**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.

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

Contents

[1 Introduction 4](#_Toc482959521)

[1.1 Software System Characteristics 4](#_Toc482959522)

[1.1.1 Design Paradigm 4](#_Toc482959523)

[1.1.2 Architecture Depth 5](#_Toc482959524)

[1.1.3 Interface Typing 5](#_Toc482959525)

[1.1.4 Process/Methodology Maturity 5](#_Toc482959526)

[1.1.5 Development Team Technology Stack Experience 6](#_Toc482959527)

[1.1.6 Development Team Functional Domain Experience 6](#_Toc482959528)

[1.2 Component Characteristics 6](#_Toc482959529)

[1.2.1 Number of Invocations over time as deployed 6](#_Toc482959530)

[1.2.2 Compute Resource Utilization 7](#_Toc482959531)

[1.2.3 Test Coverage Metrics 7](#_Toc482959532)

[1.2.4 Life-Spans 7](#_Toc482959533)

[1.2.5 Volatility 7](#_Toc482959534)

[1.2.6 Modularity 7](#_Toc482959535)

[1.2.7 Self-Descriptiveness 8](#_Toc482959536)

[1.2.8 Design Simplicity 8](#_Toc482959537)

[1.2.9 Consistency 8](#_Toc482959538)

[1.2.10 Anomaly Control 8](#_Toc482959539)

[1.2.11 Documentation 9](#_Toc482959540)

[1.2.12 Independence 9](#_Toc482959541)

[2 Assessment 10](#_Toc482959542)

[2.1 Characterize the Legacy Software System 10](#_Toc482959543)

[2.2 Enter and Rank the Components 10](#_Toc482959544)

[2.3 Results 11](#_Toc482959545)

[3 Example 12](#_Toc482959546)

[3.1 Step 1 – Characterize the Legacy Software System 12](#_Toc482959547)

[3.2 Step 2 – Enter and Rank the Components 13](#_Toc482959548)

[3.3 Results 15](#_Toc482959549)

List of Figures

[Figure 1 - Programming Paradigms 4](#_Toc482959509)

[Figure 2 - Two Tier vs Three Tier Architecture 5](#_Toc482959510)

[Figure 3 Module Depiction 8](#_Toc482959511)

[Figure 4 Error handling 9](#_Toc482959512)

[Figure 5 - SQAE Characteristics and Sub-Characteristics Tree 11](#_Toc482959513)

[Figure 6 System Characterization 12](#_Toc482959514)

[Figure 7 System Characterization Dropdown 13](#_Toc482959515)

[Figure 8 Component Rankings: Empty 13](#_Toc482959516)

[Figure 9 Component Rankings: Names 14](#_Toc482959517)

[Figure 10 Component Rankings Entered 14](#_Toc482959518)

[Figure 11 Results: Sort & Filter then Reapply 15](#_Toc482959519)

[Figure 12 Results 15](#_Toc482959520)

Revision History

|  |  |
| --- | --- |
| May 2017 | Initial Release |
|  |  |
|  |  |

# 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 follows a style of programming. 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.

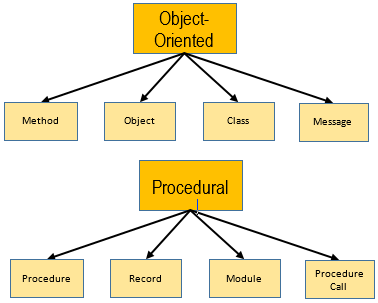


Figure - Programming Paradigms

### 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 or not 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 or not 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 test procedures—e.g., manual test procedures—in use for legacy software. Test coverage is specifically measurable when source code is instrumented by a test coverage tool, e.g., JaCoCo, Cobertura, EMMA, Gcov. While test coverage tools are often utilized in the context of automated test, if it is known which components are or aren’t covered by existing manual procedures, and to what extent they are covered, then rank the components by test coverage.

**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. 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. 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.

**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.

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

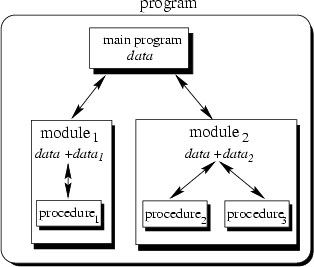


Figure Module Depiction[[4]](#footnote-4)

### 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.

**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. Complex code cannot be easily read and the intended behavior is not easily understood.

