

# *CalcPlus: Software Testing and Quality Assurance Strategies on a Noncomplex Application*

Arturo Chaidez  
SE433 SW Testing & QA  
DePaul University  
Chicago, USA  
achaide3@depaul.edu

Brendan Burkman  
SE433 SW Testing & QA  
DePaul University  
Chicago, USA  
bburkman@depaul.edu

Eliécer Peraza  
SE433 SW Testing & QA  
DePaul University  
Chicago, USA  
eperazaa@depaul.edu

**Abstract**—This electronic document describes use of tools and techniques for software testing and quality assurance applied during the analysis, design, implementation of an application that for a basic calculator and unit converter, as well as the analysis of the results obtained from those tools and techniques.

**Keywords**—quality assurance, software, testing, DesigniteJava, Java, Understand, EclEmma

## I. INTRODUCTION

CalcPlus is an application built from scratch, which includes features such as arithmetic operations, unit conversions, and more over time. During its development, techniques to assess and ensure software quality and proper testing have been consequently applied. CalcPlus will allow the user to perform the basic arithmetic operations, as well as unit conversion in an intuitive, effective, efficient and easy manner. CalcPlus analysis, design and implementation, and enhancements has been developed using design patterns, quality metrics, software testing tools, and any other tool/approach deemed necessary pursuing a product of great quality and error-free.

## II. PROBLEM STATEMENT

In the professional field, people deal with software programs that do not comply with the minimum requirements to be cataloged as decent. Code that includes multiple bugs, no readability, lack of encapsulation, bad design, no documentation, no reliability, impossible to maintain, lack of extensibility or standards, and so on. As a result, multiple individuals/users have to interact with applications, machines, websites, and other devices that do not work properly or have a very low performance. Then, why not start adopting and using techniques, strategies, and tools that allow programmers to comply with the minimum requirements of good coding from the beginning, to have a fully, efficiently, and properly working product in the end?

For this short project, a calculator/converter was selected as the product because it was determined that it would be completed in the given time yet still be challenging coding wise. However, the goal of this course will be achieved in this project: software testing and quality assurance.

## III. METHODOLOGY

After submitting a proposal, and gathering functional and nonfunctional requirements and building the use case, sequential and class diagrams, the design and implementation of the different features of the tool took place. These features were implemented using design patterns, quality metrics, software tests, and any other tools/approaches deemed necessary to achieve a product of great quality and error-free. As the product was being designed and developed, techniques of quality design have been learned, applied, and integrated into this development. As the project progressed, quality assurance software was applied on the code. Then, the results the tools provided analyzed, and then converted into useful input about the status of product quality-wise. Changes and planned future changes were made based on quality assurance (QA) metrics. In the meantime, software testing was performed on all parts and modules of the project, including individual and integrated testing. Results of the testing led to identifying and correcting bugs during the process. Google converter was used as reference for the factors involved in the different unit conversions [1].

## IV. RESULTS

A summary of the results of the analysis made on the product is presented below, focusing on coding, testing and QA tools.

### A. Coding

To make this program a reality, the design incorporates various design patterns, such as factories, object pools, singletons, and strategy patterns. This was to make the program much more readable and scalable for future modifications. The most useful pattern was the strategy pattern, which was used for various logic operations, specifically this pattern was used in the operations class. The main goal of this pattern was to avoid unnecessary switch statements and through this pattern, the design incorporated a much more object-oriented approach to making a calculator and converter. This easily made it so the code didn't have to do many if checks. So, with this pattern, the branches that needed to be checked were reduced. This also allowed debugging to be rather easy as the amount of lines of code was reduced for an increase in the amount of files.

Another pattern that was used was the singleton pattern. The goal of this pattern is to reduce reference passing and to be able to access other classes in other areas of code but keeping

everything encapsulated. There was also a small button factory, which pools JButtons for reuse. This was early on in development and could be used if the program is expanded and includes features that need many buttons.

The features of this program also fit the MVC pattern (Model, View, Controller). This design is to make it so that the visual aspect and the logic aspect of the code is separate. This made it so the design avoided having an unnecessarily large blob class and helped with scalability, so future iterations could use the same classes if the code had to put into.

