Evaluating Quality Assessment Tools: Understand, DesigniteJava, and SonarQube

Brad Coddingham, Rachael Jacobsohn, Jon Doretti, DePaul University

Abstract

Software quality is of utmost importance for organizations when developing projects. One way to help ensure the code being written meets the desired quality standards is to use a quality assessment tool. In this paper, we will compare and contrast three different quality assessment tools used to evaluate Java projects: Understand, DesigniteJava, and SonarQube. We will discuss the features of each one and how well they performed when analyzing a small, medium, and large project. We will evaluate the three tools based on several different criteria including features included, accuracy, ease of use, and informative documentation.

I. Introduction

According to IEEE the definition of software quality is “the degree to which a system, component, or process meets specified requirements and customer/user needs or expectations.” Other sources define software quality as “conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.” Both definitions imply that for software to be considered high quality it must satisfy both functional and non-functional requirements. In other words, it must work as desired without bugs as well as be easy to understand, modify, extend, and use.

A Koskinens 2009 survey found that 75-90% of business and command & control software and 50-80% of cyber-physical system software costs are incurred during maintenance rather than initial development. This implies that only a small percentage of a product’s life cycle is spent on actual development related work. Furthermore, it is predicted that by 2025, the number of professionals engaged in software maintenance will rise to 77% from just 10% in the 1950s. Another source estimates that when software is not developed with maintainability in mind, it requires approximately 4 times as much time to maintain as it did to create in the first place. These statistics point to the fact that maintaining software is expensive. It requires large amounts of time, effort, and money to do and even more to do well.

Given how costly software maintenance is, whether that be resolving errors or redesigning the code to make it more extensible so new features can be added, it is natural that organizations would want to focus on developing high quality software. In an ideal world, software would be written such that it is easy to understand, update, and extend and is without antipatterns or code smells. If this were the case, far fewer resources would have to be allocated for maintenance and could instead be used for different purposes that would help an organization increase its profits or achieve other goals.This is exceptionally important in modern times when it is very likely that the person who wrote the code originally will not be the same person responsible for testing or changing it later. In fact, it is very likely that many people will work on the same code over time.

Unfortunately, when resources are scarce, creating high quality software is sacrificed in order to work quickly and write software that may not

even satisfy bare minimum functionality requirements. Thus, by neglecting the tenants of code quality developers incur technical debt and write code that may be full of errors in order to supposedly meet short term milestones. While high quality software is important because it means the code does what it is supposed to do, it is equally important if organizations wish to produce code that meets other non-functional requirements such as those related to performance, modifiability, and reliability.

One way to avoid producing software with poor code quality is to invest in the use of quality assessment tools. These tools can be used to analyze code quality based on a variety of metrics. They can also detect code smells and antipatterns. There are many, many quality assessment options available. Each tool was created to be used for projects written in different languages, and each one also comes with different features and user experiences. In this paper, we will compare and contrast three different quality assessment tools used to evaluate Java projects: Understand, DesigniteJava, and SonarQube. We will discuss the features of each one and how well they performed when analyzing a small, medium, and large program.

II. Methods

To complete this project, our group decided to split up the work by having one person conduct the entire analysis for a single tool (Understand - Brad, DesigniteJava - Rachel, SonarQube - Jon).

Before conducting the analysis we identified three different sized projects (small, medium, and large) to use to evaluate the different quality assessment tools (small - sndcpy1.1, medium - tsunami-security-scanner-0.0.10, and large - Arduino-1.8.19). It was important to us to limit the amount of variance there would be in our analysis, especially since each person would only be working with one of the three softwares. By having each person work with the same projects we could ensure a level of consistency during our investigation.

As part of the evaluation each person reviewed the documentation available for their specific quality assessment tool, making sure to understand what features the tools offered. We then downloaded all three of the test projects and our respective tool, making sure to note how easy the setup process was. Next we ran our tool against the different sized test projects, again noting how easy the tool was to use and the time the assessments took to execute. Before meeting as a group, each person reviewed and organized their outputs.