**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.

**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.

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

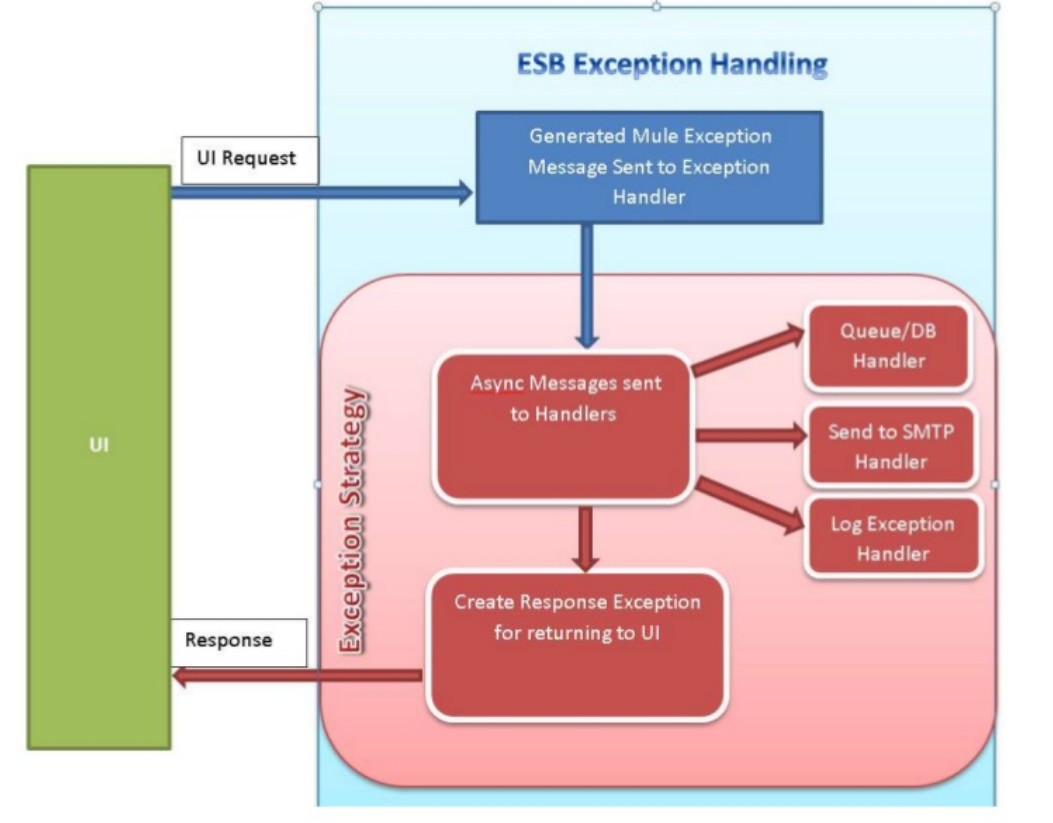


Figure Error handling[[5]](#footnote-5)

### 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

Independence means that the 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.

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

# Assessment

The Automated Test Decision Framework consists of two major steps. First, characterize the legacy software system along the system characteristics. Next, rank the components of the system along the component characteristics. The assessment is facilitated by an Excel Spreadsheet.

## Characterize the Legacy Software System

First, classify the software systems’ characteristics, located in first the Excel tab *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

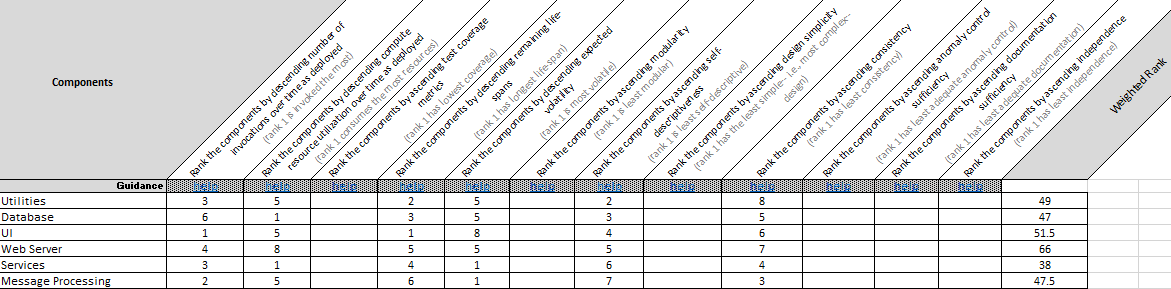
|  |  |  |  |
| --- | --- | --- | --- |
| **Characterize The Legacy Software System** | | | |
| **Characteristic** | **Classification** | | |
| Design Paradigm | OO | Neither | Procedural |
| Architecture Depth | Tiered | Neutral | Flat |
| Interface Typing | Strong | Neutral | Weak |
| Process/Methodology 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 *Step 2* tab. Component names should be entered in the Components column. Rank the software components for each of the eleven 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 to your circumstance.

