# Low Code/No Code JSL Unit Test Development:
# A guide to developing JSL unit tests with very little (or no) coding

**Joseph Morgan**

Almost twenty years ago, when Xan Gregg and I started work on the JSL unit testing framework [6] we disagreed on just one aspect of the feature set. I thought that the mechanics of unit test development should be abstracted away, and the framework should simply leverage the JMP datatable to provide a *low code/no code* (LCNC) development environment [1, 2, 3, 8]. Xan thought otherwise, he believed that such a level of abstraction would limit the utility of the framework, especially for knowledgeable JSL programmers. He argued that the framework would have wider adoption if it offered a programming mode as well as a LCNC development mode. So, we compromised and did things Xan's way. It turns out that Xan was correct. Almost twenty years since its release, we have found that over 90% of users of the framework opt for the programming mode. In addition, over the years, despite several workshops and blog posts where LCNC unit test development comes up, I still encounter users that are initially surprised and then enthused, when I demonstrate, or even just describe, the LCNC capabilities that are inherent to the JSL unit testing framework.

Recently, given the increasing prevalence of LCNC development interest in the broader community [1, 2, 8], and the rising awareness of the need for validation among JMP users who build JSL applications (see Extending Hamcrest) it seems timely to write an article that focuses on just the LCNC modality that the JSL unit testing framework offers. Fortunately, the literature on LCNC testing and the tools to support such efforts is growing by the day [1, 2, 3, 8] and so the interested reader will find many opportunities to learn more. My hope is that this article will supplement the existing body of work and, even more so, I hope that it will inform the interested JMP user of a capability that could lead to wider adoption of the framework and, consequently, the benefits that will likely accrue from such adoption.

## The JMP datatable as a development environment

All JMP users have some level of familiarity with JMP datatables. The datatable is how data is presented to platforms for analysis and it is the way that designs from JMP DOE platforms are materialized. Users know how to create datatable columns, how to investigate and manipulate their properties, and many of them have some understanding of column formulas and have likely had occasion to use the formula editor (see Figure 1) to create or manipulate formulas.

The formula editor is more than just a GUI for formulas though, it is useful to think of it as an intuitive, LCNC editor, that allows users to construct arbitrarily complex formulas (i.e., develop simple applications). Its design leverages the WIMP (Windows, Icons, Menus, Pointer) paradigm, and associated behaviors that characterize modern desktop GUIs, to dramatically simplify formula development.

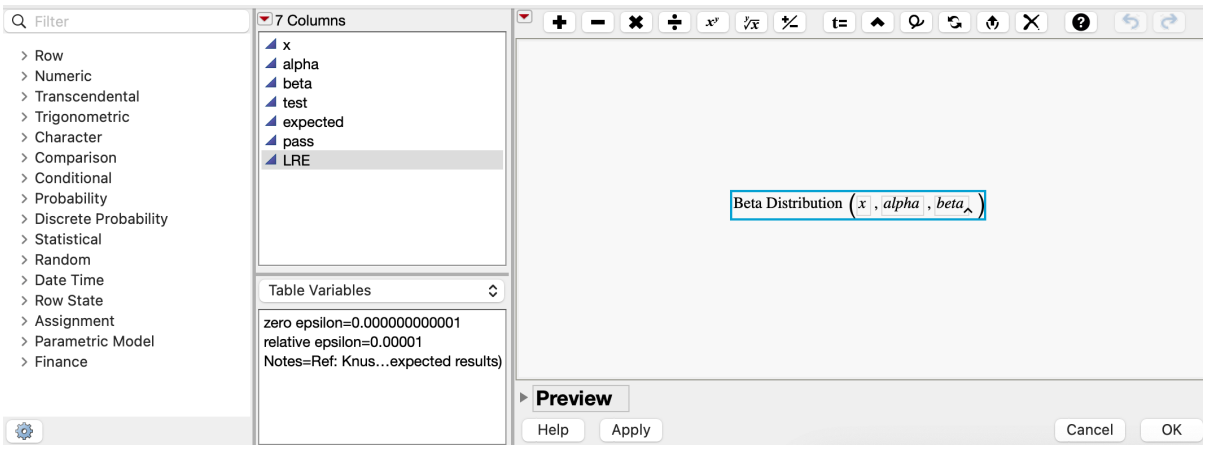



**Figure 1: Formula Editor - GUI mode**

Furthermore, the editor allows users to quickly, and naturally, toggle from a GUI mode to a direct editing mode (see Figure 2) by double clicking on the blue highlighted area surrounding a formula being developed (see Figure 1). This direct editing mode provides a modern source code editor as one would find in an integrated development environment. This simple toggling mechanism, allows users to switch between these two development modalities readily and seamlessly, thus making it easy to choose the appropriate mode for the task at hand.

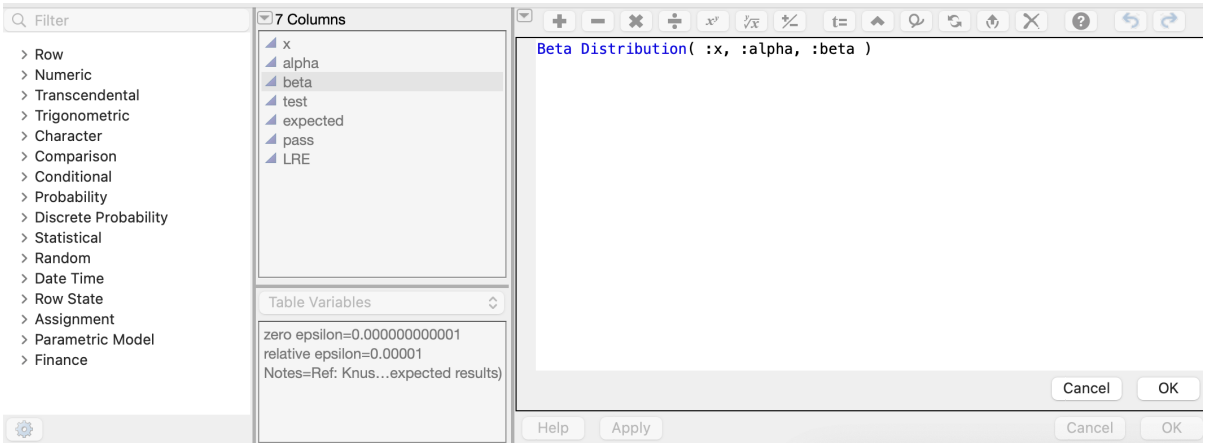



**Figure 2: Formula Editor - Editor mode**

This intuitive formula editing tool, which is widely used and well understood by JMP users, was centrally important to why the datatable was an appropriate starting point for creating a LCNC unit test development framework. Furthermore, by adopting this tool, we were able to honor and abide by a critically important principle of LCNC development frameworks, that is the principle that the framework should both provide support for developing new code/applications as well as providing support for maintaining existing code/applications [2, 8].

