**Automated Test Decision Framework**

Use the Automated Test Decision Framework tool to identify the highest-value components of a legacy codebase for which automated unit tests should initially focus. Use of this tool presumes a legacy codebase that has no, or insufficient, automated test code in its baseline. The Automated Test Decision Framework identifies the components of a legacy software system which most likely maximize Return on Investment (ROI) for resources allocated to introducing automated tests. Optionally, the Automated Test Decision Framework can also provide an estimate of the relative costs associated with introducing automated tests.

This document describes terminology, the simple steps required to run an analysis, and offers direction regarding interpretation of the results.

Contents

[1 Introduction 4](#_Toc523380030)

[1.1 Software System Characteristics 4](#_Toc523380031)

[1.1.1 Design Paradigm 4](#_Toc523380032)

[1.1.2 Architecture Depth 5](#_Toc523380033)

[1.1.3 Interface Typing 5](#_Toc523380034)

[1.1.4 Process/Methodology Maturity 5](#_Toc523380035)

[1.1.5 Development Team Technology Stack Experience 6](#_Toc523380036)

[1.1.6 Development Team Functional Domain Experience 6](#_Toc523380037)

[1.2 Component Characteristics 6](#_Toc523380038)

[1.2.1 Number of Invocations Over Time as Deployed 6](#_Toc523380039)

[1.2.2 Compute Resource Utilization 6](#_Toc523380040)

[1.2.3 Test Coverage Metrics 7](#_Toc523380041)

[1.2.4 Life-Spans 7](#_Toc523380042)

[1.2.5 Volatility 7](#_Toc523380043)

[1.2.6 Modularity 7](#_Toc523380044)

[1.2.7 Self-Descriptiveness 8](#_Toc523380045)

[1.2.8 Design Simplicity 8](#_Toc523380046)

[1.2.9 Consistency 8](#_Toc523380047)

[1.2.10 Anomaly Control 8](#_Toc523380048)

[1.2.11 Documentation 9](#_Toc523380049)

[1.2.12 Independence 9](#_Toc523380050)

[1.3 Complexity Metrics 10](#_Toc523380051)

[1.3.1 Coverage Percentage from Auto Generated Tests 10](#_Toc523380052)

[1.3.2 Source Lines of Code (SLOC) Count 10](#_Toc523380053)

[1.3.3 Mean Cyclomatic Complexity 10](#_Toc523380054)

[2 Assessment 10](#_Toc523380055)

[2.1 Characterize the Legacy Software System 11](#_Toc523380056)

[2.2 Enter and Rank the Components 11](#_Toc523380057)

[2.3 Enter the Complexity Metrics 12](#_Toc523380058)

[2.4 Result 13](#_Toc523380059)

[3 Example 14](#_Toc523380060)

[3.1 Step 1 – Characterize the Legacy Software System 14](#_Toc523380061)

[3.2 Step 2 – Enter and Rank the Components 15](#_Toc523380062)

[3.3 Step 3 – Enter the Complexity Metrics 17](#_Toc523380063)

[3.4 Result 17](#_Toc523380064)

List of Figures

[Figure 1 Design Paradigms 4](#_Toc523380065)

[Figure 2 Two Tier vs Three Tier Architecture 5](#_Toc523380066)

[Figure 3 Module Depiction 8](#_Toc523380067)

[Figure 4 Error Handling 9](#_Toc523380068)

[Figure 5 SQAE Characteristics and Sub-Characteristics Tree 12](#_Toc523380069)

[Figure 6 System Characterization 14](#_Toc523380070)

[Figure 7 System Characterization Dropdown 14](#_Toc523380071)

[Figure 8 Component Rankings: Blank 15](#_Toc523380072)

[Figure 9 Component Rankings: Names 15](#_Toc523380073)

[Figure 10 Component Rankings Entered 16](#_Toc523380074)

[Figure 11 Complexity Metrics Entered 17](#_Toc523380075)

[Figure 12 Results: Sort & Filter then Reapply 17](#_Toc523380076)

[Figure 13 Results 18](#_Toc523380077)

Revision History

|  |  |
| --- | --- |
| May 2017 | Initial Release |
| September 2018 | Updates for relative financial predictors and description of normalized rankings |
|  |  |

# Introduction

This section introduces the terms and characteristics used in the Automated Test Decision Framework in the order they are encountered when using the framework.

## Software System Characteristics

The system characteristics apply to the entire software system. The characteristics tracked by the Framework are on a simple three-value scale. Where the assessed value is neither apparent nor applicable, choose the ***Neither*** or ***Neutral*** value.

