

3.4.3 Interview Process

Since the members of the development teams are likely to be based around the world, so we organize these interviews as virtual meetings online with Zoom. After receiving consent from the interviewees, we also record our discussions to transcribe them better.

3.5 An Example of Applying the Method

This section shows how we applied our method in Section 3 on the Medical Imaging (MI) domain.

Based on the principles in Section 3.1, we selected the MI domain, because there are

numerous software products and a significant number of professionals use and develop such software in this domain. We decided to focus on MI software with the viewing function, so we needed many software candidates in the domain. Being able to include MI domain experts in our team also made this domain more preferred.

By using the method in Section 3.2.1, we identified 48 MI software projects as the candidates from publications [3] [5] [10], online articles related to the domain [7] [13] [22], forum discussions related to the domain [27], etc.

Among the 48 software packages, there were several ones that we could not find their source code, such as MicroDicom, Aliza, and jivex, etc. These packages are likely to be freeware defined in Section 2.1.2 and not OSS. So following guidelines in Section 3.2.2 we removed them from the list. Next, we focused on the MI software that could work as a MI viewer. Some of the software on the list were tool kits or libraries for other software to use as dependencies but not for end-users to view medical images, such as VTK, ITK, dcm4che, etc. We also eliminated these from the list. After that, there were 29 software products left on the list. We still preferred projects using git and GitHub and being updated recently, but did not apply another filtering since the number of packages was already below 30. However, 27 out of the 29 software packages on the final list used git, and 24 chose GitHub. Furthermore, 27 packages had the latest updates after the year 2018, and 23 after 2020.

Then we followed the steps in Section 3.3 to measure and grade the software. 27 out

of the 29 packages are compatible with two or three different operating systems such as Windows, macOS, and Linux, and 5 of them are browser-based, making them platform-independent. However, in the interest of time, we only performed the measurements for each project by installing it on one of the platforms, most likely Windows.

Going through the interview process in Section 3.4, we started with the teams with higher scores on our list, and eventually contacted all of them. As a result, developers/architects from 8 teams have participated in our interviews so far. Before contacting any interviewee candidate, we received ethics clearance from the McMaster University Research Ethics Board (Appendix D).

Chapter 4

Measurement Results

A summary of the measurement results can be found in Appendix B, and the detailed data is in the repository <https://github.com/smiths/AIMSS>.

All of the 29 software packages that we measured are OSS with functions of displaying medical images. The initial release dates of these packages are between the years 1998 and 2018. We considered 2 of them as “dead” projects since they had the latest updates more than 18 months earlier than the time of measuring, but both had updates after 2017.

We found out that at least 8 of the projects received funding for at least a certain period, but we were not sure about the rest of them due to limited information.

After analyzing the repositories and official websites of the projects, we found out that the number of contributors for a project varied between 1 to roughly 100. We considered anyone who made at least one accepted commit to the source code as a contributor, so

it does not mean that any development teams had long-term members as many as 100. Many of these projects received change requests and code from the community, such as pull requests and git commits on GitHub. At least 27 of the software projects had code commits from more than a single developer, and as far as we could find out, at least 13 of them had no less than 10 contributors.

According to our measurements, 27 out of the 29 packages supported multiple operating systems, and 25 of them could work on all three Windows, macOS, and Linux systems. However, there was a significant difference in the philosophy to achieve cross-platform compatibility. The majority of the 27 were desktop software products built for each operating system separately. On the other hand, 5 were web applications, which made them naturally platform-independent and compatible with almost all operating systems with compatible browsers. There are certain advantages and disadvantages to developing either a native or a browser-based application, and some of the details can be found from the interview answers in Section 5.

Most of the projects used more than one programming language, including a primary language that the developers used the most. The most popular primary language was C++, chosen by 11 projects, and another 3 teams selected C. The second most used one was JavaScript, with 6 projects using it as the primary language. 4 of the projects chose Java, and 3 selected Python.

4.1 Overall Scores

As described in Section 2.3, for our AHP measurements, there are 9 criteria which are the 9 software qualities and 29 software packages as the alternatives. We decided to make all 9 qualities equally important, so the score of each quality affects the overall scores on the same scale.

Figure 4.1 shows the overall scores of all 29 software packages in descending order. Since we produced the scores from the AHP process, the total sum of the 29 scores is precisely 1.