In the following sections, we illustrate how leveraging the datatable provides a mechanism for citizen developers [3] to undertake unit test development within JMP. It is worth pointing out at this point that the units to be tested need not be platforms or applications that produce a JMP report,

they could also be functions that return a value, or a platform (or an application) that produces an object that is not a value or a report. A LCNC unit testing framework should provide support for as many of these outcomes as is practical. To address the variety of possible outcomes that may arise in testing within JMP, this article will focus on two broad classes of unit tests, those that test functions that return atomic values (i.e., strings and numbers), and those that test platforms (or applications) that construct a report (i.e., a display tree). Subsequently, we will refer to this approach to developing unit tests as *datatable unit test* development.

**Datatable unit test development**

The **Unit Tests: Automated JSL Testing** whitepaper [6] provides a description of the JSL unit testing framework as well as an overview of several fundamental unit testing concepts. The whitepaper begins by pointing out that *unit testing* refers to the process of validating the smallest testable component of a software system. Software engineers usually refer to such a component as a *unit*. The important point here is that such components must be testable. For our purposes, that means that the component is any feature within JMP that can be invoked, presented with an input, and will then produce an output that can be accessed. So, functions (built-in or user defined) as well as platforms and user defined applications (i.e., add-ins) can all be considered as testable components. From a unit testing perspective, the difference between these components is in how they are invoked and how the desired output needs to be accessed.

There are three additional concepts that are worth elaborating on before getting into the details of datatable unit test development.

- **Actual result**: Given a particular input for a software system, what is the *actual* outcome when the software is executed with that input. The outcome can take on many forms but, for test engineers, the actual result is usually a single value. Note that if multiple values are involved, then each of them is usually assessed separately.
- **Expected result**: Given a particular input for a software system, what is the *expected* outcome when the software is executed with that input. Again, since the expected outcome may involve multiple values, each of them is typically treated separately. Note that it is because actual and expected outcomes may be different that testing is a necessary activity for any software system. If there is a difference between actual and expected outcomes, then software engineers refer to such a discrepancy as a *failure*.
- **LRE**: This is an acronym for **L**ogarithm **R**elative **E**rror [4], which is a measure of the number of correct significant digits when comparing two values, where one of them is deemed correct. However, when the correct value is zero, LRE is undefined, in which case we report it as missing. This is a critically important metric for anyone trying to assess numerical accuracy, especially when some difference between actual and expected outcomes (perhaps relatively small) is anticipated. Two related values, namely *relative epsilon* and *zero epsilon*, are threshold values that are used to determine if the observed difference between an actual and expected outcome is large enough to indicate a failure.

Given this background, let us examine Figures 3a, 3b, 4a, and 4b. These screenshots are datatable templates (see attached zip file) that can be used as a starting point when developing datatable unit tests. The datatables in Figures 3a and 3b may be be used to develop unit tests for functions while those in Figures 4a and 4b would be for platforms or user defined applications.

Notice that there are three columns that are common to the four templates, namely **test**, **expected**, and **pass**, whereas the **LRE** column is only present when the template is for numeric test data. The **test**, **expected**, and **LRE** columns correspond to the **actual result**, **expected result**, and **LRE** concepts discussed above while **pass** is a column that indicates whether the difference between actual and expected results is large enough to indicate a failure. The reason why **test**, **pass,** and **LRE** are formula columns will become apparent as the details of how to use these templates unfolds.