### Design Paradigm

Software design and development follows a style. There are many styles which are not always mutually exclusive. In the context of this framework, the values of interest are either *Object-Oriented* or *Procedural*. Object-Oriented — Computation is achieved by sending messages to objects; objects have state and behavior. Procedural programming is also known as structured programming. Procedural — Imperative programming with procedure calls[[1]](#footnote-1).

**Values:** Object-Oriented, Neither, or Procedural.

**Synonyms:** Programming paradigm, Design styles.

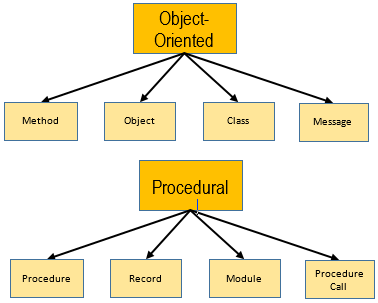


Figure Design Paradigms

### Architecture Depth

The structure of a software system is described by its layout, which may include layers. A tiered application architecture places the major components and stacks into many layers, such as: presentation (client/UI), domain and business logic (application), and data access (database). Generally, each layer only interfaces with the layer above and below it. A flat system (single tier) has only one layer and the components within the system can interface with all the other components.

|  |  |
| --- | --- |
| Related image | Image result for single tier application |

Figure Two Tier vs Three Tier Architecture[[2]](#footnote-2)

**Values:** Tiered, Neutral, or Flat.

**Synonyms:** Layered architecture, architecture pattern, multitier architecture.

### Interface Typing

For the purposes of this Framework, the interfaces between components and modules of the system are classified into general levels. Strongly typed interfaces are structured in terms of the payloads of data that may cross an interface boundary; strongly typed interfaces are considered safe but inflexible. Weakly typed interfaces are more free-form in terms of the interface data payloads; weakly typed interfaces are considered more flexible and extensible.

**Values:** Strong, Neutral, or Weak.

**Synonyms:** Interface Coupling.

### Process/Methodology Maturity

Software development processes and methodologies, and the tools and technologies that support them, differ widely. There are many industry best practices, and the over-arching field of Software Engineering considers processes and methodologies of the software lifecycle in the context of producing quality software. In this context, the Process/Methodology Maturity considers only the presence of institutionalized processes and methodologies. There are no specific measurable discriminators. Simply self-assess to determine if there are processes in place, if they are followed by the development team(s), and if they do or do not work well.

**Values:** Mature, Neutral, or Immature.

### Development Team Technology Stack Experience

Is the development team well experienced with the software technologies that are used in the software system? Without quantifying the parameters that define *Experienced*, *Neutral* or *Inexperienced* in this context, perform a self-assessment. Contemplate and answer whether the development team possesses or lacks the requisite knowledge to best exploit the software technologies.

**Values:** Experienced, Neutral, or Inexperienced.

**Synonyms:** Subject Matter Expertise (Technology).

### Development Team Functional Domain Experience

Is the development team well experienced and knowledgeable with the software system’s operational usage in the context of completing a mission or a business task? Without quantifying the parameters that define *Experienced*, *Neutral* or *Inexperienced* in this context, perform a self-assessment. Contemplate and answer whether the development team possesses or lacks the requisite knowledge to understand the software system’s functionality as it relates to the software system’s actual application in a larger system or organizational architecture.

**Values:** Experienced, Neutral, or Inexperienced.

**Synonyms:** Subject Matter Expertise.

## Component Characteristics

Use of the Automated Test Decision Framework presumes that a software system may be decomposed or defined by a finite set of components.

The following section describes the component characteristics and provides guidance on how to rank the components.

### Number of Invocations Over Time as Deployed

This characteristic looks at how frequently the component is executed or invoked when the system is running. This characteristic is specifically measurable; e.g., for Java, the *JConsole* tool can be used to report the number of invocations.

**Values:** Ranking from 1 - n (most invoked – least invoked).

### Compute Resource Utilization

Compute resources include memory, CPU, network, storage channels. This characteristic is specifically measurable; e.g., G*anglia* or the *Tivoli* suite tools provide metrics detailing resource utilization over time.

**Values:** Ranking from 1 - n (most resources consumed – least consumed).

### Test Coverage Metrics

There may be existing tests—manual test procedures or an inadequate base of automated tests—in use for legacy software. Measure percent coverage when executing the existing legacy tests utilizing a test coverage tool; e.g., JaCoCo, Cobertura, Emma, Gcov.

