Test Run

The Analytic Hierarchy Process

James McCaffrey

Contents

The System Under Analysis
Setting Up the Problem
Determining the Relative Weights of the Comparison Attributes
Compare the Builds on Each Attribute
Aggregate Weights to Produce Final Evaluation
Discussion
Conclusion

Most software testing takes place at a relatively low level. Testing an application's individual methods for functional correctness is one example. However, some important testing must take place at a very high level—for example, determining if a current build is significantly better overall than a previous build. In this column I'll show you a powerful technique called the analytic hierarchy process that will enable you to perform high-level quality analysis of your software systems.

The best way to illustrate where I'm headed is with a concrete example. Suppose you are developing a product that searches for files across a local network. You want to know if the current build is better overall than the previous builds, and if so, how much better? Comparing the overall quality of the builds with each other is not so easy because you have to consider many attributes such as performance, functionality, and so forth. Using the techniques I'll explain in this month's column will enable you to compute metrics like those shown in **Figure 1**.

Figure 1 Multi-Criteria Quality Analysis

	Performance		Functionality			Overall
	Startup	File Save	User Interface	Database	Network	
Build A	0.648	0.571	0.443	0.639	0.638	0.531
Build B	0.230	0.286	0.387	0.274	0.273	0.329
Build C	0.122	0.143	0.170	0.087	0.089	0.140

If you examine the table in **Figure 1** you'll see that the overall quality of each build consists of a performance attribute and a functionality attribute. Each of these attributes is composed of subattributes: Startup and File Save in the case of performance, and User Interface, Database, and Network in the case of functionality.

The technique I'll demonstrate is very flexible and can be adapted to any number of attributes with any number of subattributes. Based on build comparison data that I'll show you in the following sections, the overall quality metric for the current build, Build A, is 0.531 and the overall quality for the two previous builds is 0.329 and 0.140, respectively. I'll explain how to interpret these quality metrics later, but for now it's enough to say larger numbers are better. So, you have a fairly strong indication that the current build is significantly better than the previous two builds. The overall quality metrics for each build depend on the quality metrics for each attribute. In this example, for startup performance, Build A is best at 0.648, Build B is next at 0.230, and Build C is worst at 0.122.

In the sections that follow I will briefly describe the hypothetical Windows®-based application I am analyzing for quality, walk you through the analysis of the app using the analytic hierarchy process, explain how to interpret the resulting quality metrics, and conclude with a discussion of how you can adapt this technique to meet your own needs. I think you'll find that the ability to quickly evaluate the overall quality of a software system composed of many attributes is a valuable addition to your software testing, development, and management skill sets.

The System Under Analysis

Let's briefly look at the application under test so you'll understand the goal of quality analysis. The dummy search application is a simple form-based app. If I want to compare the overall quality of a particular build with previous builds, I have to take into account many system attributes. For the purposes of this column I'm using only performance and functionality, but the technique I'll show you can deal with any number of attributes. Although each software system is different, the most common attributes to consider when evaluating overall quality include accessibility, documentation, functionality, internationalization, module/API, performance, security, setup, scalability, stability, usability, and UI.

Of course there are many attributes, and each will have many subcomponents. In addition, many of these attributes overlap each other. But in practice, this list is a good place to start. This technique can deal with virtually any kind of software system—Windows-based applications, ASP.NET Web applications, class libraries, and so forth.

My goal is to produce a single value that represents the overall quality of my software system as a whole. There are several multi-attribute analysis techniques available, but the one I prefer is called the analytic hierarchy process. The analytic hierarchy process has been around since the early 1980s and has been used in a wide variety of fields. The seminal reference on the topic is the book *The Analytic Hierarchy*

Process for Decisions in a Complex World by Thomas Saaty (McGraw-Hill, 1980).

Without a structured technique, trying to evaluate the overall quality of different builds of even this small dummy application would be very difficult. After choosing your attributes, you'd have to determine how to compare each build on each attribute, how to quantify that information, and how to aggregate all that data into a meaningful metric. Then you'd have to decide how to interpret your results. A much better approach is to use the elegant analytic hierarchy process to quickly evaluate your builds.