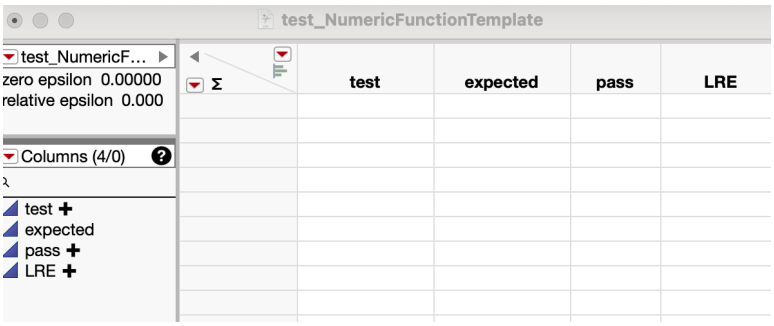



**Figure 3a: Function template for numeric data**

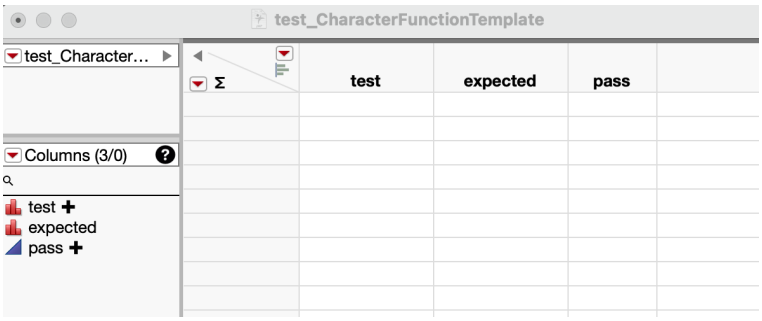



**Figure 3b: Function template for character data**

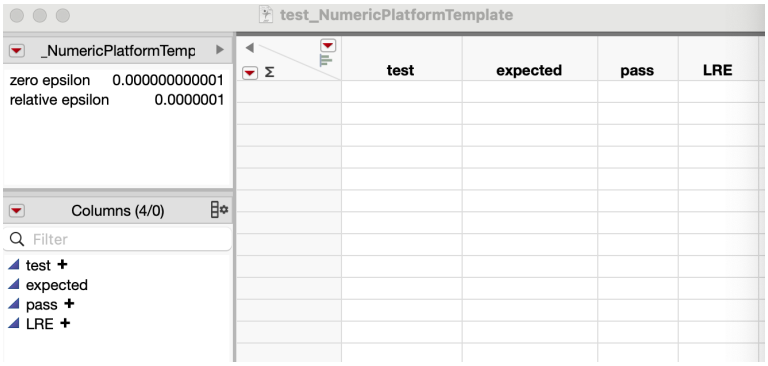



**Figure 4a: Platform template for numeric data**

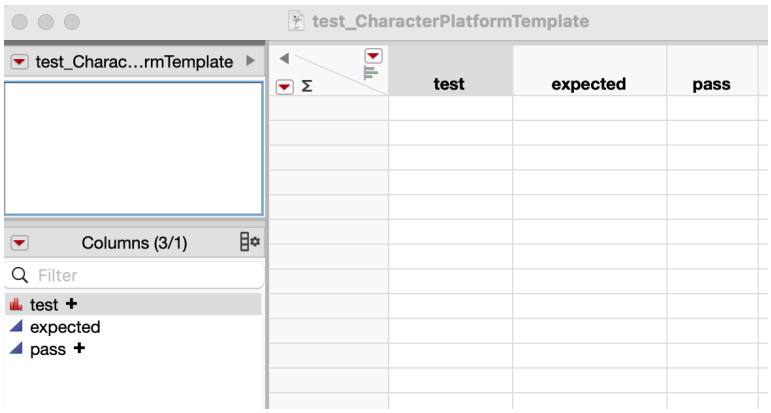



**Figure 4b: Platform template for character data**

With these templates as a starting point, just three steps are needed to create a unit test:

1. Open the template corresponding to the component class (i.e., function or platform/user defined application) and the type of the value to be tested (i.e., numeric or character/string) then save the datatable with an appropriate name to identify the component being tested, making sure to use the prefix *test*.
2. If the component is a function, add a column for each input of the function to be tested, populating each column with data so that each row corresponds to an input. For either component class, populate the expected column with the expected value for each input. This value will be automatically compared to the actual value from the **test** column.
3. Use the formula editor to edit the formula for the **test** column. For functions, specify the function to be tested, making sure to use the columns that correspond to the inputs of the function. For platforms/user defined functions the formula will retrieve data from the display tree. Note that it is this **test** column that will contain actual results.

**Unit test development for functions:** For subsequent examples, we will only consider functions where the output is either a numeric or a string value. Also, for user defined functions, we will assume that the JMP built-in function `addcustomfunctions()` [9] has been used to register (i.e., activate) the function within JMP. In general, it is good practice to register user defined functions, since registered functions appear in both the formula editor and the scripting index. This registration capability has been available since JMP 14 so, for the purposes of this article, we will assume that it is sufficient to use JMP built-in functions for illustration.

The following screenshots show how the steps to create a unit test would unfold for the `substr()` function. Since the `substr()` function returns a string value, we will need to use the **test_CharacterFunctionTemplate** datatable as our starting point.

**Step 1:** Create a unit test datatable named test_SubstrDatatableUnitTest.jmp (see Figure 4a).

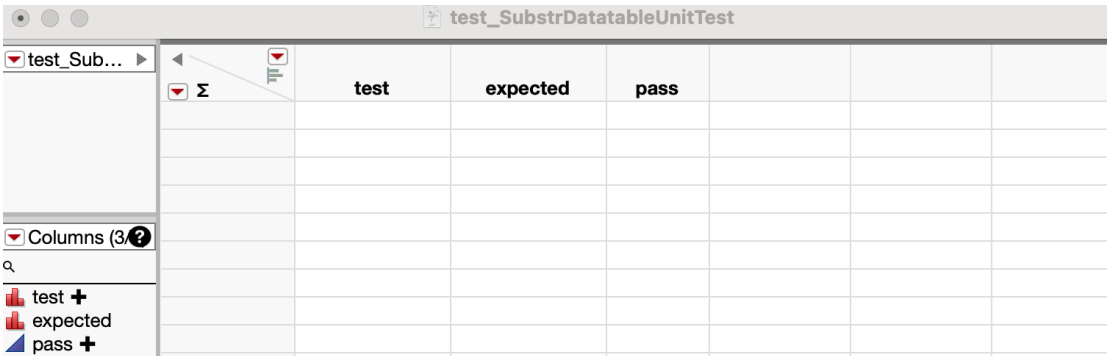



**Figure 4a: Create unit test datatable**

**Step 2:** The `substr()` function contains three inputs, a string input and two numeric inputs and so three columns, one for each input, are needed. In this case (see Figure 4b), columns have been added to the left of column **test** and are named **s**, **start**, and **length**. Note though that any name could have been used and column position does not matter. Five test cases along with the expected results have been added.

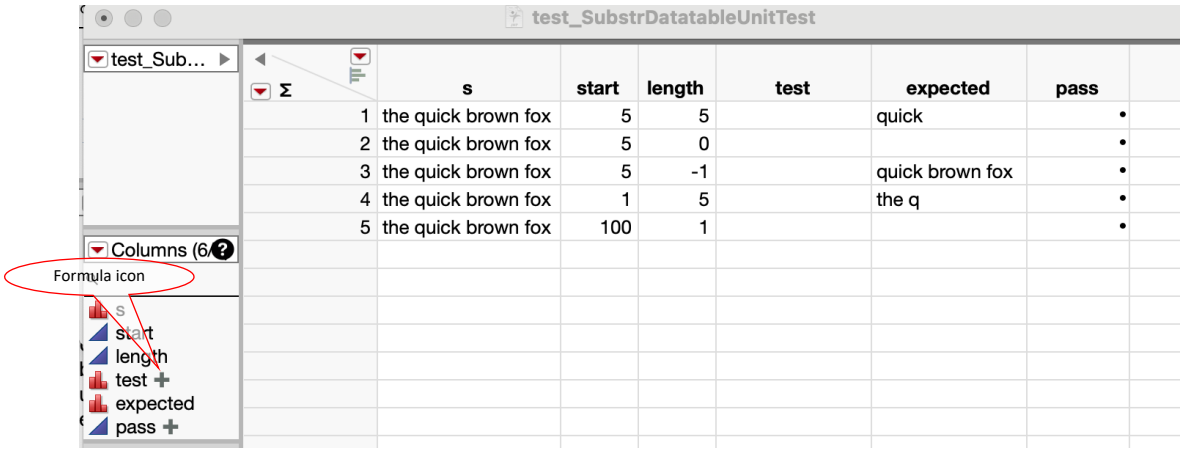



**Figure 4b: Add input data and expected results**

**Step 3:** Launch the formula editor for column **test** by clicking on the formula icon (see Figure 4c). Notice that the left panel of the formula editor is a list of built-in (and registered) functions, organized into groups, the middle panel is a list of columns, and the right panel is a drop zone/editor for assembling the formula.