For JUnit testing, the program internally essentially simulates pressing buttons and selecting items, but in the code [2]. The thinking behind this was to be able to test the logic, but also the controller and the visual aspect of the programs which includes the buttons and the combo boxes. Essentially for the JUnit, the design makes it so that it is able to test the features as a whole and not just parts of it. The design also made it so for the JUnit tests, testers could turn on the actual programs without having to visually show the windows. This was needed as the program is simulating the button pressing and needed to set up all the windows, the controllers, and models.

This was done through functions of the controller CalculatorController like StartJUnit(), it also tested the shutdown of the singletons through functions like Turnoff().

## B. Testing

The main tool used for testing was JUnit. JUnit is a unit testing framework for Java and is one of the most commonly used external libraries in Java. After a feature is developed, a JUnit test is created to make sure this feature works as intended, as well as find unexpected errors. When designing tests, what is important to keep in mind is to design the tests where bugs may possibly occur. These are called edge cases. For addition and subtraction, the tests not only check if the calculator can handle simple add and subtract but also handle other situations as well. Both adding and subtracting must handle going from negative and positive numbers, and vice versa. The tests must prove calculations return the correct decimal place. For the tests for multiplication and division, they must handle the same edge cases as well as handling zero arithmetic correctly. Overall, many of the same edge cases appear across all four operations.

Addition:

- positive + positive
- positive + negative (answer: positive)
- positive + negative (answer: negative)
- negative + negative
- non-decimal + non-decimal
- non-decimal + decimal OR decimal + non-decimal
- decimal + decimal (carry over decimal)
- decimal + decimal (don't carry decimal)
- pos. decimal + neg. decimal ("borrow" digit across decimal)

Subtraction:

- positive - positive (answer: positive)
- positive - positive (answer: negative)
- positive - negative
- negative - negative (answer: positive)
- negative - negative (answer: negative)
- non-decimal - non-decimal

- non-decimal - decimal OR decimal - non-decimal
- decimal - decimal (carry over decimal)
- decimal - decimal (don't carry decimal)
- neg. decimal - neg. decimal ("borrow" digit across decimal)

Multiplication:

- positive \* positive
- positive \* negative
- negative \* negative
- non-decimal \* non-decimal
- non-decimal \* decimal OR decimal \* non-decimal
- decimal \* decimal
- zero \* number

Division:

- positive / positive
- positive / negative
- negative / negative
- non-decimal / non-decimal
- decimal / non-decimal OR non-decimal / decimal
- decimal / decimal
- number / zero
- zero / number

Additionally, these JUnit tests checked for functional requirements for the calculator. Most of these requirements were tested in the edge case/button JUnit tests, while a couple needed new JUnit tests for them. The first five were operations: add, minus, multiply, division, percentage. The four main operations have their own test class (see Fig. 1, Fig.2), while percentage was tested in another class – *MoreCalcTests.java* (see Fig. 3) These requirements are straight forward, tested repeatedly, and no bugs appeared. Another requirement inputting any real number, either integers or the decimal form representation of the number. Once again, this requirement was checked in multiple JUnit tests with no issues. The next requirements are the calculator must have an initial value must be 0 and most show a precision of at least two decimals. The first requirement required new tests to be made while the second was checked in different tests. The initial value of 0 had no issues, but there were rounding errors for decimals. What was discovered was that using GetAnswer() did not round while GetText() did. When doing assertEquals(), the first argument was the text from GetText(), while the second argument was the text returned by the calculator. This way, what the calculator returns is what is being tested. These tests returned equal answers, even with the decimal answers. Handling positive and negative numbers were checked multiple times with no bugs.

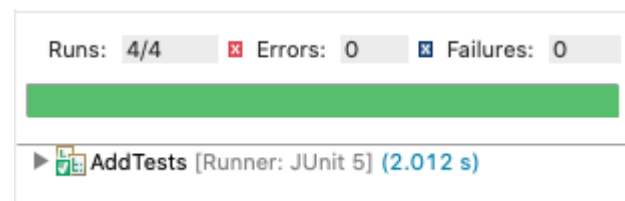


Fig. 1. Calculator toll - Junit tests results. Part I - Addition tests.