**Values:** Ranking from 1 - n (least coverage – most coverage).

### Life-Spans

The time remaining until a component will be replaced or retired, if known.

**Values:** Ranking from 1 - n (longest life-span – shortest life-span).

### Volatility

Volatility is the rate of change over time. Changes to a given component may be the result of updated functional scope, defects identified, and the prioritization of the scope or defects relative to those priorities for other components. What is the expected rate, density, and extent of changes? Historical volatility metrics-- not just counts but rates and velocity trends-- may be a predictor for future volatility. For example, the historical measure of cumulative SLOC deltas for all commits over a period of time may be a viable measure of historical volatility.

**Values:** Ranking from 1 - n (most expected volatility – least expected volatility).

### Modularity[[3]](#footnote-3)

In a modular design, the functionality is divided into independent, typically small and simple, pieces or *modules*. The opposite of a modular design is a monolithic design, where independence between modules does not exist—the modules are either tightly coupled to one another or there is no decomposition of the design to smaller constructs. Modularity may therefore be measured as inversely proportional to the degree of coupling. Tools exist which measure coupling of software components; e.g., PMD measures coupling violations at the class level[[4]](#footnote-4).

**Values:** Ranking from 1 - n (least modular – most modular).

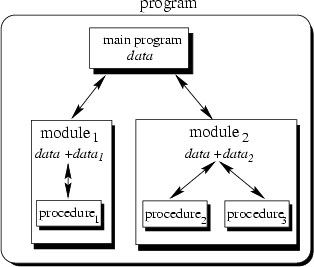


Figure Module Depiction[[5]](#footnote-5)

### Self-Descriptiveness

Self-descriptive software provides the naming constructs, comments, and descriptions in the code to facilitate the analysis and understanding of the code. White space and use of unambiguous names may be used to help the legibility and comprehensibility of the code. For example, the completeness of Javadoc or Doxygen comments for interfaces, classes, methods and their parameters could be used to measure the self-descriptiveness, particularly for a public-facing Application Program Interface (API).

**Values:** Ranking from 1 - n (least descriptive – most descriptive).

### Design Simplicity

This component characteristic relates to the readability and traceability of the code. Simply designed software is easy to understand in terms of its intended behavior. Simple code designs yield source code that is easily read with intended behavior easily understood. One traditional measure of code complexity is cyclomatic complexity. Many code analysis tools provide measures of cyclomatic complexity; e.g., Cobertura, SonarQube.

**Values:** Ranking from 1 - n (most complex – least complex).

### Consistency

Consistency implies uniform styles in project code and documentation; high consistency usually requires adherence to documented standards. Measures of consistency may be based on density of style violations. Style compliance tools can be applied to identify style violations; e.g., checkstyle.

**Values:** Ranking from 1 - n (least consistent – most consistent).

### Anomaly Control

This component characteristic concerns the error handling and exception processing. Sufficient anomaly control prevents errors from crashing the system and keeps the system in a stable and recoverable state when errors are encountered. Proper ranking of this component characteristic requires contextual knowledge of the software design. Identify anomaly handling within a codebase by searching for language-specific keywords, e.g., try, throw, catch, exception.