Setting Up the Problem

The first step in the analytic hierarchy process is to set up the problem. In the case of multi-attribute software quality analysis this means deciding which builds you want to evaluate, and which attributes you will use as a basis for the evaluation. I have chosen to compare the three most recent builds of my hypothetical system—Build A, Build B, and Build C. The technique can accommodate any number of builds, but based on my experience three or four is a good number for evaluation purposes.

Next you must decide which attributes you will use to evaluate the overall quality of each build. Then, you'll decompose each attribute into subattributes. I decided to break down performance into startup performance and file save performance, and to break down functionality into user interface, database, and network functionality. The technique here allows you to decompose subattributes into even smaller sub-subattributes, and so forth, to any depth. As a general rule, you'll most often have top-level attributes and between two and five subattributes.

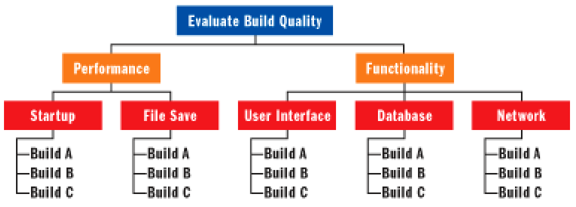


Figure 2 **Traditional Representation of the Problem**

Figure 2 shows a diagram traditionally used to represent the problem. I don't like it because it combines the goal with the build alternatives and the quality attributes in a way that is difficult for software engineers to interpret. I prefer to represent the problem in tabular form, as shown in **Figure 3**. Whichever representation format you decide to use, it must clearly summarize the problem in order to convey it to others.

Figure 3 Setup

Problem	
Evaluate build quality	
Alternatives	
Build A, Build B, Build C	
Attributes	
Performance	Startup
	File Save
Functionality	User Interface
	Database
	Network

Determining the Relative Weights of the Comparison Attributes

After I've set up my problem, the next step is to determine the relative weights of each of the comparison attributes. In my example so far, I have not determined exactly how important my chosen attributes are relative to each other. For instance, startup performance may be much more important to me than file save performance, or vice versa, or they may be equally important.

The analytic hierarchy process uses a pair-wise comparison technique. It works like this: I start with my highest-level attribute classifications, which in this example are performance and functionality. I create a table structure like so:

	Performance	Functionality
Performance		
Functionality		

Because I have two highest-level attributes, my table is size 2 × 2. If I had three highest level attributes (for example, an additional security attribute), then my table would have been size 3 × 3, and so forth. Next I will compare the relative importance of product functionality versus product performance using the table that is shown in **Figure 4**.

Figure 4 Comparison Values

Relative Importance	Value
Equal importance/quality	1
Somewhat more important/better	3
Definitely more important/better	5
Much more important/better	7
Extremely more important/better	9

For my example suppose I determine that functionality is between "somewhat more important" (value = 3) and "definitely more important" (value = 5) than performance, and so I get a value of 4 from the table. It is most common to use the values 1, 3, 5, 7, and 9, but you can use an intermediate value as I've done in my example. The loose interpretation of this value is that functionality is four times as important to me as performance in this software system.

Now the upper-left to lower-right diagonal elements of the table are comparing the relative importance of performance against itself, and the relative importance of functionality against itself. Using the values from the table in **Figure 4**, you can see I get values of 1 because each attribute has equal importance compared to itself. By definition, all the upper-left to lower-right diagonal elements of an analytic

hierarchy process comparison table will be 1's. This just leaves the upper-right cell of my comparison table which is the relative importance of performance versus functionality. Because I previously specified that functionality versus performance is a 4 (that is, that functionality is about 4 times as important to me as performance), the relative importance of performance versus functionality must be ¼ or 0.250. In other words, performance is ¼ as important as functionality to me. Placing these values in the table gives me the following:

	Performance	Functionality
Performance	1	0.250
Functionality	4	1

By definition, the values of cells in a comparison table that are diagonal (45 degrees) from each other will be mathematical inverses of each other. After completing the relative importance table, I've got to compute the "priority vector" from the table. Again, the process is best explained by example. I first sum each column, getting 5 and 1.250, respectively. Next I divide each entry in the relative importance table by its column sum. In this case I have 1/5 and 4/5 in the first column, and 0.250/1.250 and 1/1.250 in the second column. This gives me the matrix:

	Performance	Functionality
Performance	0.200	0.200
Functionality	0.800	0.800

The last step to determine the priority vector is to take the average of each row. This is (0.200 + 0.200) / 2 = 0.200 and (0.800 + 0.800) / 2 = 0.800, or expressed as a column vector:

Performance	0.200
Functionality	0.800

The calculations are very easy to produce using Microsoft Excel, or if you are more ambitious you can write a C# or Visual Basic .NET program to do it.