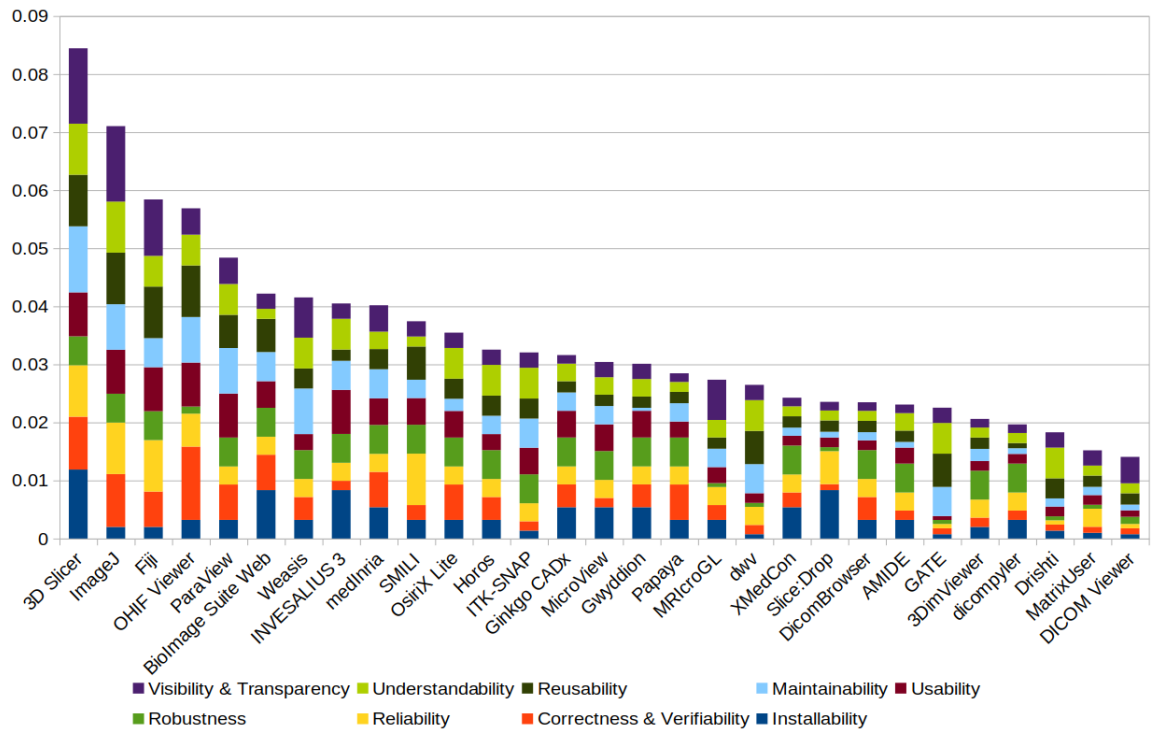


Figure 4.1: Overall AHP scores for all 9 software qualities

The top two software products had higher scores in each criterion. For example, *3D Slicer* [16] ranked at the first or second place for 8 of the 9 qualities; *ImageJ* [25] was within the top three products for 7 qualities. There were 3 software packages that we could not install or build correctly. Among them, *DICOM Viewer* [1] was the only one that did not have an online test version, so that we could not finish all measurements for it. Consequently, we might underestimate its score.

4.2 Installability

When measuring the *installability*, we checked the existence and quality of installation instructions. The user experience was also an important factor, such as the ease to follow the instructions, number of steps, automation tools, and the prerequisite steps for the installation. If any problem interrupted the process of installation or uninstallation, we gave a lower score to this quality. We also recorded the operating system for the installation test and whether we could verify the installation.

Figure 4.2 lists the scores of *installability*, with a total sum of 1.

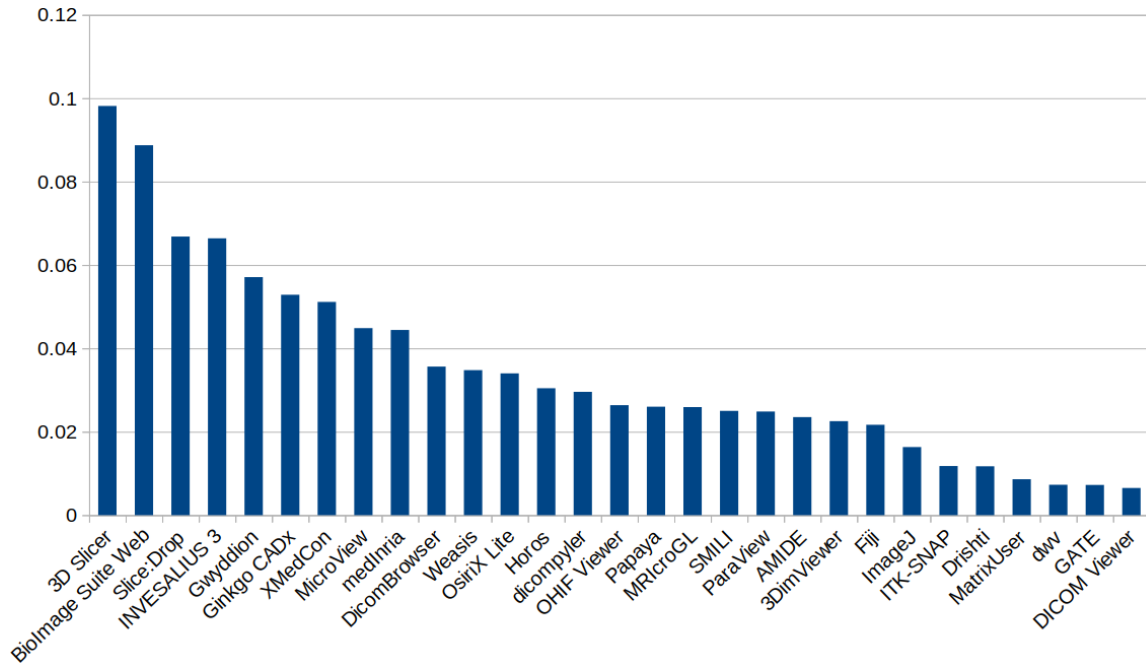


Figure 4.2: AHP installability scores

The projects with higher scores had easy-to-follow installation instructions, and the installation processes were automated, fast, and frustration-free, with all dependencies automatically added. There were also no errors during the installation and uninstallation steps.

The ones with the lowest scores often showed severe problems. We could not install some of them due to some issues that we could not solve. We spent a reasonable amount of time on these problems, then considered them major obstacles for normal users if we still did not figure out any solutions. For example, we needed to build some software from

the source code, and we failed the building for two of them following the instructions. Furthermore, *DICOM Viewer* depended on a cloud platform, and we could not successfully install the dependency. We suspect that only a part of the users faced the same problems, and given a lot of time, we might be able to find solutions. However, the difficulties greatly impacted the installation experiences, and we graded these software packages with lower scores.