**Values:** Ranking from 1 - n (least sufficient anomaly control – most sufficient anomaly control).

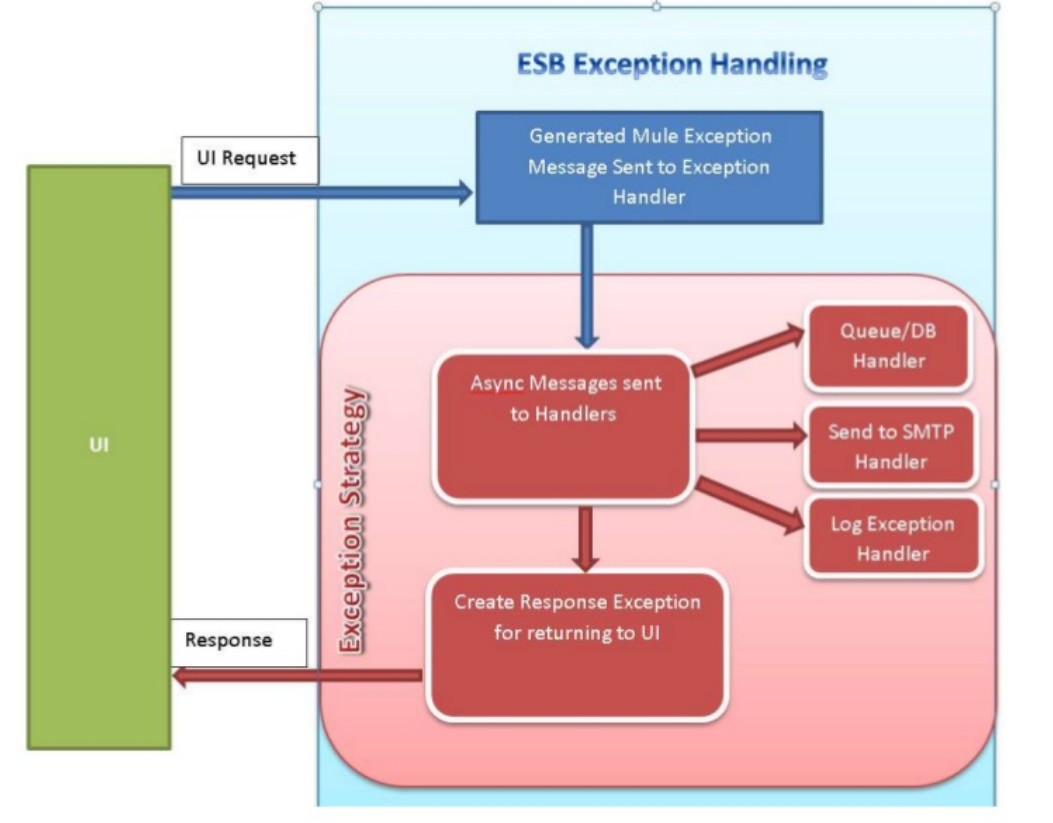


Figure Error Handling[[6]](#footnote-6)

### Documentation

Documentation supports the future maintenance, porting, and modification of code. Sufficient documentation provides a clear understanding of how the component functions. Examples include: requirements, specifications, design artifacts, etc. In ranking documentation sufficiency, consider not only the presence and breadth of documentation, but its readability, accuracy, and utility as it relates to the component.

**Values:** Ranking from 1 - n (least sufficient documentation – most sufficient documentation).

### Independence

Independent software is not tied to any specific host environment which would make it difficult or impossible to migrate, evolve, or enhance the software. In ranking independence, consider whether ties to operating systems, extensions, interfaces, and other components are minimized and facilitate future code migration, evolution and/or enhancements. The prevalence of environment specific keywords may indicate dependence on operating system, software container(s), or proprietary interfaces, e.g., Runtime, exec, Process.

**Values:** Ranking from 1 - n (least independent – most independent).

## Complexity Metrics

To get an estimate of the relative cost associated with introducing sufficient automated tests on a component-by-component basis, calculate and record these explicit metrics.

### Coverage Percentage from Auto Generated Tests

There are several industry tools, both commercially licensed and open source, that will automatically generate a set of unit tests against a given source code baseline. Examples of such tools for a Java codebase include Randoop, EvoSuite, and Jtest. Similarly, there are tools that measure the branch and line coverage of a source code baseline; e.g., Cobertura, Emma, JaCoCo.

This metric entails using both a test generator and a test coverage tool: automatically generate a set of unit tests for a source codebase, then execute that set of tests against the source code with a coverage tool applied. On a component-by-component basis, this value is the percent of component source code *covered* via execution of the automatically generated tests. Specifically, it is the lower of branch or line coverage.

**Values:** Absolute percentage in the range [0%, 100%]

### Source Lines of Code (SLOC) Count

There are numerous SLOC Counters available in the public domain; e.g., Universal Code Counter (<http://csse.usc.edu/ucc_new/wordpress/>). Do not use a simple text file line counter such as ‘wc -l’. SLOC is one metric for the size or volume of a source code component.

**Values:** A positive integer.

### Mean Cyclomatic Complexity

Cyclomatic Complexity is a theoretical Computer Science term that measures paths through a unit of source code. The algorithms to compute cyclomatic complexity typically rely on graph theory. Apply any cyclomatic complexity measure (algorithm or tool) consistently across all components. One prevalent algorithm is attributed to McCabe, and it is calculated by a tool like Cobertura. The mean cyclomatic complexity for a component is calculated across all methods or functions or procedures or sub-routines of that component.

**Values:** Real number greater than or equal to 1.00.

# Assessment

The Automated Test Decision Framework consists of three major steps. First, characterize the legacy software system along the system characteristics. Next, rank the components of the system along the component characteristics. Finally, and optionally, calculate and record complexity metrics. The assessment is facilitated by an Excel Spreadsheet.