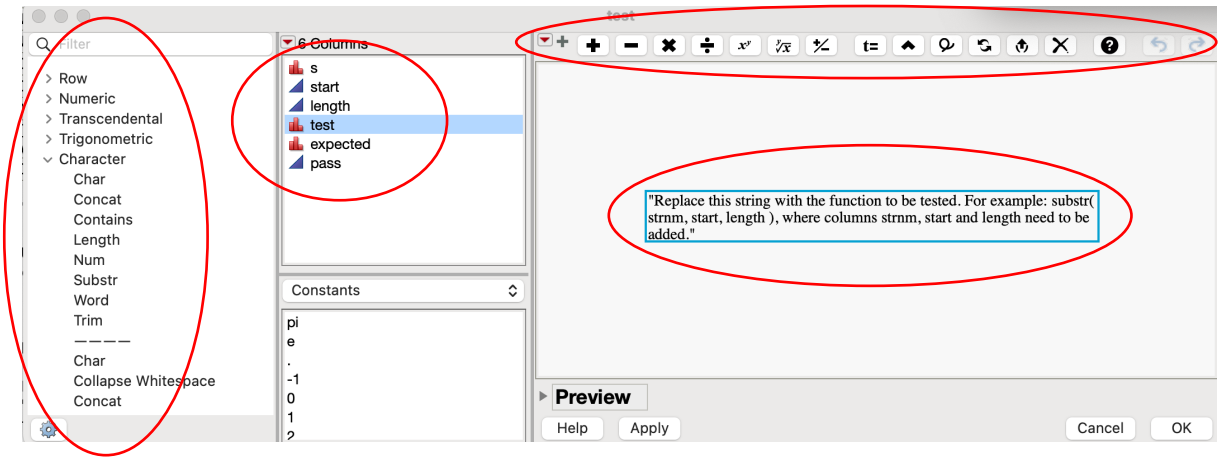



**Figure 4c: Formula editor**

When the formula editor is launched, you will notice that text appears in the drop zone (see Figure 4c). This text is just a tip that the template provides as a reminder of how to assemble the formula. So, to assemble the formula, first delete this text, select the desired function from the function list (leftmost panel), then drag and drop columns from the column list (middle panel) into the slots for the input arguments. Figure 4d shows the assembled formula for our `substr()` example..

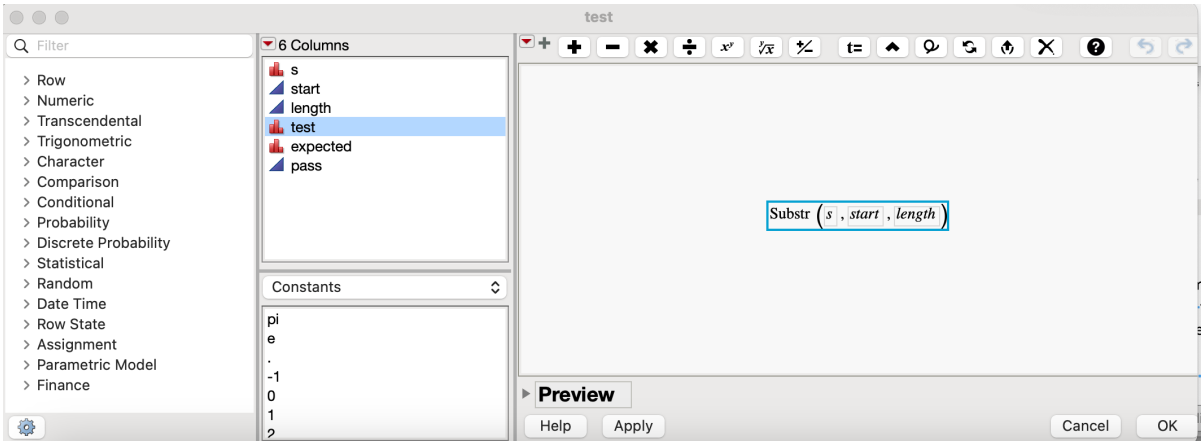



**Figure 4d: Assembled `substr()` unit test formula**

The unit test is almost done, but there is one more action that is needed. That is, click the "OK" button to save and dismiss the formula editor, then use the column information dialog to disable formula evaluation suppression (see Figure 4e) for each formula column (i.e., **test** and **pass**).

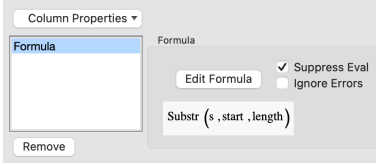
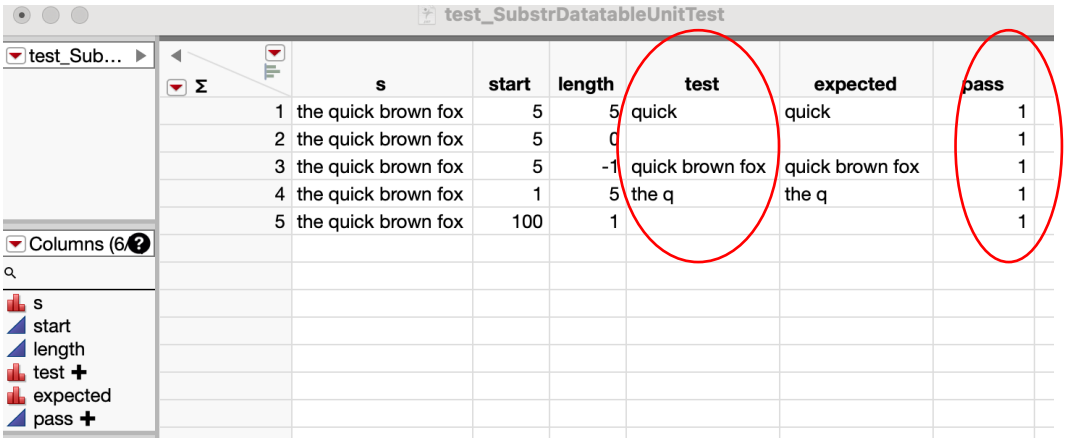