Although we could not locally build two software packages, there were deployed online versions for them. With that, we finished all the measurements for them.

4.3 Correctness & Verifiability

For *correctness*, we checked the projects to identify the techniques to ensure this quality, such as literate programming, automated testing, symbolic execution, model checking, unit tests, etc. We also examined that whether the projects used continuous integration. As for *verifiability*, we went through the documents of the project to check the requirements specifications, theory manuals, and getting started tutorial. If a getting started tutorial existed and provided expected results, we followed it and checked if the outputs match.

The scores of *correctness & verifiability* are shown in Figure 4.3. Generally speaking, the packages with higher scores adopted more techniques to improve *correctness*, and had better documents for us to verify it.

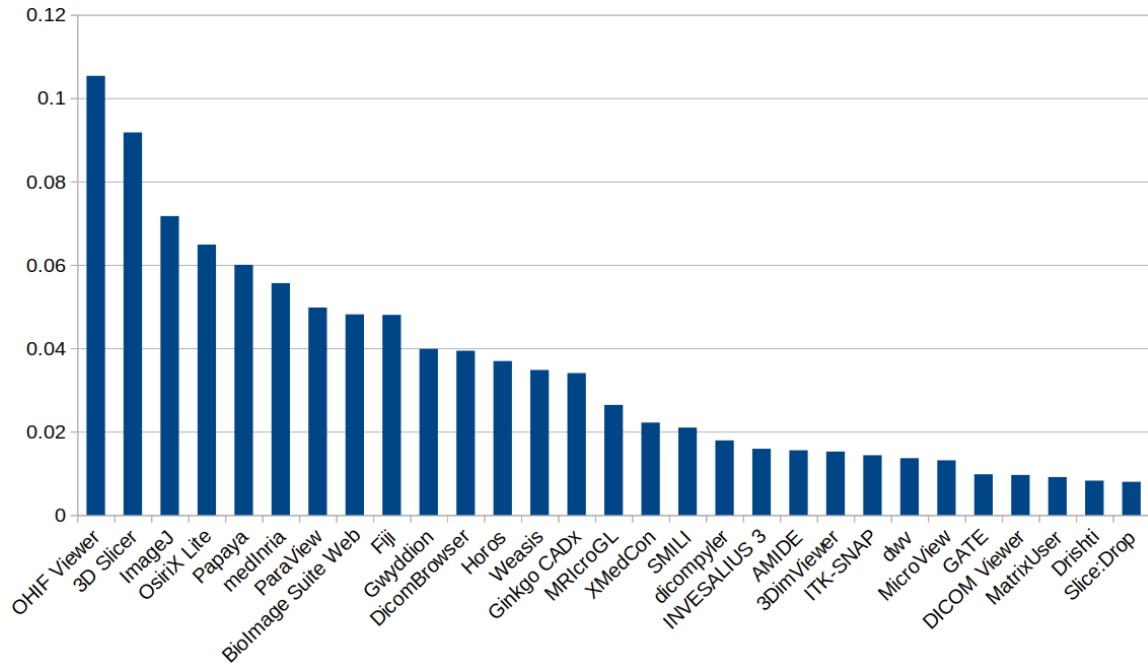


Figure 4.3: AHP correctness & verifiability scores

After examining the source code, we could not find any evidence of unit testing in more than half of the projects. Unit testing benefits most parts of the software's life cycle, such as designing, coding, debugging, and optimization [12]. It can reveal the bugs at an earlier stage of the development process, and the absence of unit testing may cause worse *correctness & verifiability*. We also could not identify the requirements specifications or theory manuals for most of the projects. It seems that even for some projects with well-organized documents, requirements specifications and theory manuals were still missing.

4.4 Surface Reliability

For *Surface Reliability*, we checked that whether the software broke during the installation and tutorial, whether there were descriptive error messages, and if we could recover the process after the errors.

As described in Section 4.2, we could not build two software packages, however, since there were online versions of them, which might be proof that successful deployments were possible. So the failure of installation did not affect their scores in *Surface Reliability*.

Figure 4.4 shows the AHP results.