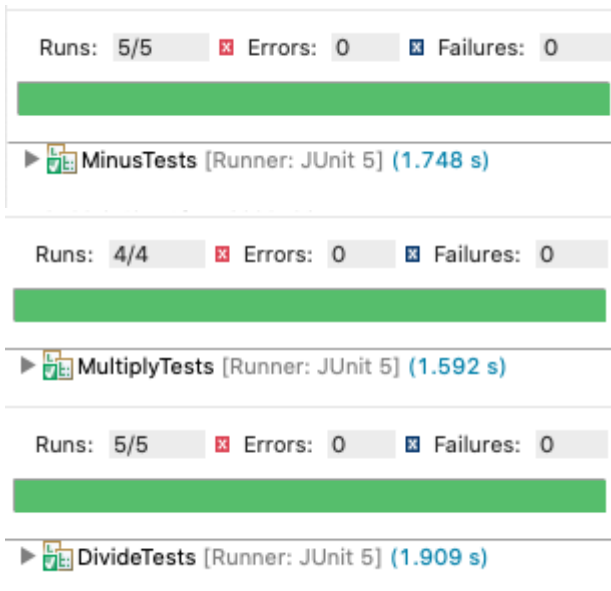


Fig. 2. Calculator tool - JUnit tests results. Part II – division, multiplication and subtraction tests.

However, there were two functional requirements that did fail during testing. One requirement was to return an ERROR when warranted. This requirement was tested when dividing by zero and returning infinity. This bug was fixed and now returns “Error”, as initially expected. The calculator will correctly accept other inputs after this event. Another requirement is to be able to carry multiple operations. For example, the calculator should be able to handle  $2 + 2 + 2 = 6$ . However, what was discovered is the calculator does not calculate the first  $2 +$ , only calculating  $2 + 2$  to equal 4. Inputting the next operation should execute the previous operation and provide the partial solution, which the calculator does not do. This is something that is planned to be fixed, and how this will be achieved is discussed in the future plans section of the paper. This issue was uncovered by the unit test represented as failure in Fig. 3.

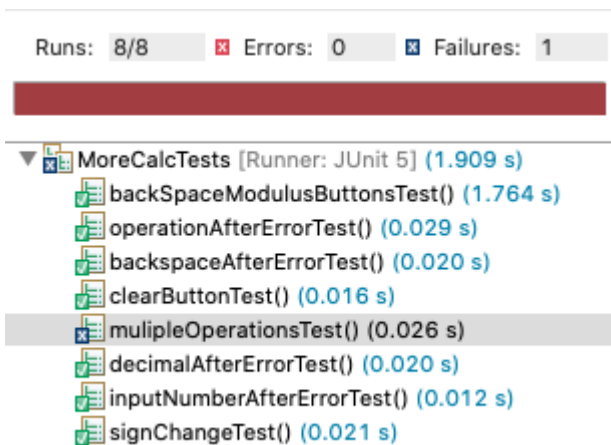


Fig. 3. Calculator tool - JUnit tests results. Part III – uncovering issues.

Regarding the converter tool, the JUnit tests covered all unit categories on the scope of the requirements: area, digital storage, length, mass, temperature, time and volume (package test.converter). Initially, a single test was created for each category to evaluate overall functionality (BasicConverterTest.java). Then, all unit type pairs in the form (From Unit, To Unit) were added to the unit tests for every category, using JUnit5 Parameterized Tests, to cover all available conversions (Test<Category>Conversion.java). Tests were run multiple times and uncovered issues related to rounding, integer division vs decimal division, and misconfigurations. As a next step, issues were tracked down and fixed. After a few iterations, all encountered bugs were corrected and the final run of the tests for the converter tool was 100% successful (see Fig. 4).