**Figure 4e: Column information dialog**



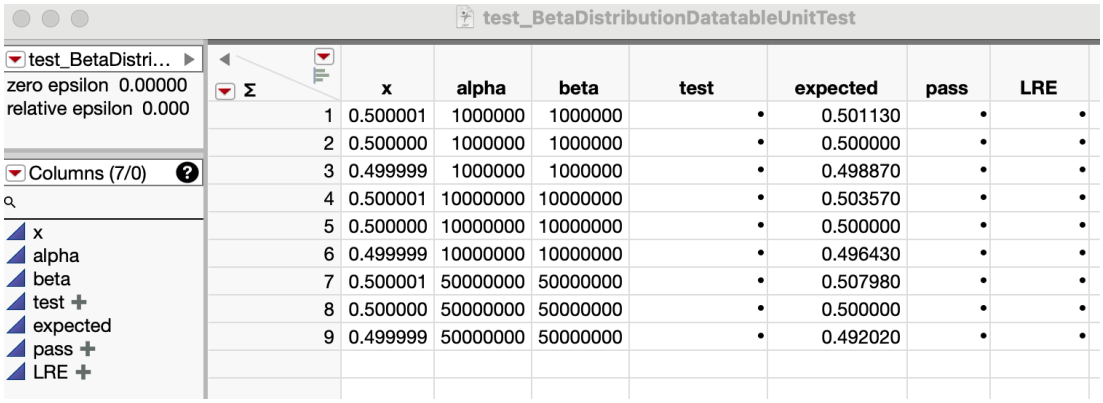

| | s | start | length | test | expected | pass |
|---|---|---|---|---|---|---|
| 1 | the quick brown fox | 5 | 5 | quick | quick | 1 |
| 2 | the quick brown fox | 5 | 0 | | | 1 |
| 3 | the quick brown fox | 5 | -1 | quick brown fox | quick brown fox | 1 |
| 4 | the quick brown fox | 1 | 5 | the q | the q | 1 |
| 5 | the quick brown fox | 100 | 1 | | | 1 |

**Figure 4f: Completed and evaluated unit test**

Once formula evaluation is enabled, the formulas will evaluate for each row of the datatable, and so the test outcomes will be immediately available. In this case, the pass column indicates that all unit tests are successful, that is, actual and expected results agree.

Let us now go through the same steps for a function that returns a numeric value. The following screenshots show how these steps would unfold for the `betadistribution()` function. In this case, we will need to use the **test_NumericFunctionTemplate** datatable as our starting point.

**Step 1, 2:** Create a unit test datatable named test_BetaDistributionDatatableUnitTest.jmp and then add inputs and expected results (see Figure 5a).




| | x | alpha | beta | test | expected | pass | LRE |
|---|---|---|---|---|---|---|---|
| 1 | 0.500001 | 1000000 | 1000000 | • | 0.501130 | • | • |
| 2 | 0.500000 | 1000000 | 1000000 | • | 0.500000 | • | • |
| 3 | 0.499999 | 1000000 | 1000000 | • | 0.498870 | • | • |
| 4 | 0.500001 | 10000000 | 10000000 | • | 0.503570 | • | • |
| 5 | 0.500000 | 10000000 | 10000000 | • | 0.500000 | • | • |
| 6 | 0.499999 | 10000000 | 10000000 | • | 0.496430 | • | • |
| 7 | 0.500001 | 50000000 | 50000000 | • | 0.507980 | • | • |
| 8 | 0.500000 | 50000000 | 50000000 | • | 0.500000 | • | • |
| 9 | 0.499999 | 50000000 | 50000000 | • | 0.492020 | • | • |

**Figure 5a: Add input data and expected results**

**Step 3:** Launch the formula editor for column **test**, assemble the formula (see Figure 5b), then disable formula evaluation suppression for each formula column (i.e., **test, pass** and **LRE**). Figure 5c shows the evaluated unit test.

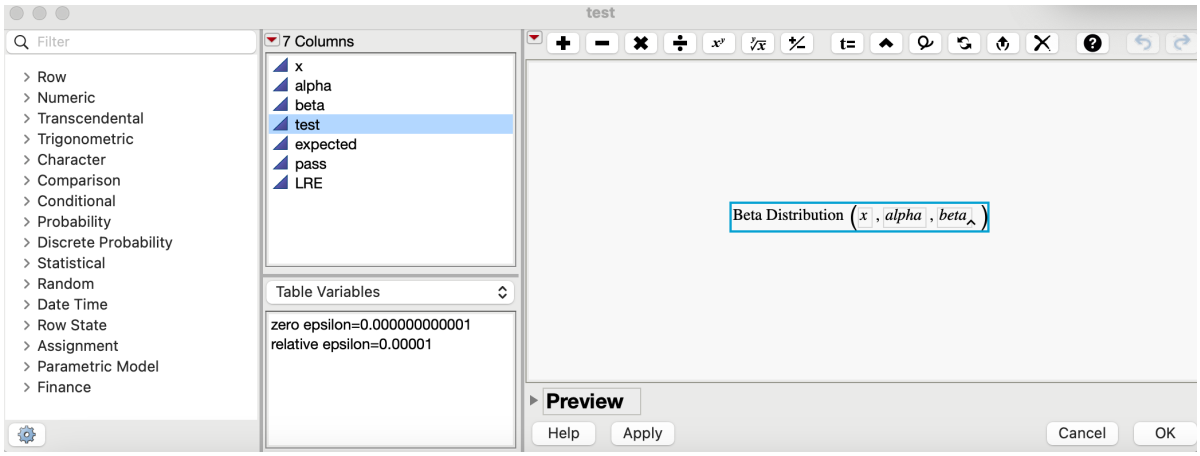
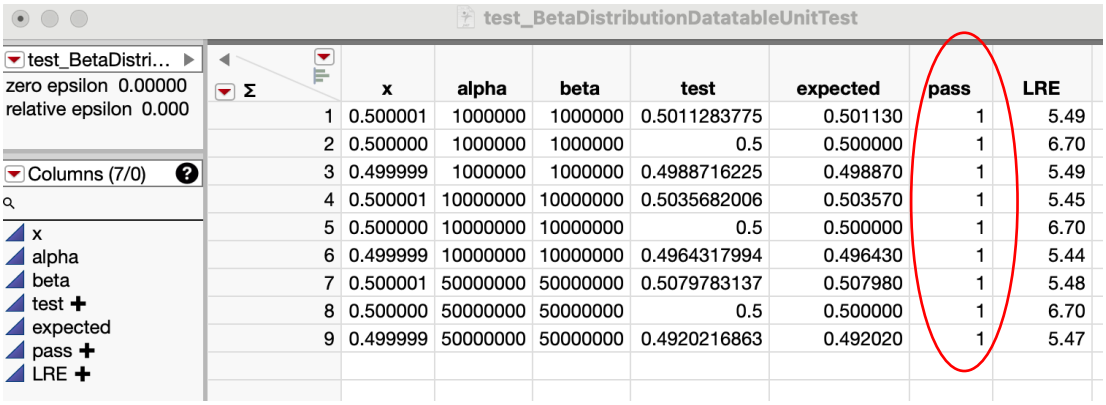