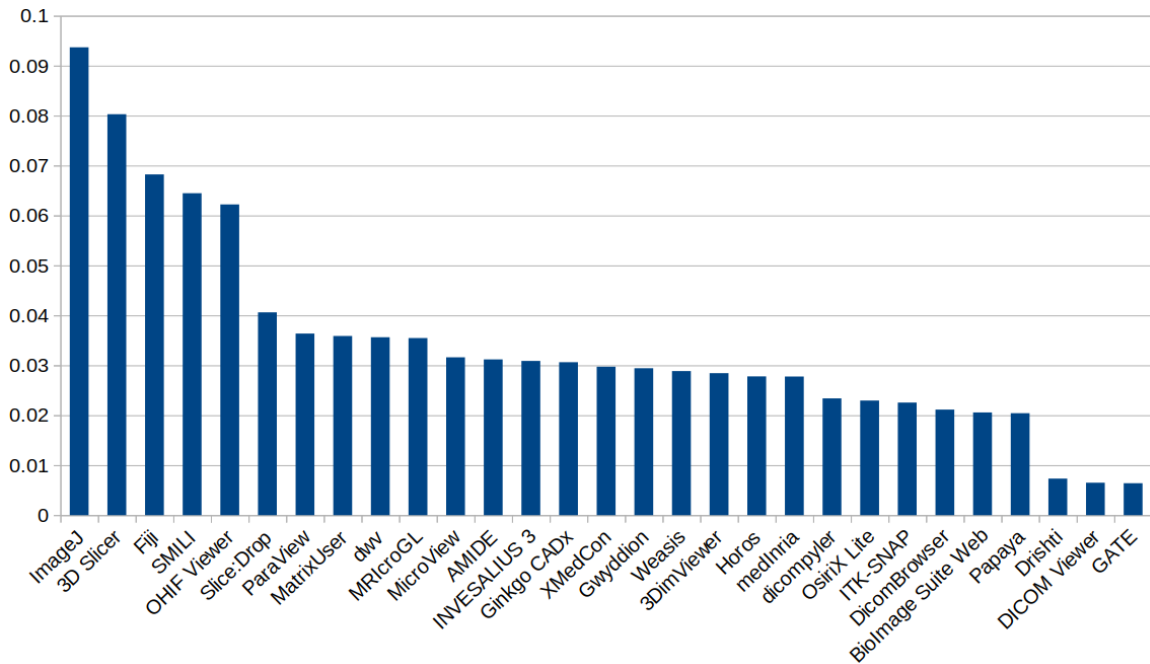


Figure 4.4: AHP surface reliability scores

For the software with the lowest scores, *Drishiti* [18] and *GATE* [15] crashed during our operations, and we might underestimated the score of *DICOM Viewer* since we could not install or test it.

4.5 Surface Robustness

To test *Surface Robustness*, we checked that how the software handle unexpected/unanticipated input. Since all 29 MI software packages had functions to load image files, we prepared broken image files for the software to open. For example, a text file (.txt) with a modified extension name (.dcm) can be used as an unexpected/unanticipated input. We first tried to load a few correct input files with the software to ensure the function was working correctly, then with the unexpected/unanticipated ones. Figure 4.5 presents the scores.

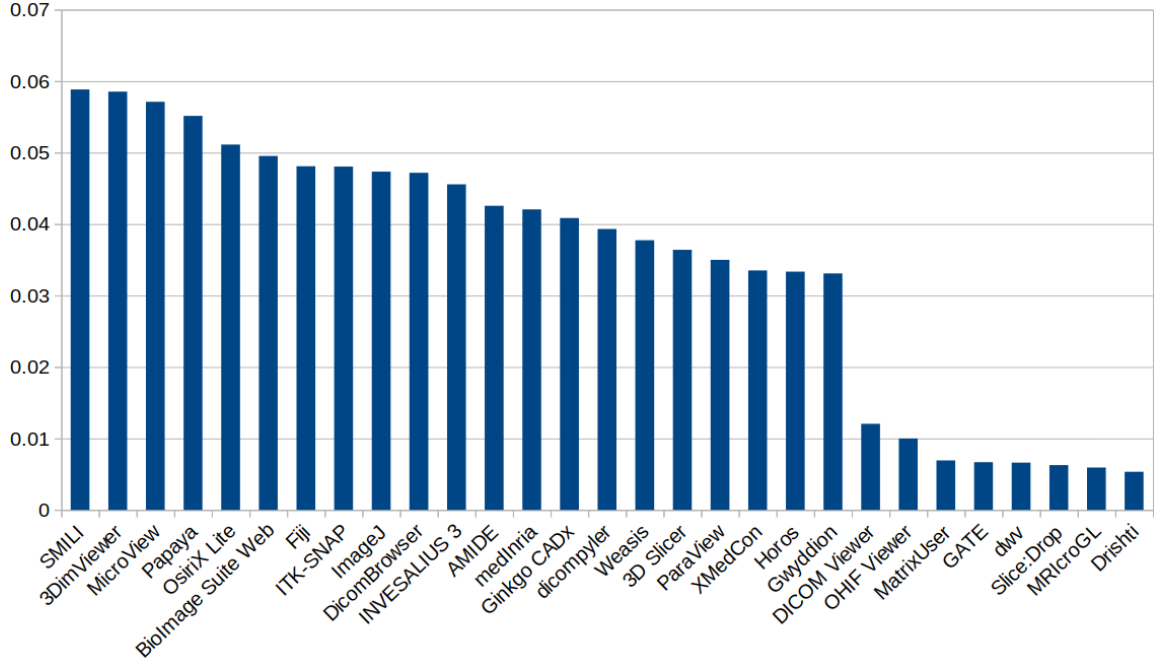


Figure 4.5: AHP surface robustness scores

The packages with higher scores elegantly handled the unexpected/unanticipated inputs, normally by showing a clear error message. Again, we could not test *DICOM Viewer*. We also might underestimate the score of *OHIF Viewer* [32] since we needed further customization to load data and the test was not complete. *MatrixUser* [19], *dvw* [20], *Slice:Drop* [11], and *MRICroGL* [17] ignored the incorrect format of the input files, and displayed blank or meaningless images. *Drishti* successfully detected the unexpected/unanticipated inputs, but the software crashed as a result. For unknown reasons, *GATE* failed to load both correct and incorrect inputs.

4.6 Surface Usability

We measured *Surface Usability* by examining the documents of the projects and considered software with a getting started tutorial and a user manual easier to use. Meanwhile, we checked if users had any channels to get supports. Our impressions and user experiences when testing the software also affected the scores a lot. For example, easy-to-use graphical user interfaces gave us better experiences.

Figure 4.6 lists the AHP scores for all packages. The ones with higher scores usually provided both comprehensive document guidance and a good user experience. *INVESAL-IUS 3* [2] set an good example of detailed and clear user manual. *GATE* also provided a large number of documents, but we think that they conveyed the ideas poorly, so we had trouble understand and use them.