After producing the priority vector for the highest level of comparison attributes (performance and functionality), I need to compute priority vectors for each sublevel. The process is just like that for the higher-level attribute categories. I'll illustrate by showing how I calculate the priority vector for the functionality attributes (user interface, database, and network).

I start by making an empty 3×3 table. Next I compare the relative importance of each attribute in a pair-wise fashion. In this example I decided that user interface functionality is "somewhat more important" than database functionality (value = 3), user interface functionality is "much more important" than network functionality (value = 7), and database functionality is "slightly more important" (value = 2) than network functionality. I enter these three values into the comparison table, place 1s on the main diagonal, and enter inverses for corresponding cells across the main diagonal. Then I compute each column sum. At this point I have the table shown in the top of **Figure 5**. Then I divide each table cell by its corresponding column total to get the table at the bottom of **Figure 5**. And then I compute each row average to obtain the final priority vector, as shown here:

User Interface	0.681
Database	0.216
Network	0.103

Notice that each individual value in any priority vector will be between 0.0 and 1.0, and that the values in any priority vector will sum to 1 (subject to rounding error). What you end up with are so-called eigenvalues—normalized priority weights of each attribute. You can think of the priority weights as the values that are the most consistent with the pair-wise comparison values you enter. You could generate these weights directly, but one of the nice features of the analytic hierarchy process is the pair-wise comparison process that allows you to generate the weights in a consistent manner.

Figure 5 Two Tables

	User Interface	Database	Network
User Interface	1	3	7
Database	0.333	1	2
Network	0.143	0.500	1
Sum	1.476	4.500	10.000
	User Interface	Database	Network
User Interface	0.678	0.667	0.700
Database	0.226	0.222	0.200
Network	0.097	0.111	0.100

After I compute the priority vector for the functionality attributes, I will then compute the priority vector for the two performance attributes. The following shows my comparison table and the resulting priority vector:

	Startup	File Save
Startup	1	5
File Save	0.200	1
Startup	0.833	
File Save	0.167	

So at this point, I have set up my problem (evaluate build quality for Build A, Build B, and Build C), determined my comparison attributes (startup performance, file save performance, user interface functionality, database functionality, network functionality), and computed the priority vectors to be used for each level of the comparison attributes.

Compare the Builds on Each Attribute

After setting up the build quality evaluation problem and determining the priority vectors for the comparison attributes, the next step in the analytic hierarchy process is to perform a comparison of each build based on each of the lowest level comparison attributes. In this example that means I need to compare Build A versus Build B versus Build C on each of the five comparison attributes: startup, file save, user interface, database, and network. The build comparison process is exactly the same as the attribute comparison process. So, for startup performance for example, I compare each build using the criteria in the table in **Figure 4**. In my example I got the following values:

Startup	Build A	Build B	Build C
Build A	1	3	5
Build B	0.333	1	2
Build C	0.200	0.500	1

In other words, I determined that startup performance for Build A is "somewhat better" than it is for Build B, that startup performance for Build A is "definitely better" than that for Build C, and that startup performance for Build B is "slightly better" than that of Build C. Using the same algorithm as in the previous section (compute column totals, divide each cell by its column total, compute each row average), I establish the startup performance ranking vector:

Startup	
Build A	0.648
Build B	0.230
Build C	0.122

Using the same process for the remaining four comparison attributes, I compare builds and compute ranking vectors, as shown in **Figure 6**.