For every component characteristic, the system components are ranked 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. Not all of 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 components characteristics: Modularity, Self-Descriptiveness, Design Simplicity, Consistency, Anomaly Control, Documentation, and Independence, may be obtained from Software Quality Assurance Evaluation (SQAE) results—if SQAE, or *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.

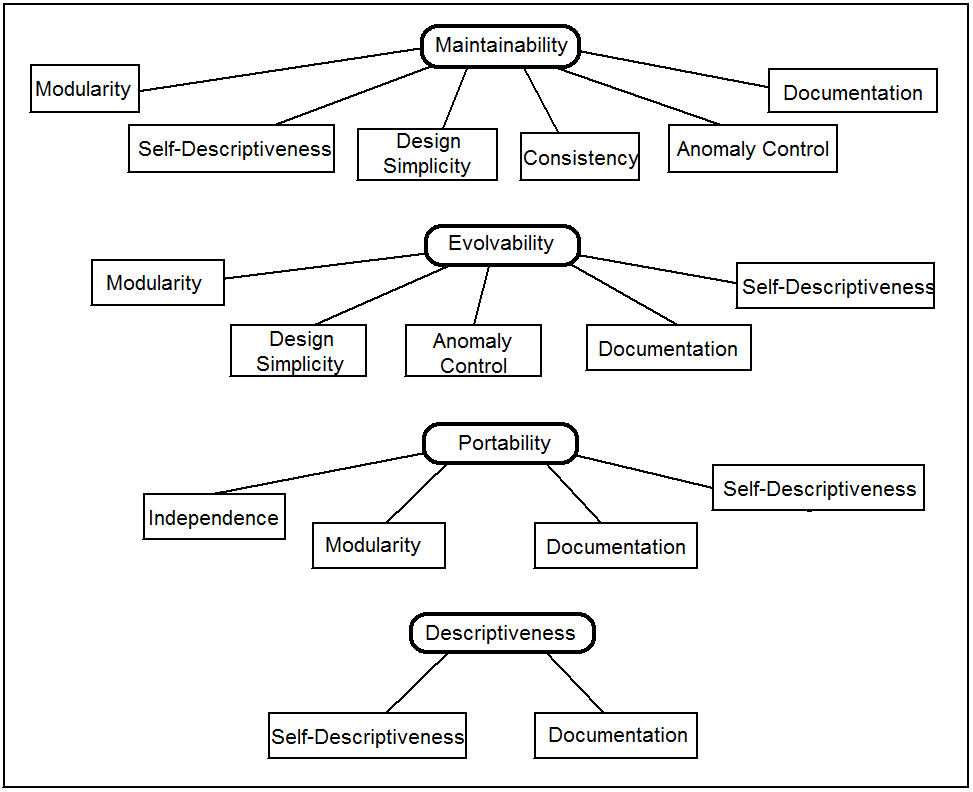


Figure - SQAE Characteristics and Sub-Characteristics Tree

Weighted rank is calculated based on the particular 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.