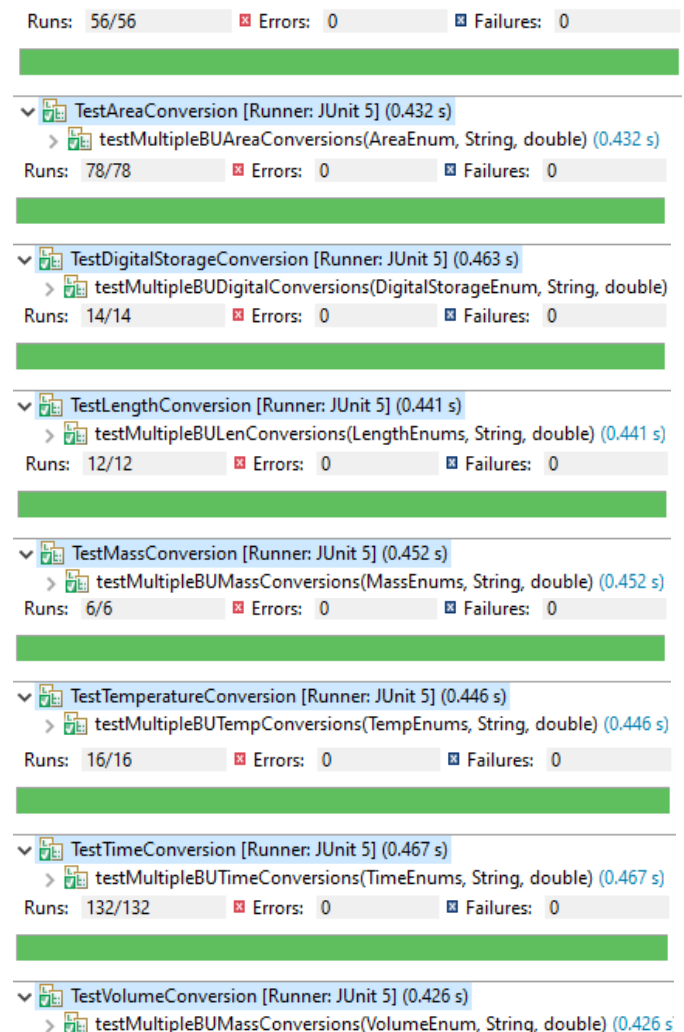
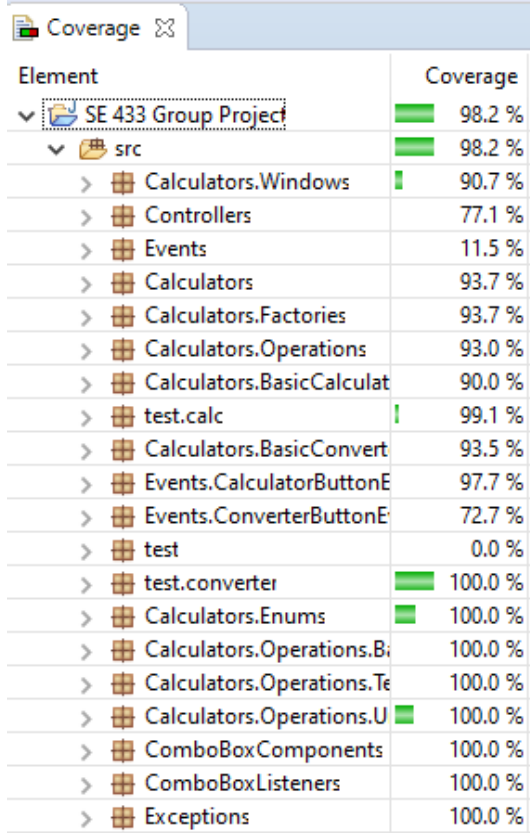


Fig. 4. Converter tool – Junit tests results. All categories reresented.

Eventually, the calculator tests and the converter tests were grouped into a test suite, *CalcSuite.java*, to assess testing coverage, using Eclemma [3]. The coverage focused efforts in the logic components rather than the visual components. However, most visual components were tested while testing the

converter and calculator operations (see Fig. 5). The overall coverage of the project was 98.2%. The coverage of the logic reached 99%. Methods only intended for visual purposes were not taken into consideration regarding the logic coverage assessment.



Element	Coverage
SE 433 Group Project	98.2 %
src	98.2 %
Calculators.Windows	90.7 %
Controllers	77.1 %
Events	11.5 %
Calculators	93.7 %
Calculators.Factories	93.7 %
Calculators.Operations	93.0 %
Calculators.BasicCalculat	90.0 %
test.calc	99.1 %
Calculators.BasicConvert	93.5 %
Events.CalculatorButtonE	97.7 %
Events.ConverterButtonE	72.7 %
test	0.0 %
test.converter	100.0 %
Calculators.Enums	100.0 %
Calculators.Operations.Bi	100.0 %
Calculators.Operations.Te	100.0 %
Calculators.Operations.U	100.0 %
ComboBoxComponents	100.0 %
ComboBoxListeners	100.0 %
Exceptions	100.0 %

Fig. 5. Project testing coverage – Powered by EclEmma

### C. QA Tools

The two QA tools used were Understand [4] and DesigniteJava [5]. Understand has a variety of features that point out unhealthy code to software developers. Specifically, there are various metrics that may indicate design flaws in the code. For DesigniteJava, this tool returns metrics and some of these metrics will be the same as those from Understand. This way, if one metric indicates the code is fine while the other does not, this may indicate flawed code. However, a metric being within an acceptable, healthy range does not indicate the code is good code. These metrics are useful in finding flawed, poor code, which is why two separate software metrics are being used. DesigniteJava also gives code smells: architecture smells, design smells, and implementation smells.