## Characterize the Legacy Software System

First, classify the software systems’ characteristics, located in the Excel tab labeled *Step 1.* The system characteristics (described in detail in Section 1.1, and shown below) are classified along a three-value scale. Perform a self-assessment of the software system, as a whole, by selecting the best value that describes the legacy software system for each characteristic. Where the assessed value is neither apparent nor applicable, choose the *Neither* or *Neutral* value.

Table Initial System Characterization

|  |  |  |  |
| --- | --- | --- | --- |
| **Characterize The Legacy Software System** | | | |
| **Characteristic** | **Classification** | | |
| Design Paradigm | OO | Neither | Procedural |
| Architecture Depth | Tiered | Neutral | Flat |
| Interface Typing | Strong | Neutral | Weak |
| Process/Methodology Maturity | Mature | Neutral | Immature |
| Dev Team Technology Stack Experience | Experienced | Neutral | Inexperienced |
| Dev Team Functional Domain Experience | Experienced | Neutral | Inexperienced |

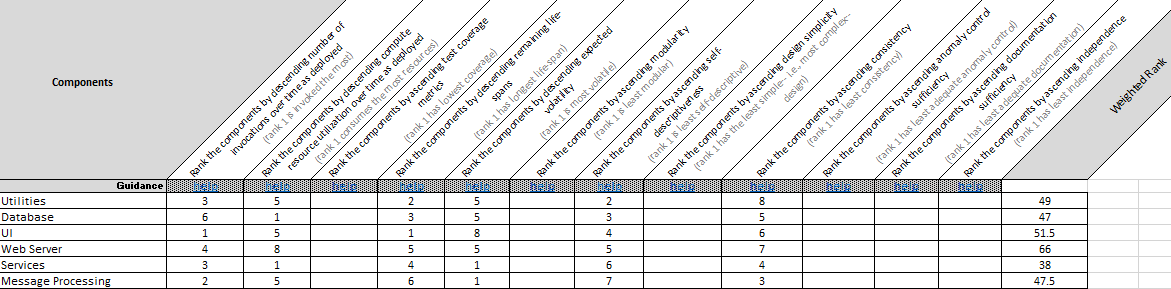
## Enter and Rank the Components

In the next step, advance to the tab labeled *Step 2*. Component names should be entered in the Components column. Rank the software components for each of the twelve characteristics described in Section 1.2. Each numerical ranking goes in the respective characteristic column. Rank the components only for those characteristics that are applicable; not all component characteristics will apply to all situations.

For every component characteristic, the system components may be ranked ordinally from 1 to n where n is the number of components. There may be *ties*, e.g., 1-2-3-3-3-6-6-8. The rankings may also be *normalized*, such that rankings imply a distribution of values. For example, consider a notional system with four components: a, b, c, d. Now consider the characteristic “remaining life-spans”. In this example, components a, b, c, and d have remaining life-spans of 6, 12, 24, and 48 months respectively. Instead of ordinal rankings 1, 2, 3, 4, these could be normalized to 1, 2, 4, and 8. If normalizing the rankings, normalize the rankings for all component characteristics for consistent results.

Note that not all the component characteristics may be rank-able for all software systems; the Framework considers only those component characteristics for which rankings are provided.

Table Component Rankings



The rankings for the component characteristics: Modularity, Self-Descriptiveness, Design Simplicity, Consistency, Anomaly Control, Documentation, and Independence, may be obtained from Software Quality Assurance Evaluation (SQAE) results—if SQAE, *Code Wash*, or *Code Spin* results are available on a per component basis. Figure 5 depicts the SQAE quality characteristics and the associated quality sub-characteristics.