**Figure 5b: Assembled `betadistribution()` unit test formula**




| | x | alpha | beta | test | expected | pass | LRE |
|---|---|---|---|---|---|---|---|
| 1 | 0.500001 | 1000000 | 1000000 | 0.5011283775 | 0.501130 | 1 | 5.49 |
| 2 | 0.500000 | 1000000 | 1000000 | 0.5 | 0.500000 | 1 | 6.70 |
| 3 | 0.499999 | 1000000 | 1000000 | 0.4988716225 | 0.498870 | 1 | 5.49 |
| 4 | 0.500001 | 10000000 | 10000000 | 0.5035682006 | 0.503570 | 1 | 5.45 |
| 5 | 0.500000 | 10000000 | 10000000 | 0.5 | 0.500000 | 1 | 6.70 |
| 6 | 0.499999 | 10000000 | 10000000 | 0.4964317994 | 0.496430 | 1 | 5.44 |
| 7 | 0.500001 | 50000000 | 50000000 | 0.5079783137 | 0.507980 | 1 | 5.48 |
| 8 | 0.500000 | 50000000 | 50000000 | 0.5 | 0.500000 | 1 | 6.70 |
| 9 | 0.499999 | 50000000 | 50000000 | 0.4920216863 | 0.492020 | 1 | 5.47 |

**Figure 5c: Completed and evaluated unit test**

**Unit tests for platforms/applications**: In this case, we will only consider platforms where the output is numeric. We saw from the previous set of examples that the approach is the same whether the output is numeric or string data, so we will omit a string example here. Also, we will illustrate the approach using a JMP platform only. For JMP applications, the approach will essentially be the same if the application can be launched using the `mainmenu()` command.

The following screenshots show how the steps to create a unit test would unfold for the Distribution platform. The intent is to write a unit test to validate summary statistics. Since summary statistics are numeric values, we will need to use the **test_NumericPlatformTemplate** datatable as our starting point.

**Step 1, 2:** Create a unit test datatable named test_DistributionSummaryDatatableUnitTest.jmp. Unlike our previous examples, platform unit tests access data from a datatable and so input data are not defined by columns in the unit test datatable but, instead, by way of a separate datatable. We will see how to access this datatable in step 3 but for this step, note that additional columns will not be needed.

**Step 3:** Launch the formula editor for column **test** by clicking on the formula icon (as shown in Figure 4b) then, in the drop zone, toggle to editor mode (see Figure 6a). Notice that there is

a comment that indicates what needs to be done to complete the formula. Unlike datatable unit tests for functions, this is a low code endeavor, so we will remain in editor mode to complete the steps outlined in the comment.

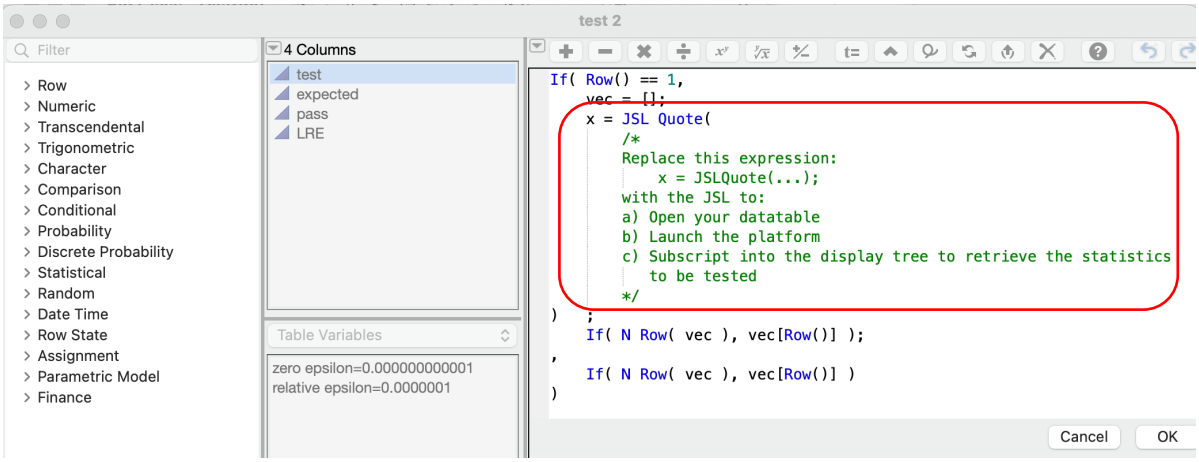



**Figure 6a: Formula editor**

a) For this example, we will use BigClass.jmp from the Sample Data Folder in JMP (see the Help menu) and so we will need the following statement.

```
Open( "$SAMPLE_DATA/Big Class.jmp" );
```

b) The most convenient way to determine the statement needed to launch a platform is to request it from the platform. All JMP platforms have a *Red Triangle Menu* that contains a *Save Script* menu option (see Figure 6b). This option generates a statement that will reproduce the state of the platform, if the required datatable is the current datatable. So, to get the statement that we need, we will just open Big Class.jmp, launch the distribution platform, go to the *Red Triangle Menu*, choose the *Save Script* menu option, then the *To Clipboard* sub-menu option. The statement saved to the clipboard will be the following:

```
Distribution( Continuous Distribution( Column( :weight ) ) );
```

c) It turns out that the most convenient way to determine the statement needed to subscript into the display tree for any JMP report, is to request it from the platform. This can be done by context clicking on any part of a report that contains data and selecting the *Show Properties* menu option. For example, context clicking on the Summary Statistics column in Figure 6b, will result in the report shown in Figure 6c, where a *Properties* panel now augments the report. The *Box Path* outline node contains the needed statement (see Figure 6d). Before copying the statement to the clipboard make sure that you have selected XPath mode (see Figure 6d).