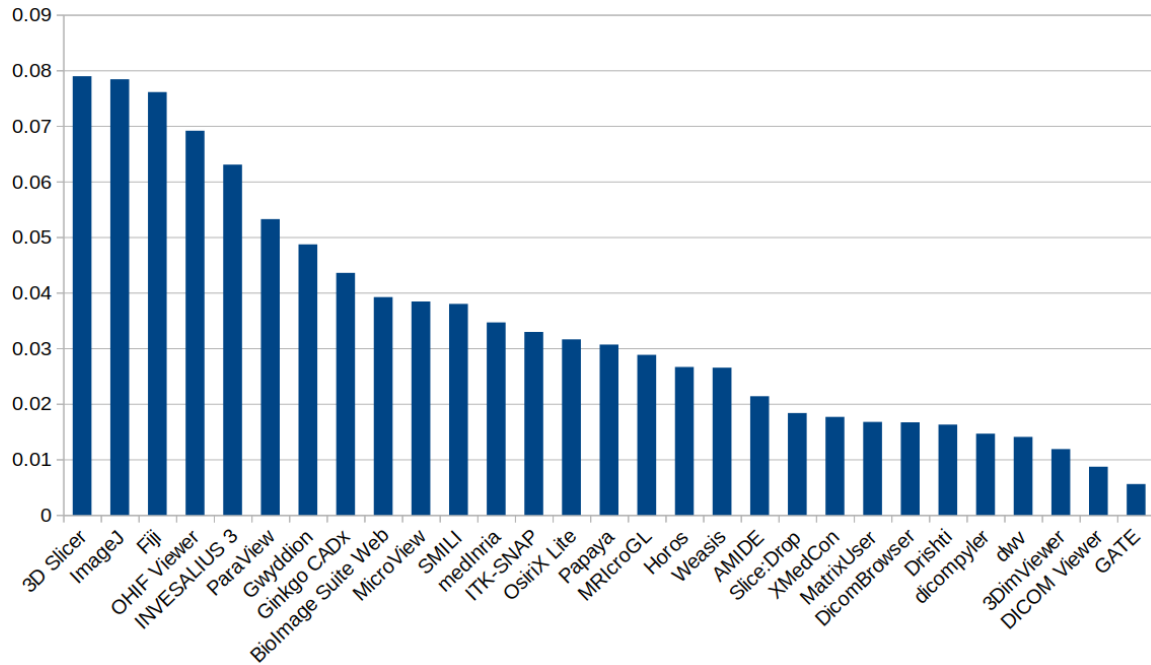


Figure 4.6: AHP surface usability scores

4.7 Maintainability

Regarding *maintainability*, we tried to search the projects' documents and identify the process of contributing and reviewing code. We believe that the artifacts of a project - including source code, documents, building scripts, etc. - can significantly affect its *maintainability*. Thus we checked each project for its artifacts, such as API documentation, bug tracker, release notes, test cases, build files, version control, etc. We also checked which tools supported issue tracking and version control and the percentages of closed issues and comment

lines in code. Figure 4.7 shows the AHP results for *maintainability*.

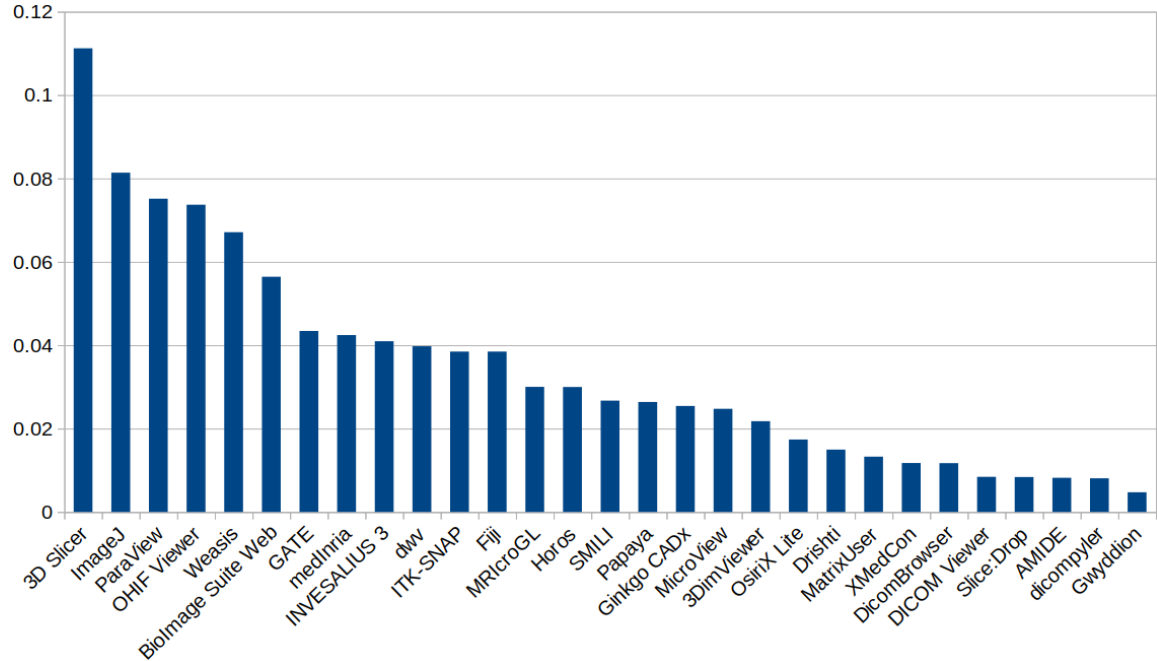


Figure 4.7: AHP maintainability scores

We marked *3D Slicer* with a much higher score than others because it did very well at closing the identified issues, and more importantly, we found it to have the most comprehensive artifacts. For example, as far as we could find out, only 5 out of the 29 projects had a project plan, 6 of them had a developer's manual, and 7 with API documentation, and *3D Slicer* included all these documents.