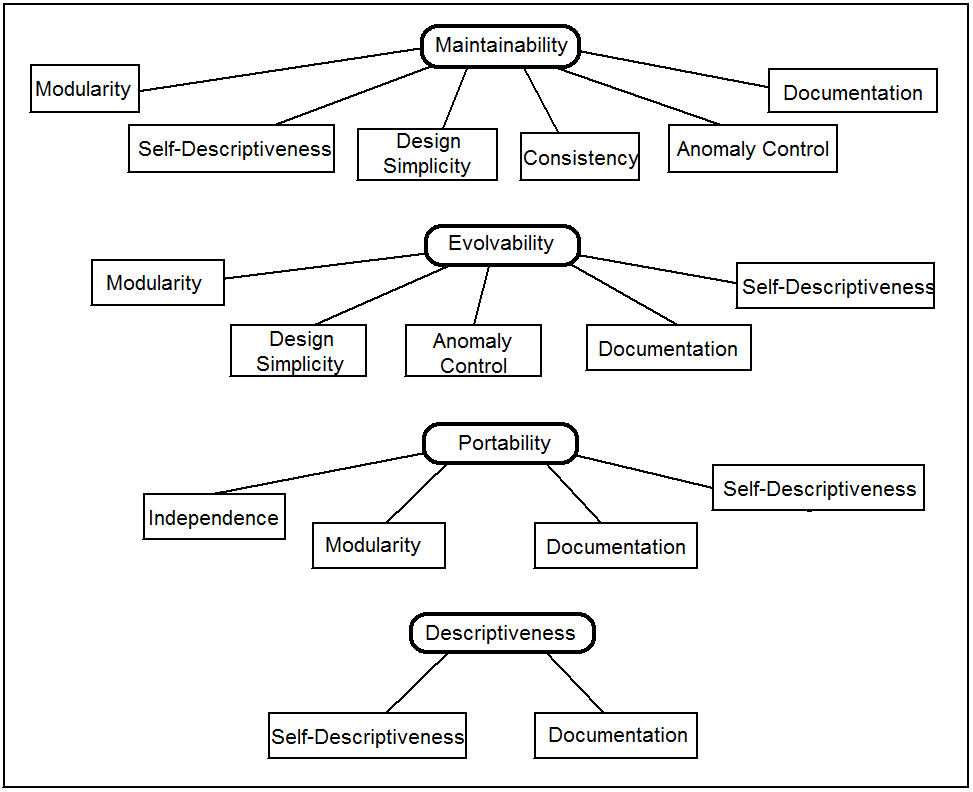


Figure SQAE Characteristics and Sub-Characteristics Tree

Weighted rank is calculated based on the assessed values for System Characteristics and Component Rankings. It should not be edited.

Component names should be unique. When ranking the components along a characteristic, note the ordering, as some rankings are ascending and others are descending. Note that components can be tied in the ranking for a given Component Characteristic. There may also be cases where the components cannot be ranked for a given characteristic.

## Enter the Complexity Metrics

Complexity Metrics are optional. Their entry will yield relative cost measures to engineer and introduce sufficient automated tests on a component-by-component basis. Advance to the tab labeled *Step 3*. Enter the data values as calculated by the associated tools: Coverage Percentage from Automatically Generated Tests, SLOC Count, and Mean Cyclomatic Complexity.

Component names should auto-populate, as they were entered on the previous tab. Relative Cost, and Relative Cost (Normalized) are calculated values; these should not be edited.

Table Complexity Metrics



## Result

The results are shown in the tab labeled *Result*. After all assessments are completed, the results can be updated by clicking the “Sort and Filter” Excel feature in the tab to correctly show the components in order of expected ROI when introducing automated tests, i.e., the first item on the list is expected to produce the greatest ROI when introducing automated tests.

If Complexity Metrics have been entered, the Relative Cost to Introduce Automated Test is also calculated. The relative cost is an estimate of the relative level of effort to engineer and introduce a sufficient set of automated tests for each component. Use relative cost and ROI to aid in decision-making; e.g., in specific circumstances, if both ROI and relative cost are low, it may be advisable to prioritize introducing automated tests for such a component.

Table Result



# Example

## Step 1 – Characterize the Legacy Software System

First the user navigates to the Excel tab labeled *Step 1*, which is depicted in Figure 6.

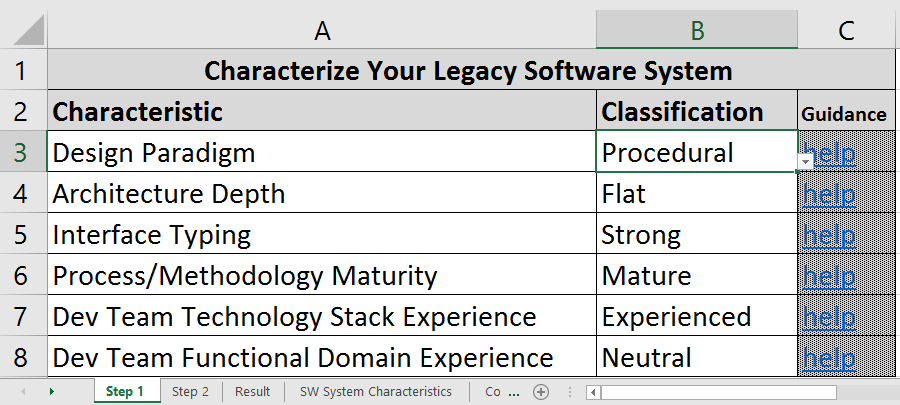


Figure System Characterization

The user clicks and sets the Classification value for each of the Characteristics. Each Classification is selected via drop-down list. For a specific example, Figure 7 depicts the selection of OO, i.e., Object-Oriented, as the design paradigm.