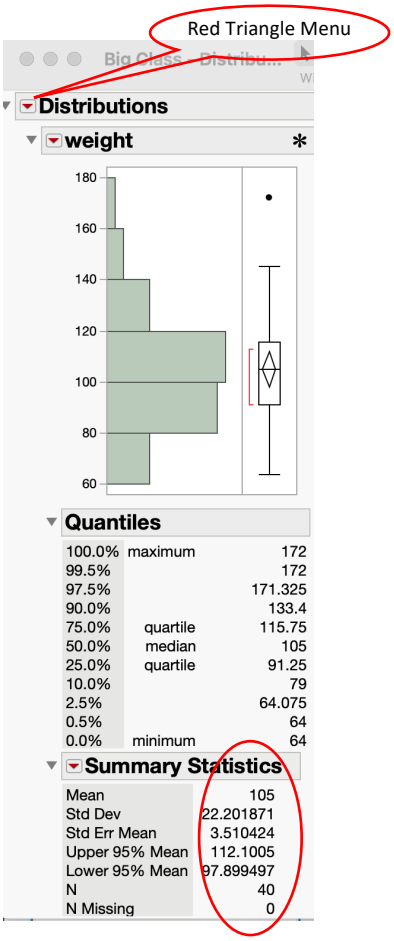



**Figure 6b: JMP report**

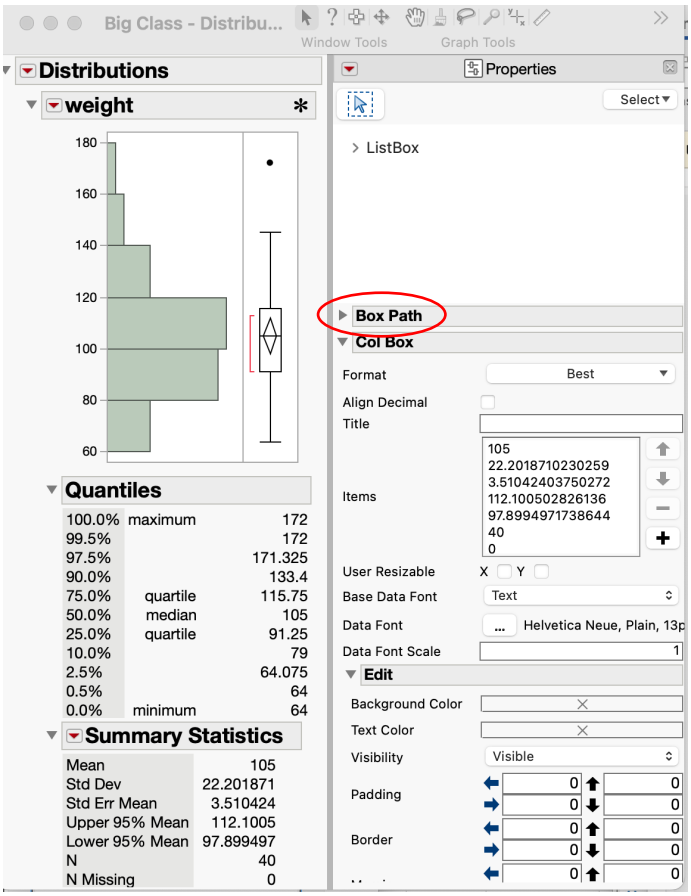



**Figure 6c:JMP report with Properties panel**

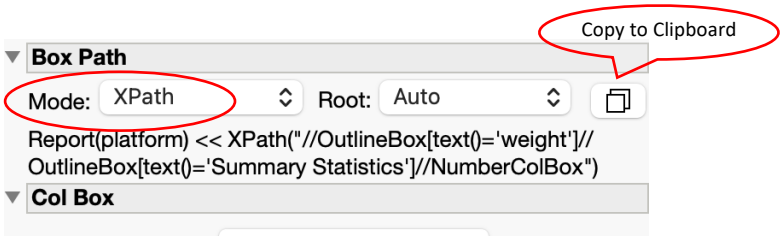



**Figure 6d: Box Path outline node**

Now that JMP has generated the statements that we need, let us now assemble them. We will do the assembly in reverse order, beginning with the XPath statement.

```
Report(platform) << XPath("//OutlineBox[text()='weight']//
      OutlineBox[text()='Summary Statistics']//NumberColBox")
```

Placeholder

**Figure 6e: XPath statement to retrieve summary statistics**

The platform argument in the XPath statement is just a placeholder (see Figure 6e) and we will just replace it with the platform launch statement from part b) above (see Figure 6f).

```
Report( Distribution( Continuous Distribution( Column( :weight ) ) ) ) <<
    XPath("//OutlineBox[text()='weight']//OutlineBox[text()='Summary Statistics']//NumberColBox")
```

**Figure 6f: Statement to launch platform and access summary statistics displaybox**

Let us now put it all together (see Figure 6g) and then replace the comment in Figure 6a by pasting the assembled statement into the formula editor (see Figure 6h). Most of the assembled statement was generated by JMP, we just needed to add a statement to open the datatable, add an assignment clause, so that the statistics retrieved from the displaybox are assigned to a variable, then add a statement to close the datatable.

```
dtRef = Open( "$SAMPLE_DATA/Big Class.jmp" );
vec = (Report( Distribution( Continuous Distribution( Column( :weight ) ) ) ) <<
    XPath(
        "//OutlineBox[text()='weight']//OutlineBox[text()='Summary Statistics']//NumberColBox"
    ))[1] << get asmatrix;
close(dtRef, nosave);
```

**Figure 6g: Assembled statement for formula editor**

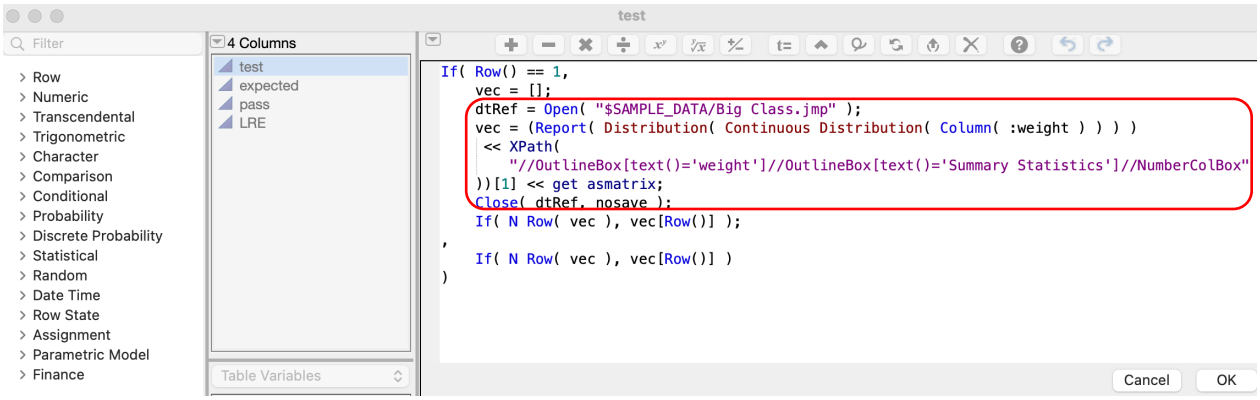



**Figure 6h: Formula editor**

The unit test is almost done but, as for our previous example, there is one more action that is needed. That is, click the "OK" button to save and dismiss the formula editor, then use the column information dialog to disable formula evaluation suppression (see Figure 4e) for each formula column (i.e., **test, pass** and **LRE**). Once formula evaluation is enabled, the formulas will evaluate for each row of the datatable (see Figure 6i). In this case, the pass column indicates that all unit tests are successful, that is, actual and expected results agree within the relative epsilon threshold value. Also, the **LRE** column indicates at least 10 digits of accuracy for the set of statistics chosen for evaluation, except for the case where the expected value is zero in which case LRE is undefined [4], a quite reasonable outcome [4].