4.8 Reusability

To measure *reusability*, we counted the total number of code files for each project. We considered the projects with more code files to be more reusable. As mentioned in Section 3.3.2, the tools *GitStats* and *scc* count this number with minor differences, and we used the result from *GitStats* for all projects. We also decided that the projects with API documentation can deliver better *Reusability*. The AHP results are in Figure 4.8.

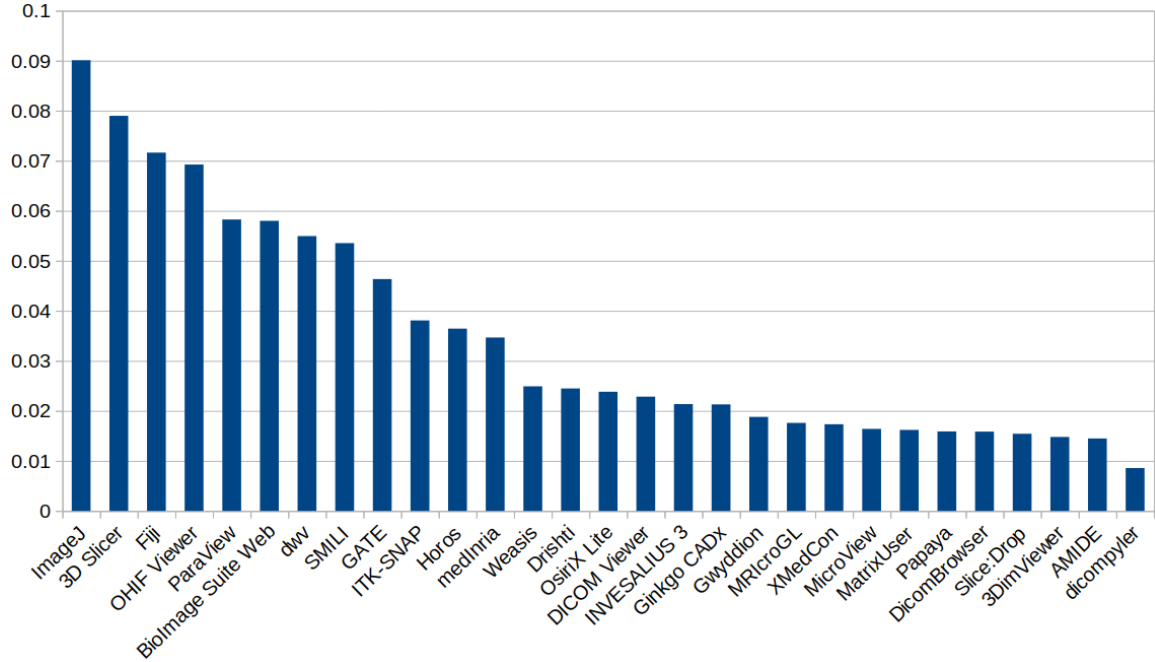


Figure 4.8: AHP reusability scores

4.9 Surface Understandability

To determine the *surface understandability* for each project, we randomly examined 10 code files. We checked the code's style within each code file, such as whether the identifiers, parameters, indentation, and formatting were consistent, whether the constants (other than 0 and 1) were hardcoded into the code if the developers modularized the code. We also checked the descriptive information for the code, such as documents mentioning the coding standard, the comments in the code, and the descriptions or links for algorithms used in the code. Figure 4.9 shows the scores.

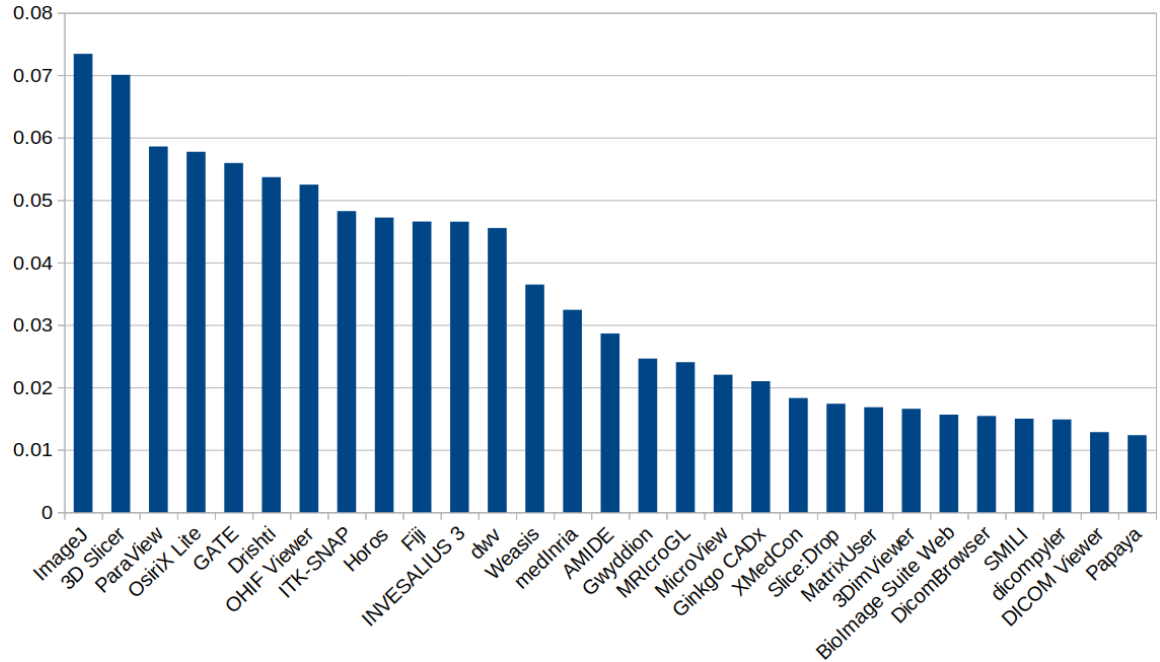


Figure 4.9: AHP surface understandability scores

Most projects had a consistent style for the code, but we only found explicit identification of a coding standard for only 3 out of the 29, which are *3D Slicer*, *Weasis* [24], and *ImageJ*.