Starting with design code smells, there were numerous design smells. However, almost all of these have a disclaimer to ignore this smell if the class serves a specific purpose. In fact, almost all the design smells are unutilized abstraction and can be explained why they should be ignored. For all the unit conversion JUnit tests, DesigniteJava said these are unutilized abstractions. DesigniteJava does not understand these classes are for testing, so these are not code smells. Class CalcSuite is also used for testing and is not a code smell. The purpose of ComboBoxListener is to be a listener. BasicCalculator is a class

mostly of getters and setters, a common misclassification by DesigniteJava. All the calculator Operations and Basic Unit Conversions have this code smell as well, they all extend class Operation and do serve a purpose. All the window classes (CalculatorWindow, MainMenuWindow, UnitConverWindow) are used to create the view. There are a few code smells that are not unutilized abstraction. Operation class has a code smell of a wide hierarchy, stating that this class has too many children. Its children are all the calculator operations and unit conversions. Operation was designed this way to add new features easily in the future, which is why this class does have many children. However, combining the temperature conversions is explained thoroughly in the future plans section of the paper.

In implementation smells, there are two recurring smells: long statements and magic numbers. In every case, they should be ignored. The long statements appear in the unit conversion JUnit tests, where parameterized tests are utilized to accept multiple inputs. This allowed checking each conversion type with one test, instead of one test for every conversion. This dramatically reduces the amount of total code here. For the magic numbers, they appeared in two situations. One was to create the borders of the buttons, main menu, calculator, and converter. These specific numbers are needed. The other situation was in operations and the tests. These are not random numbers that will create issues. These numbers are needed for the operations to correctly work and are strictly defined; for example, Kelvin to Celsius is an exact scientific number of subtracting 273.15. In the JUnit tests, exact numbers are used to check if the calculation is correct.

For architecture smells, one code smell is feature concentration in the unit conversion JUnit tests. DesigniteJava does not understand these are tests and should be separated to test different conversions, as this is common practice in JUnit. Feature concentration also appears for both calculator operations and unit conversion operations. Once again, these are similar operations but needed to be separated as each operation is a very defined and specific operation. DesigniteJava is correct about feature concentration in the temperature conversions, which is something that will be fixed in the future.

The last QA tool are the software quality metrics. For Understand, the metrics used were weighted methods per class (WMC), depth of inheritance (DIT), number of children (NOC), response per class (RFC), and coupling per objects (CBO). These were the averages for each metric: 2.532 WMC, 1.351 DIT, 0.293 NOC, 3.514 RFC. For DesigniteJava, the metrics looked at are WMC, NOC, DIT, and lack of cohesion methods (LCOM). The averages were: 3.646 WMC, .215 NOC, .215 DIT, and .051 LCOM. Overall, these are healthy metrics. There is a slight difference between the two tools for WMC and DIT but not alarming.

### V. CONCLUSIONS

Simple projects can turn into very challenging tasks. The implementation of the logic for developing tools can be simple by itself; however, when adding design strategies, visual components, and mathematical subtleties to achieve the desired product quality, time and effort rapidly increase. This project is not an exception.

### A. Significant Results

The base logic project was developed quite fast, but it was not until the testing phase and the running of QA tools that subtleties not related to the inner logic started to mess up with the final result. A high-quality, bug-free product has been pursued all along the process, and even when not all the goals were met regarding the requirements, the work completed until this phase 3 has been developed using design patterns and strategies, testing over all of the components and applying quality assurance software analysis; hence, the main purpose of the project has been met.