After everyone had completed the independent investigation of their given tool, we came back together to compare the results, research, and personal experiences. We took time to ask each other questions and ultimately agreed on what to say as our evaluation. The final step of the project was to document our findings.

III. Results

As mentioned previously, for this study, our group investigated three quality assessment tools: Understand, DesigniteJava, and SonarQube. Below is an evaluation of each and a comparison between them.

A. Understand

The Understand code analyzer is a multipurpose tool developed by SciTools. It is capable of static code analysis through various methods which include a wide range of visual, documentary, and metric gathering tools. Compared to the other tools that were evaluated, Understand features the easiest set-up and installation. The analyser is installed through a normal .exe process which means no command line or console use is needed.

Additionally, Understand features a smooth user interface that is both efficient and easy to use for all user experience levels. Importing source code is as simple as selecting a folder. Once done the user is immediately shown a visual breakdown of the pertinent data such as number of lines of

code, complex functions, and directory structure. Each category can be selected and dissected visually into more specific data. Users are able to stack multiple projects together on the home screen to allow an effortless comparison of multiple programs or applications. Performing a deep analysis is surprisingly quick and agile. Even when analyzing the largest test program, Arduino with its 27,000 lines of code, only took a minute.

Furthermore, users are able to review specific parts of the source code using several graphing features included in Understand. These graphs include UML, butterfly, calls, and dependency to name a few. Reviewing a specific part of the source code is made easier by double clicking on the part they wish to inspect and Understand will show the actual source code of the item selected. Understand can also review and export all quality metrics of the source code and even export them to an excel file so the user can easily organize and sort the data.

Despite the easy to use and navigate setup, the Understand tool does come with some limitations when compared to other tools. When specifically trying to analyze code smells, Understand falls a bit short. The software does not specifically point out code smells or antipatterns in any meaningful way. It is up to the user to sort through the code and manually find them on their own.

Another slight limitation to Understand is that there isn't a rating or scoring system as part of the code analysis. With other tools, a score or grade is given for the overall selected source code, which can help a developer understand how their code is performing. Unfortunately, this helpful feature is missing in Understand and there is distinct lack of any quality summary in all the visually impressive data provided by the code analyser.

Overall, Understand is a great tool for both beginners and seasoned developers alike. It's easy to use and set up for one or many projects simultaneously. It has many options that can visually represent data in such a way that makes code analysis more comfortable. Its limited drawbacks of pinpointing or rating code smells are a minor inconvenience but if the user is looking for fast metric code analysis then the Understand tool is the right fit for the job.

B. DesigniteJava

DesigniteJava is the second code quality assessment tool with the goal of reducing technical debt and improving maintainability we investigated for this project. Its primary features include detecting various architecture, design, and implementation smells as well as computing many commonly used object-oriented metrics.

One of the advantageouses of DesginiteJava is that it works on machines using Windows or Mac operating systems. Academic licenses are also available to researchers and students who can confirm that they are affiliated with an accredited institution. While very generous, it can take some time for the academic license to be approved, so those wishing to use DesigniteJava should take that into account.

The documentation for DesigniteJava was fairly helpful when it came to explaining the features provided by the software. It would have been nice to have more information about how they calculated their metrics and what the different code smells meant, but we appreciated that the documentation linked to the source that provided the foundation for building the tool. One of our biggest struggles using DesginiteJava was that it was slightly confusing that the documentation and other resources online indicated that there is a GUI that can be used to create visualizations from the assessment outputs. We did not find instructions for how to set this up and were not able to get this working.

Downloading the software is as simple as clicking a button on the website and dropping the jar file and the config file in the file explorer location from which you would like the software to run. It is important to note that if someone would like to move the jar file, they have to move the config file too. Running the software is just as easy as downloading. After authenticating, simply open a command prompt window or terminal and run a command that tells the program to analyze the

contents of a given folder and place the outputs in another folder. The assessment of larger projects took longer than smaller projects, but all of our runs were very quick, only taking a couple of seconds.

After running the command, DesginiteJava provides a summary of the results in the command prompt or terminal and populates more detailed data in the output folder. The information provided by DesigniteJava runs was good, but the outputs were hard to digest. The information in the command line outputs contained useful summarizing information about the categorization of the different code smells detected in the project being analyzed, but the presentation was not well organized. Similarly, the excel spreadsheets in the output folder were hard to parse.