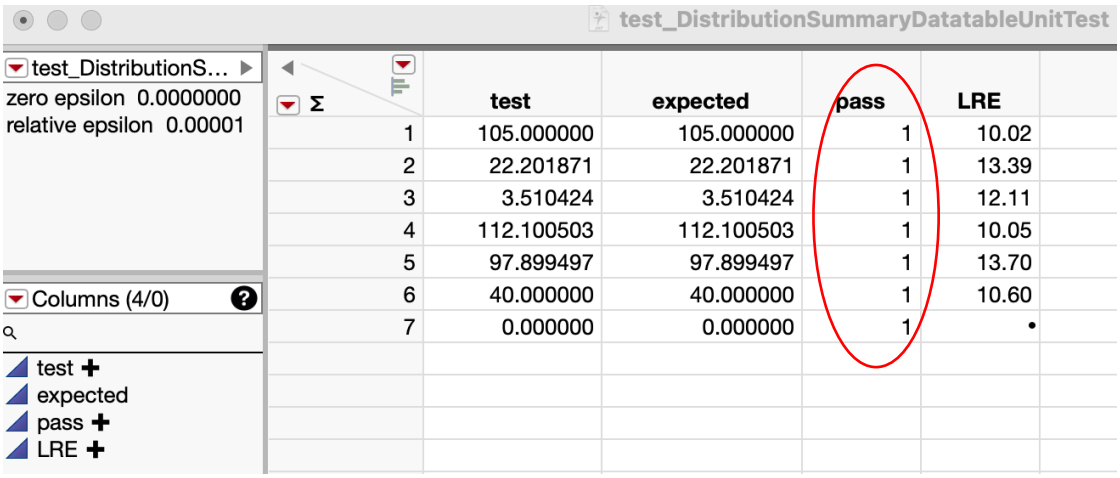



**Figure 6i: Completed and evaluated unit test**

A useful strategy for any datatable unit test is to add a column for comments. Like any software artifact, unit tests will evolve over time, usually because the software itself evolves, but often because the test cases themselves will be augmented or refined. A comment column can be used to capture pertinent notes about each test case which is especially useful (and perhaps necessary) for platform unit tests. Without a comment column it may be difficult to recall why the value being tested was chosen as a test case.

**A few parting comments**

It is worth pointing out at this point, that datatable unit tests are self-contained, complete, test artifacts. That is, they can be used independently of the unit test framework driver. The obvious advantage that this provides is that these unit tests can be shared with others who may not have the unit test framework but may be interested in the test cases that the datatable unit tests represent. Another advantage is that, by using the datatable as a testing mechanism, the benefits of datatables, such as sub setting, using the data analysis capabilities of JMP, are available. Nevertheless, the unit test framework provides necessary, additional value. It provides a mechanism to manage and execute suites of unit tests, both scripted and datatable unit tests. Since it knows the structure of unit tests, it can make use of this structure to provide useful summary information when tests fail. For example, when a datatable unit test for testing a function fails, the framework reports the exact statement that precipitated the failure, by traversing the formula and substituting formula arguments with the actual input values, thus easing the challenge of diagnosing the failure.

It is also worth making a few comments about the nature and rationale for testing. In his seminal textbook, *The Art of Software Testing* [6], Glenford Myers made the point:

"Testing is the process of executing a program or system with the intent of finding errors."

The key point here is that selecting test cases should not be an ad-hoc activity, rather it should be systematic and principled, with a singular goal, that is finding errors. The good news is that there are principled ways of selecting test cases to aid in ensuring this goal [5]. Furthermore, if failures occur during testing, it is then necessary to identify the errors that precipitated those failures. This is known as the fault localization problem and, fortunately, there are also principled ways to

identify those errors [11]. It turns out that JMP provides tools to select test cases and to analyze outcomes so that the root cause(s) of failures may be more readily identified [10].

**References**


1. Alamin, M. A. A., Uddin, G., Malakar, S., Afroz, S., Haider, T., & Iqbal, A. "Developer discussion topics on the adoption and barriers of low code software development platforms." *Empirical software engineering*, 28(1), (2023): 4.
2. Khankhoje, Rohit. "Beyond Coding: A Comprehensive Study of Low-Code, No-Code and Traditional Automation." *Journal of Artificial Intelligence & Cloud Computing. SRC/JAICC-160. DOI: doi. org/10.47363/JAICC/2022 (1)* 148 (2022): 2-5.
3. Khorram, Faezeh, Jean-Marie Mottu, and Gerson Sunyé. "Challenges & opportunities in low-code testing." *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. (2020): 1-10.
4. McCullough, Bruce D. "Assessing the reliability of statistical software: Part I." *The American Statistician* 52.4 (1998): 358-366.
5. Morgan, Joseph. "Combinatorial testing: an approach to systems and software testing based on covering arrays." *Analytic methods in systems and software testing* (2018): 131-158.
6. Morgan, Joseph & Xan Gregg. "Unit Tests: Automated JSL Testing." Whitepaper: JMP Statistical Discovery LLC. (2007) https://www.jmp.com/en_us/articles/unit-tests.html
7. Myers, Glenford J. *The Art of Software Testing*. John Wiley & Sons, 2006.
8. Sahay, A., Indamutsa, A., Di Ruscio, D., & Pierantonio, A. (2020, August). "Supporting the understanding and comparison of low-code development platforms." *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. (2020): 171-178.
9. SAS Institute Inc., JMP Scripting Guide, 2024, JMP Statistical Discovery LLC., https://www.jmp.com/support/help/en/19.0/?os=mac&source=application#page/jmp/create-custom-functions-transforms-and-formats.shtml
10. SAS Institute Inc., JMP Design of Experiments Guide, 2024, JMP Statistical Discovery LLC., https://www.jmp.com/support/help/en/19.0/?os=mac&source=application#page/jmp/covering-arrays.shtml#
11. Wong, W. E., Gao, R., Li, Y., Abreu, R., Wotawa, F., & Li, D. "Software Fault Localization: an Overview of Research, Techniques, and Tools." *Handbook of Software Fault Localization: Foundations and Advances*, (2023): 1-117.