4.10 Visibility/Transparency

To understand the *visibility/transparency*, such as all of the steps of a software development process and the current status described in Section 2.2, we tried to access this information from the existing documents. We tried to examine the development process, current status, development environment, and release notes for each project. If any information was missing or poorly conveyed, the *visibility/transparency* was not ideal.

Figure 4.10 shows the AHP scores. Generally speaking, the teams that actively documented their development process and plans scored higher because they delivered better communication to people outside the team.

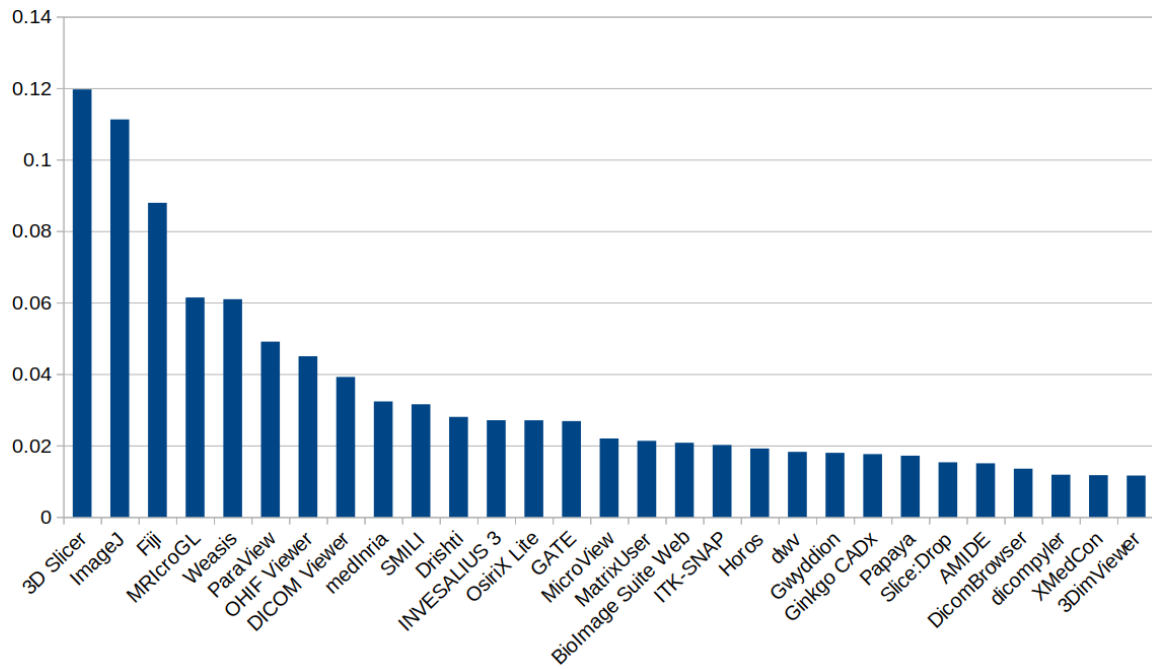


Figure 4.10: AHP visibility/transparency scores

Chapter 5

Interviews with Developers

5.1 Summary of Answers

- Start with one by one, with commonalities and interesting special cases.
- Shorten and summarize later.

5.2 Discussions

Any conclusions?

Chapter 6

Threat to Validity

- limited time to each software, surface quality
- impressions, subjective
- searched all information by ourselves, so maybe missed some info
- mainly measured by one person, so maybe biased

Chapter 7

Recommendations

I think the recommendations can originate from both parts - measurements and interviews.

Chapter 8

Conclusions

No clues yet. Should be started at a later stage.

Bibliography

- [1] Aysel Afsar. Dicom viewer. <https://github.com/ayselafsar/dicomviewer>, 2021. [Online; accessed 27-May-2021].
- [2] Paulo Amorim, Thiago Franco de Moraes, Helio Pedrini, and Jorge Silva. Invesalius: An interactive rendering framework for health care support. page 10, 12 2015.
- [3] Kari Björn. Evaluation of open source medical imaging software: A case study on health technology student learning experience. *Procedia Computer Science*, 121:724–731, 01 2017.
- [4] Ben Boyter. Sloc cloc and code. <https://github.com/boyter/scc>, 2021. [Online; accessed 27-May-2021].
- [5] Andreas Brühshwein, Julius Klever, Anne-Sophie Hoffmann, Denise Huber, Elisabeth Kaufmann, Sven Reese, and Andrea Meyer-Lindenberg. Free dicom-viewers for veterinary medicine: Survey and comparison of functionality and user-friendliness of

medical imaging pacs-dicom-viewer freeware for specific use in veterinary medicine practices. *Journal of Digital Imaging*, 03 2019.