## Results

The results are shown in the *Results* tab. 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

Table Results

|  |
| --- |
| If you've completed the Assessments, simply go the "Sort and Filter" Excel feature and click Re-Apply to get the results.   \*\*\*\*\*Excel macro-free zone\*\*\*\*\* |
| Components Ranked by Expected ROI When Introducing Automated Test |
| Utilities |
| Database |
| UI |
| Web Server |
| Services |
| Message Processing |

# Example

## Step 1 – Characterize the Legacy Software System

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

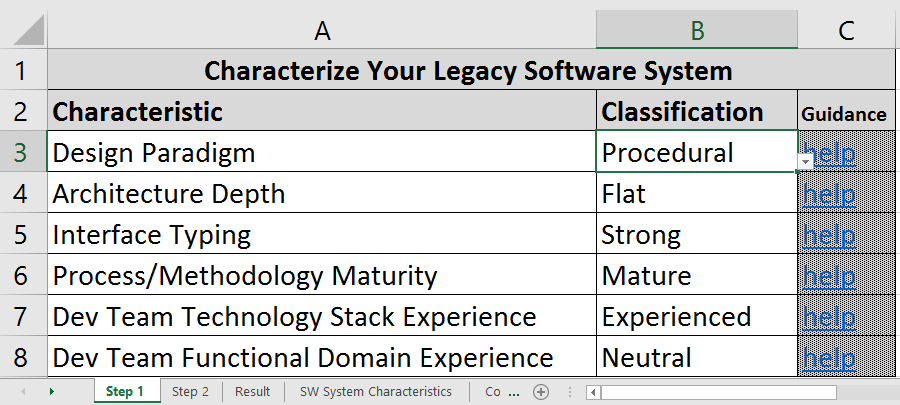


Figure System Characterization

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.

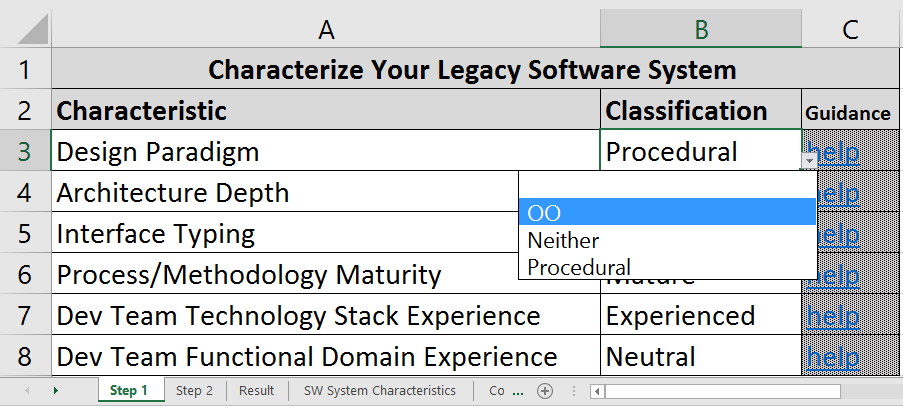


Figure System Characterization Dropdown

## Step 2 – Enter and Rank the Components

Next the user clicks the Excel tab for *Step 2*, which looks as depicted in Figure 8.

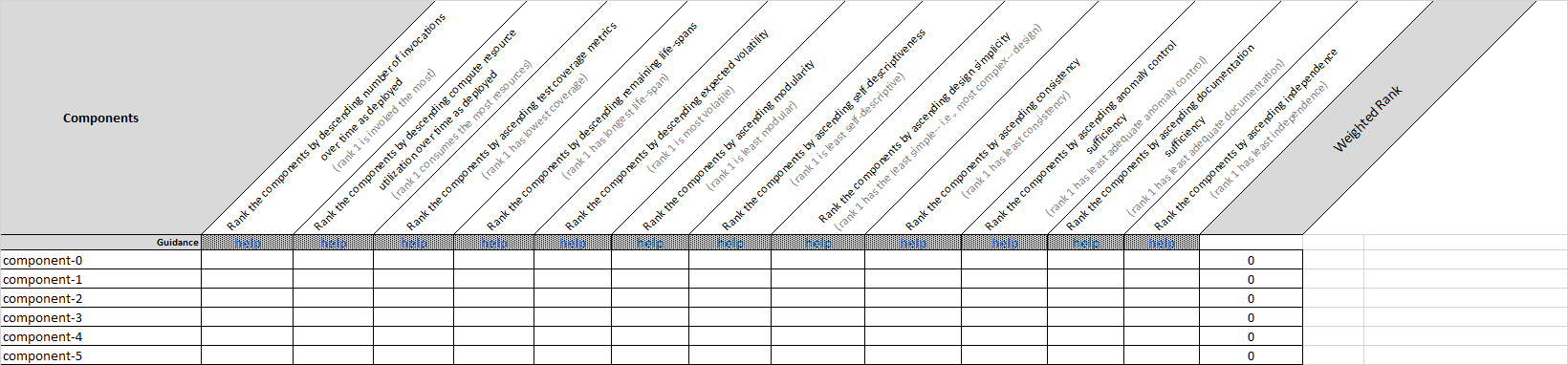


Figure Component Rankings: Empty

The first task on this tab is to add the names of the software components. In this example, there are six components. The user adds the names of all six components under the Components header as show 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. 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 approximate the same amount of compute resources but less than those ranked as 1; finally, Web Server component utilizes the least resources.