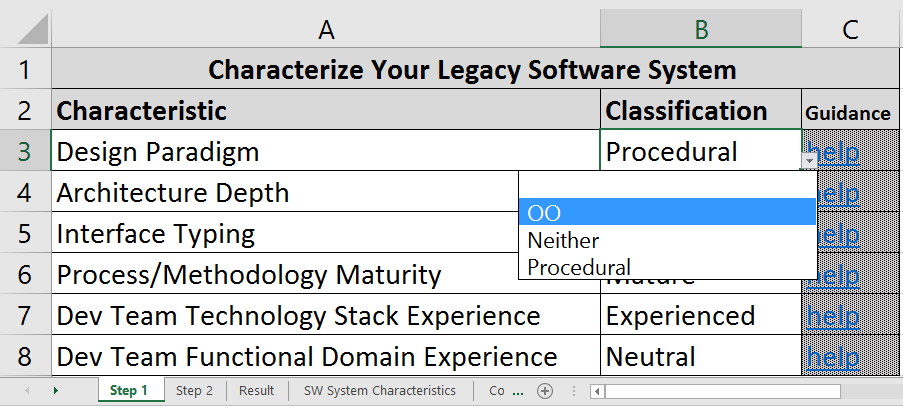


Figure System Characterization Dropdown

## Step 2 – Enter and Rank the Components

Next the user navigates to the Excel tab labeled *Step 2*, which is depicted in Figure 8.

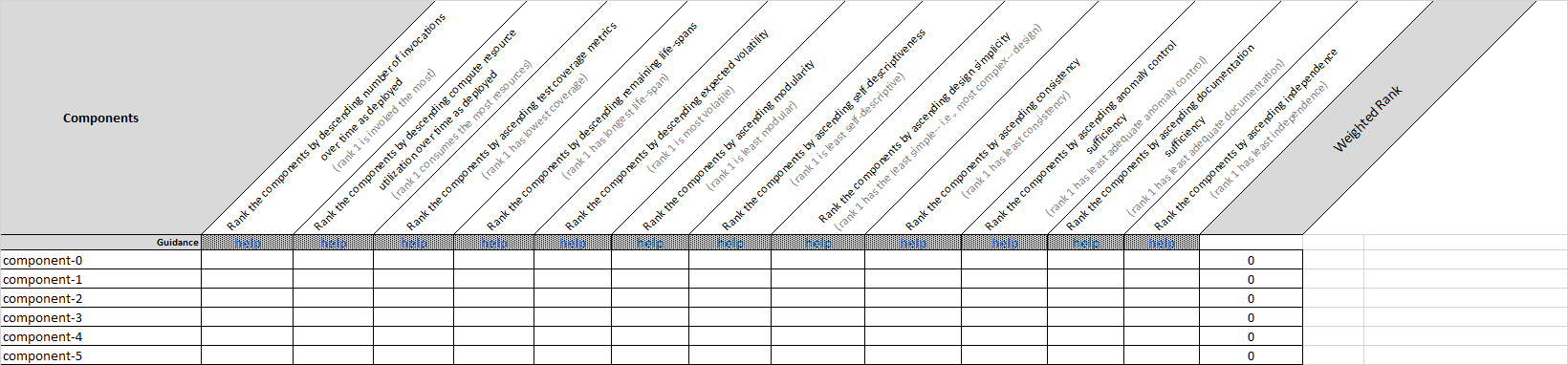


Figure Component Rankings: Blank

The first task on this tab is to enter the names of the software components. In this example, there are six components. The user enters the names of all six components under the Components header as shown in Figure 9.



Figure Component Rankings: Names

Next the user enters the ranks for each of the component characteristics which are applicable; not all component characteristics will apply to all situations. As there are six components in this example, the user ranks the components from one to six if applying ordinal rankings. Ties are allowed. In the hypothetical example depicted in Figure 10, the Component Characteristic of Compute Resource Utilization is ranked as 1 – Database, Services; 3 – Utilities, UI, Message Processing; 6 – Web Server. So, in this example, Database and Services all use the same amount of compute resources, followed by Utilities, UI and Message Processing which all use approximately the same amount of compute resources but less than those ranked as 1; finally, Web Server component utilizes the least resources.