Despite the disorganization, one of the best parts of using DesigniteJava was how explicitly it called out different code smells. Not only did it identify the specific package and file, it explained what the code smell was and how it was detected. If someone were trying to refactor their code, this information would be very helpful in choosing what to change and how.

One thing that was challenging about using DesigniteJava was that it only supported a limited number of object-oriented metrics. For example, in its output, DesginiteJava does not include any information about the coupling between object classes (CBO). If we needed metric values beyond what is provided, we would have to compute them ourselves. Additionally, some of the metrics were labeled with acronyms that we were not familiar with. While we could easily check the documentation, it would have been ideal if DesginiteJava had used the more standard names.

C. SonarQube

The last tool used is SonarQube. The goals of SonarQube consists of three parts: (1) improving maintenance, (2) improving security, and (3) track who solved the problems. In other words, SonarQube is a corporate level software that is suited for a large development company - for example, Google or Facebook - to diagnose security failure, code smells, and track what employee completes each task.

SonarQube requires a lot to set up and is not beginner friendly. This tool is not made for personal projects as the setup is different for every piece of code. There are many steps required for setting up SonarQube: (1) download zip file, (2) setup web server, (3) PortgreSQL database. Also requires the user to configure many settings; for example, the user must configure sonar groups and users. SonarQube has a very clean and well designed web interface that allows you to see individual problems within the codebase. SonarQube tests for three metrics: (1) Reliability - bugs, (2) Security - Vulnerabilities, (3) Maintainability - Code Smells. SonarQube also gives a security review discussing vulnerabilities. However, this feature could not be used as it required upgrading from the free version to the enterprise version.

Documentation for SonarQube did not go into depth on how to set up the software. It listed a lot of tools to help when the server and software has already been set up. This required using second hand sources to get SonarQube working properly. As set-up is different for each OS: Windows, MacOS, and Linux - documentation went into depth for Windows and Linux users, however, not MacOS. A lot of the tools that are talked about in the documentation are locked in the free version of the software as SonarQube is paid software that has a free version.

Testing software projects is complicated. Setup for all projects must be done from the command line even though SonarQube has an extensive web interface. SonarQube also has various options as to what type of project may be tested. For example, our medium software we tested - tsunami-security-scanner-0.0.10 - was built using gradle. SonarQube requires gradle files to be set up differently from Maven files and regular java projects. For testing purposes on tsunami-security-scanner-0.0.10, the build.gradle file must be edited so that it includes a plugin for SonarQube. SonarQube aslo is not compatible with .class or .java files. To test a software project like our large project - Arduino-1.8.19 - the direct path to the projects .jar files must be linked within the set up commands.

Metrics on SonarQube were not displayed on the web interface. The web interface allowed for individuals to see each issue found in the code, look directly at the lines of code or class, select this error, and assign it to themselves. SonarQube resembles a help desk where tickets are sent in and accepted as a way to track employees productivity and work. SonarQube does allow for metrics to be exported, however, metric data is blocked as exporting metrics data is not available for the free version of the software. Metric data files are also exported as .pb files with no option to change to .csv files. This is not efficient as it is easy to set up a python script that parses through .csv files and collects the data.

In all, SonarQube is a really extensive and a very well designed testing software. It is not suitable for personal projects as SonarQube's design focuses on corporate code bases. This is supported by their assigning and tracking system for issues found by the software. In other words, for tracking corporate code SonarQube is the software to use. For smaller projects or personal projects, Understand or DesigniteJava may be the softwares to use.

D. Comparison

When comparing quality metrics from all three source code applications between the three code analysis tools, it's clear that there are some inconsistencies.

Fig. 1.