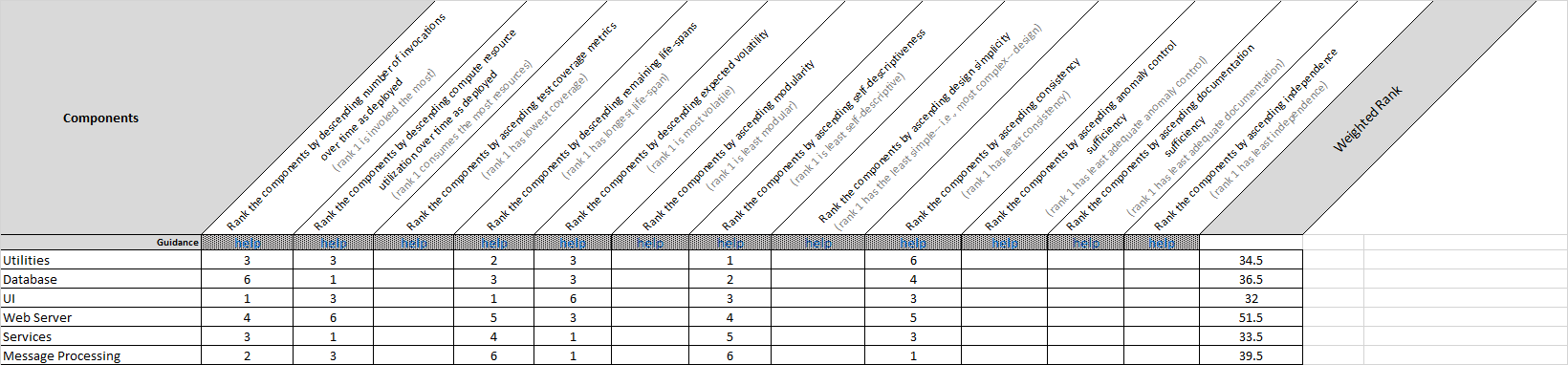


Figure Component Rankings Entered

## Results

Advancing to the *Results* Excel tab, and clicking *Sort & Filter* button, then selecting the *Reapply* option as depicted in Figure 11, the user views the results given in Figure 12.

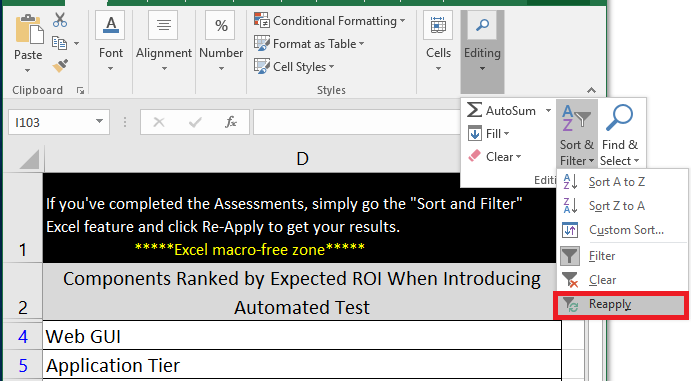


Figure Results: Sort & Filter then Reapply

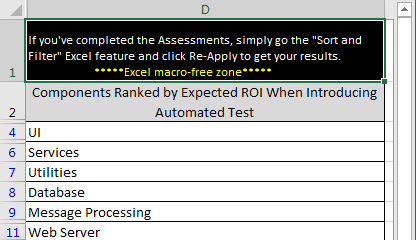


Figure Results

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. Introduction to OOP Programming - Tiem.utk.edu [↑](#footnote-ref-4)
5. <http://www.slideshare.net/gdssrao/error-handling-framework-in-mule/6> [↑](#footnote-ref-5)