- [6] James Edward Corbly. The free software alternative: Freeware, open source software, and libraries. *Information Technology and Libraries*, 33(3):65–75, Sep. 2014.
- [7] Steve Emms. 16 best free linux medical imaging software. <https://www.linuxlinks.com/medicalimaging/>, 2019. [Online; accessed 02-February-2020].
- [8] Tomasz Gieniusz. Gitstats. https://github.com/tomgi/git_stats, 2019. [Online; accessed 27-May-2021].
- [9] GNU. Categories of free and nonfree software. <https://www.gnu.org/philosophy/categories.html>, 2019. [Online; accessed 20-May-2021].
- [10] Daniel Haak, Charles-E Page, and Thomas Deserno. A survey of dicom viewer software to integrate clinical research and medical imaging. *Journal of digital imaging*, 29, 10 2015.
- [11] Daniel Haehn. Slice:drop: collaborative medical imaging in the browser. pages 1–1, 07 2013.
- [12] Paul Hamill. *Unit test frameworks: Tools for high-quality software development*. O'Reilly Media, 2004.

- [13] Mehedi Hasan. Top 25 best free medical imaging software for linux system. <https://www.ubuntupit.com/top-25-best-free-medical-imaging-software-for-linux-system/>, 2020. [Online; accessed 30-January-2020].
- [14] Alessio Ishizaka and Markus Lusti. How to derive priorities in ahp: A comparative study. *Central European Journal of Operations Research*, 14:387–400, 12 2006.
- [15] Sama Jan, Giovanni Santin, Daniel Strul, S Staelens, K Assié, Damien Autret, Stéphane Avner, Remi Barbier, Manuel Bardiès, Peter Bloomfield, David Brasse, Vincent Breton, Peter Bruyndonckx, Irene Buvat, AF Chatziioannou, Yunsung Choi, YH Chung, Claude Comtat, Denise Donnarieix, and Christian Morel. Gate: a simulation toolkit for pet and spect. *Physics in medicine and biology*, 49:4543–61, 11 2004.
- [16] Ron Kikinis, Steve Pieper, and Kirby Vosburgh. *3D Slicer: A Platform for Subject-Specific Image Analysis, Visualization, and Clinical Support*, volume 3, pages 277–289. 01 2014.
- [17] Chris Rorden’s Lab. Mricrogl. <https://github.com/rordenlab/MRIcroGL>, 2021. [Online; accessed 27-May-2021].
- [18] Ajay Limaye. Drishti, a volume exploration and presentation tool. volume 8506, page 85060X, 10 2012.

- [19] Fang Liu, Julia Velikina, Walter Block, Richard Kijowski, and Alexey Samsonov. Fast realistic mri simulations based on generalized multi-pool exchange tissue model. *IEEE Transactions on Medical Imaging*, PP:1–1, 10 2016.
- [20] Yves Martelli. dwv. <https://github.com/ivmartel/dwv>, 2021. [Online; accessed 27-May-2021].
- [21] Hemant Kumar Mehta. *Mastering Python scientific computing: a complete guide for Python programmers to master scientific computing using Python APIs and tools*. Packt Publishing, 2015.
- [22] Hamza Mu. 20 free & open source dicom viewers for windows. <https://medevel.com/free-dicom-viewers-for-windows/>, 2019. [Online; accessed 31-January-2020].
- [23] The Linux Information Project. Freeware definition. <http://www.linfo.org/freeware.html>, 2006. [Online; accessed 20-May-2021].
- [24] Nicolas Roduit. Weasis. <https://github.com/nroduit/nroduit.github.io>, 2021. [Online; accessed 27-May-2021].
- [25] Curtis Rueden, Johannes Schindelin, Mark Hiner, Barry DeZonia, Alison Walter, and Kevin Eliceiri. Imagej2: Imagej for the next generation of scientific image data. *BMC Bioinformatics*, 18, 11 2017.

- [26] Thomas L. Saaty. How to make a decision: The analytic hierarchy process. *European Journal of Operational Research*, 48(1):9–26, 1990. Decision making by the analytic hierarchy process: Theory and applications.
- [27] Ravi Samala. Can anyone suggest free software for medical images segmentation and volume? https://www.researchgate.net/post/Can_anyone_suggest_free_software_for_medical_images_segmentation_and_volume, 03 2014. [Online; accessed 31-January-2020].
- [28] Spencer Smith, Jacques Carette, Olu Owojaiye, Peter Michalski, and Ao Dong. Quality definitions of qualities. Manuscript in preparation, 2020.
- [29] Spencer Smith, Jacques Carette, Olu Owojaiye, Peter Michalski, and Ao Dong. Methodology for assessing the state of the practice for domain x. Manuscript in preparation, 2021.
- [30] Omkarprasad S. Vaidya and Sushil Kumar. Analytic hierarchy process: An overview of applications. *European Journal of Operational Research*, 169(1):1–29, 2006.
- [31] Greg Wilson, Dhavide Aruliah, C. Titus Brown, Neil Chue Hong, Matt Davis, Richard Guy, Steven Haddock, Kathryn Huff, Ian Mitchell, Mark Plumbley, Ben Waugh, Ethan White, and Paul Wilson. Best practices for scientific computing. *PLoS Biology*, 12:e1001745, 01 2014.

- [32] Erik Ziegler, Trinity Urban, Danny Brown, James Petts, Steve D. Pieper, Rob Lewis, Chris Hafey, and Gordon J. Harris. Open health imaging foundation viewer: An extensible open-source framework for building web-based imaging applications to support cancer research. *JCO Clinical Cancer Informatics*, (4):336–345, 2020. PMID: 32324447.

Appendix A

Full Grading Template

appendix here

Appendix B

Summary of Measurements

appendix here

Appendix C

Interview Answers

appendix here

Appendix D

Ethics Approval

appendix here