Understand Tool						
Version	**LCOM**	**DIT**	**CBO**	**NOC**	**RFC**	**WMC**
Sndcpy	22.5	1.7	12	0	17	8
Tsunami	13.8	1.4	9.5	0.18	19.6	6.4
Arduino	23.5	1.2	10.4	0.14	81.9	12.1
Criterion	Low is Good	less 5	less 14	Mid is Good	Below 50	20-50
DesigniteJava Tool						
Version	**LCOM**	**DIT**	**CBO**	**NOC**	**RFC**	**WMC**
Sndcpy	11	0	N/A	0	N/A	11.33
Tsunami	50	1.1	N/A	0.24	N/A	6.6
Arduino	44	1.2	N/A	0.15	N/A	15.8
Criterion	Low is Good	less 5	less 14	Mid is Good	Below 50	20-50

Figure 1: C&K metrics compared (Laing & Coleman, 2001).

After extensive testing, the results in *Figure 1* were produced. There are no C&K metrics for SonarQube as that application was unable to produce metrics without purchasing a premium plan. However, both Understand and DesginiteJava were able to analyze the three sample source code projects and produce metrics. Additionally, DesginiteJava wasn't able to analyze a few C&K metrics such as CBO and RFC.

It is clear that there are some inaccuracies with these metrics given that the two different tools do not always yield the same results. The metrics that were produced by Understand and DesginiteJava were fairly close in range to one another for some of the metrics, but were vastly different for others like LCOM. From a metrics standpoint it would appear that Understand is superior. DesginiteJava is capable of generating metrics to an extent, but it is clear that the tool is primarily focused on identifying code smells and antipatterns.

While Understand and DesigniteJava were the frontrunners when it comes to usability in terms of set up and execution, they provide very different experiences for interacting with the results of the analyses. With Understand the graphical user interface is very built out and allows users to probe into both the code and the results from a variety of perspectives. These features can be very helpful, but could also be overwhelming to a user looking only for a specific piece of information. On the other hand, DesigniteJava does not provide any sort of graphical user interface. While the outputs were much more streamlined, they were difficult to navigate and hard to parse. Because much of the SonarQube output is inaccessible without a paid subscription, we would not even consider it as the primary candidate when choosing a code quality assessment tool.

Ultimately, as with anything in software development, the choice of which tool to use is highly dependent on the type of information the user is trying to extract, and the experience they are hoping to have with the tool. We would likely choose Understand because of the variety of features it offers, the fidelity of the metrics, and its ease of use. However, if we were more interested in gauging what code smells and antipatterns were present in our code and where they appear, we would likely select DesigniteJava instead. If the budget (time, cost, and people hours) allowed, the best option would actually be to use Understand

and DesgniteJava in conjunction, so the features of both tools could be leveraged.

IV. Conclusion

As discussed in the introduction, the majority of development time is spent maintaining software. As a result, investing in high quality software is extremely important, especially as organizations and their projects begin to scale. Doing so will make the code easier to modify and extend in the future. Using a quality assessment tool to analyze a project is a good way to make sure the software being produced meets the desired quality standards. For this project we evaluated three different tools based on the user experience they provide and the features they offer. If possible, we suggest using Understand and DesigniteJava, so we could take advantage of the different features offered by each one. However if we can only pick one, it would be Understand.

References

[1] Laing, V., & Coleman, C.J. (2001). Principal Components of Orthogonal Object-Oriented Metrics.

[2] M. Schnappinger, A. Fietzke and A. Pretschner, "Defining a Software Maintainability Dataset: Collecting, Aggregating and Analysing Expert Evaluations of Software Maintainability," 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 278-289, doi: 10.1109/ICSME46990.2020.00035.

[3] C. Chen, S. Lin, M. Shoga, Q. Wang and B. Boehm, "How Do Defects Hurt Qualities? An Empirical Study on Characterizing a Software Maintainability Ontology in Open Source Software," 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2018, pp. 226-237, doi: 10.1109/QRS.2018.00036.

[4] S. Elmidaoui, L. Cheikhi and A. Idri, "A survey of empirical studies in software product maintainability prediction models," 2016 11th International Conference on Intelligent Systems: Theories and Applications (SITA), 2016, pp. 1-6, doi: 10.1109/SITA.2016.7772267.

[5] IEEE, IEEE Standard: 828-1998 -- IEEE Standard for Software Configuration Management Plans, IEEE Computer Society (1998).