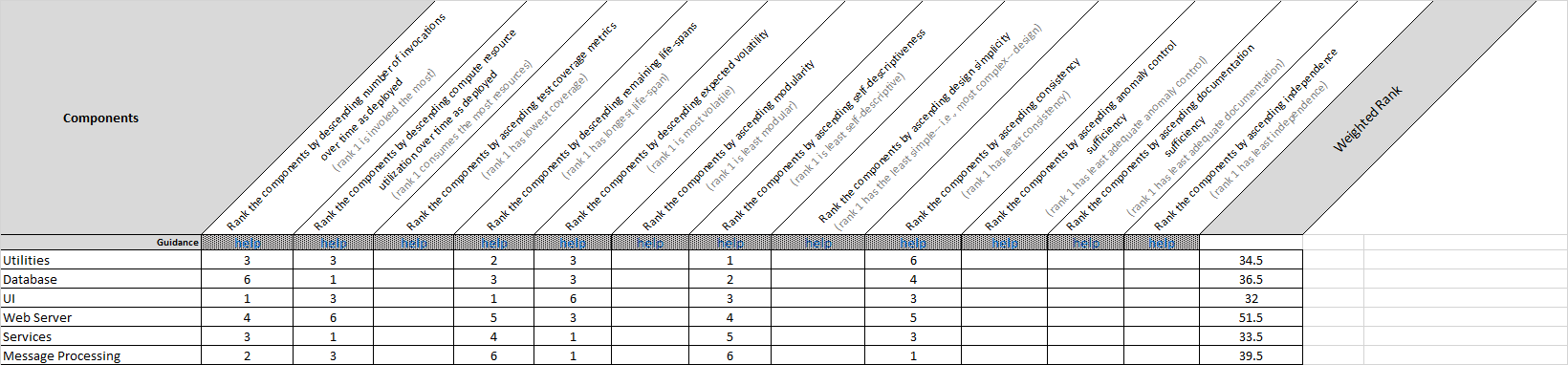


Figure Component Rankings Entered

## Step 3 – Enter the Complexity Metrics

Optionally, advance to the Excel tab labeled *Step 3*, component names will auto-populate on this tab. Enter the data values as calculated by the associated tools: Coverage Percentage from Automatically Generated Tests, SLOC Count, and Mean Cyclomatic Complexity. Enter data for all three columns for each component, or omit data entry on this tab altogether.



Figure Complexity Metrics Entered

## Result

Advance to the Excel tab labeled *Result*, and click the *Sort & Filter* button, then select the *Reapply* option as depicted in Figure 12. The results are shown in Figure 13.

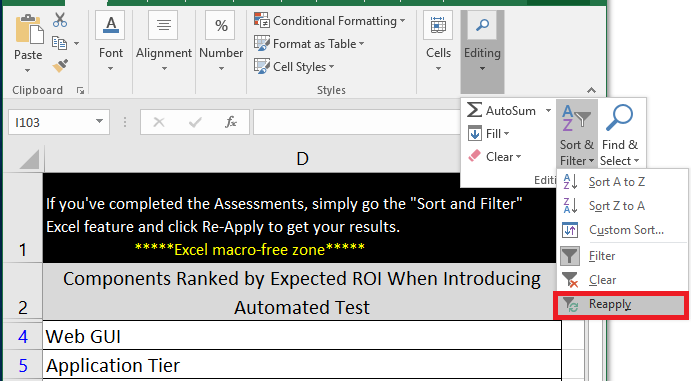


Figure Results: Sort & Filter then Reapply



Figure Results

1. <http://cs.lmu.edu/~ray/notes/paradigms/> [↑](#footnote-ref-1)
2. <http://flylib.com/books/en/2.642.1.12/1/> [↑](#footnote-ref-2)
3. The component characteristics: Modularity, Self-Descriptiveness, Design Simplicity, Consistency, Anomaly Control, Documentation, and Independence are taken from the MITRE Software Quality Assurance Evaluation framework. <https://www.mitre.org/research/technology-transfer/technology-licensing/software-quality-assurance-evaluation-sqae> [↑](#footnote-ref-3)
4. <http://pmd.sourceforge.net/pmd-4.3.0/rules/coupling.html> [↑](#footnote-ref-4)
5. Introduction to OOP Programming - Tiem.utk.edu [↑](#footnote-ref-5)
6. <http://www.slideshare.net/gdssrao/error-handling-framework-in-mule/6> [↑](#footnote-ref-6)