- As mentioned above, there was a specific requirement that was not met but was targeted and uncovered by the testing cases: conducting multiple operations and showing partial results before hitting the equal key “=”. A solution was presented by the designers but time constraints prevented developers from coding it. The solution would be to have history of the previous operation and mask the equal action execution for subsequent operation keys pressed along the way.
- Design patterns used along the project made the code readable and scalable. The key pattern applied was the strategy pattern, used for the main operations to avoid unnecessary code and add a very focused object-oriented design.
- Even when not all of JUnits capabilities were applied to the project, its findings allowed the testers and developers to identify relevant bugs and fix them in a timely manner. Individual tests offered a way to test out specificities from the logic while the parameterized tests provided the testers with a tool to compute multiple inputs at once in an organized manner for simple functionalities and features, as the ones from the converter tool. Testers are willing to learn more about this powerful tool to apply it more extensively in future releases.
- Overall, in both code smells and metrics, the QA tools returned almost no issues. The only issue was having too many classes for temperature conversions. For example, Kelvin to Celsius is nearly the same as Celsius to Kelvin, only in reverse. The same goes for the other temperature conversions. In fact, this was something discussed and acknowledged before using the QA tools. However, emphasizing fixing other bugs and issues that were identified were deemed more critical. The conversions here work as intended with no issues and this cannot be said about other parts of the code that were eventually fixed.

### B. Future Plan

#### 1) Enhancements:

Change some structures to make the code more maintainable. Even when the patterns applied work greatly, there are few structural changes that can make the code easier to maintain, such as the converter enums modification. Separate the converter enums into only the list of units, and use a base unit as an intermediate for every conversion on the same category. It is easier to maintain and reduce significantly the amount of code, the number of configurations, and in consequence, the quantity of testing

cases to be configured. Also, the enums could implement an interface to provide more scalability and ease maintainability. Additionally, Converter Enums should have a property (the factor of the operation); implementing the operation this way would reduce the number of created operations, and maintainability would be a lot easier.

Completely separate the Logic from the Presentation layer. The project approached an MVC pattern but few instances were not separated completely. The intention is to comply with this pattern completely in future iterations.

Generate configurations for the converter tool to be retrieved from file or database. In this way, the code does not have to change to increase the conversion units for existing categories or add new categories. However, program loading could be affected by this change.

A data type of bigger precision (than double) could be beneficial when looking to include conversions throwing very large numbers.

#### 2) Potential future add-ons:

As initially presented in the proposal, this project is simple, functionality wise. However, it is itself a great candidate to add many more tools and turn it into a well-rounded and useful toolbox.

The following functionalities are considered to be added for future releases: scientific calculator, graphing calculator. Additional buttons for the calculator: sin, cos, tan, pie, ln, log, exponents, squared, parenthesis.

Exporting the tool to different platforms as iOS, web, Android has been also considered as part of the future plan. Since Java is cross platform, some of the targets can be covered immediately; however, additional work is required to have the tool available on other platforms without JVM.

### C. Tasks performed by members

All of the team members were constantly involved in all the phases of the project, creating space for discussions that derived in a collaborative work and contributed positively in the development of this application. They also individually contributed based on their own experiences and skills.

#### 1) Brendan Burkman:

Coded and prototyped the various systems that are incorporated into the program. Also set up the github and project layout/foundation. Also made the program JUNIT compatible.

#### 2) Arturo Chaidez:

Wrote JUNIT tests for the calculator, that includes multiple and more complex cases. Ran the source code through metric tools described previously. Also helped to write the metric data reports for the project. Created the Class Diagram and QA Analysis report for phase two.

#### 3) Eliécer Peraza:

Improved the code for the calculator by adding more side cases like catching exceptions. Improved the code for the converter by adding in more unit types for converting. Wrote JUNIT tests for the converter. Created drafts and formatted the documentation provided for all the phases of the project:

proposal, requirements and final report. Specified functional and nonfunctional requirements of the project, as well as created the use cases, sequence diagrams and project tasks and status for previous phases.

#### REFERENCES

- [1] Google unit converter. Software available at [www.google.com](http://www.google.com)
- [2] JUnit 5. 2021 The JUnit Team; software available at [www.junit.org/junit5](http://www.junit.org/junit5)
- [3] Eclemma 3.1.4, Eclipse Contributors and others, 2006; software available at [www.eclemma.org](http://www.eclemma.org)
- [4] UnderstandTM. 2021 Scientific Toolworks, Inc. Software available at [www.scitools.com](http://www.scitools.com)
- [5] DesigniteJava. 2021 Designite. Software available at [www.designite-tools.com](http://www.designite-tools.com)