Figure 6 Quality Attributes and Ranking Vectors

File Save	Build A	Build B	Build C	File Save Ranking Vector	
Build A	1	2	4	Build A	0.571
Build B	0.500	1	2	Build B	0.286
Build C	0.250	0.500	1	Build C	0.143
User Interface	Build A	Build B	Build C	User Interface Ranking Vector	
Build A	1	1	3	Build A	0.443
Build B	1	1	2	Build B	0.387
Build C	0.333	0.500	1	Build C	0.170
Database	Build A	Build B	Build C	Database Ranking Vector	
Build A	1	3	6	Build A	0.639
Build B	0.333	1	4	Build B	0.274
Build C	0.167	0.250	1	Build C	0.087
Network	Build A	Build B	Build C	Network Ranking Vector	
Build A	1	4	5	Build A	0.638
Build B	0.250	1	5	Build B	0.273
Build C	0.200	0.200	1	Build C	0.089

Aggregate Weights to Produce Final Evaluation

After setting up the build comparison problem, computing the attribute priority vectors, and computing the build comparison ranking vectors, the next step is to aggregate all the intermediate data to produce the final evaluation metrics. As with the previous steps, it's easiest to explain by example. If I put the intermediate results into a table, I get one like you see in **Figure 7**.

Figure 7 Intermediate Results

	Performance (.200)		Functionality (.800)		
	Startup (.833)	File Save (.167)	User Interface (.681)	Database (.216)	Network (.103)
Build A	0.648	0.571	0.443	0.639	0.638
Build B	0.230	0.286	0.387	0.274	0.273
Build C	0.122	0.143	0.170	0.087	0.089

The final quality value for Build A is:

```
(.200)(.833)(.648) +
(.200)(.167)(.571) +
(.800)(.681)(.443) +
(.800)(.216)(.639) +
(.800)(.103)(.638) = 0.531
```

In other words, the final quality metric for each build is the weighted sum of its attribute rankings. In the same way, the final quality value for Build B is:

```
(.200)(.833)(.230) +
(.200)(.167)(.286) +
(.800)(.681)(.387) +
(.800)(.216)(.274) +
(.800)(.103)(.273) = 0.329
```

And the final quality value for Build C is:

```
(.200)(.833)(.122) +
(.200)(.167)(.143) +
(.800)(.681)(.170) +
(.800)(.216)(.087) +
(.800)(.103)(.089) = 0.140
```

At this point I can now interpret my build quality. Because the ranking vector values sum to 1.0 you can roughly compare each build on a percentage basis. In this example, given the three builds, Build A is loosely 0.53 - 0.33 = 0.20, or 20 percent better than Build B. However this interpretation is somewhat misleading because it only applies when all three builds are taken into account. In other words, if I added a fourth build to my analysis, my quality vector would change and Build A would be less than 20 percent better than Build B. And let me caution you that the 3-decimal precision of the data is somewhat deceptive. Because the input data derived from the table in **Figure 4** is relatively crude, it's a good idea to compare builds using only two decimals at most.

In my opinion the best way to interpret the results of build quality using the analytic hierarchy process is to view it strictly as trend data. Here you can see a relatively clear improvement trend. Based on my experience, as a general rule of thumb when comparing three builds, a difference of 0.15 or more is a significant (in the normal sense of the word, not in the statistical sense) difference. After you have some experience working with the analytic hierarchy process in your environment you'll gain an understanding of the metrics you get.

Discussion

As I've shown, the analytic hierarchy process is an elegant and powerful method for quantifying the overall quality of software systems. There are many other techniques you could use for meta-quality analysis, but the analytic hierarchy process has several appealing characteristics. The use of a subjective scale ("much more important") rather than a quantitative scale is particularly useful in software quality meta analysis. For example, suppose you are measuring performance. A difference of 2 seconds versus 3 seconds (50 percent difference) for a file save operation may not be equally as important as a 4 second versus 6 second (also 50 percent) time difference. Also useful is the pair-wise comparison technique used by analytic hierarchy process. It's usually much easier to compare two items at a time than to compare many items all at once.

There are certain disadvantages to using the analytic process to measure your build quality. As I mentioned, the subjective scale is very useful in software quality analysis but precisely because the scale is subjective, it is subject to human error. Take for example my comparison of the three builds based on network functionality:

Database	Build A	Build B	Build C
Build A	1	3	6
Build B		1	4
Build C			1

If you examine the data closely you'll see it is not entirely consistent. If Build A is 3 "good units" better than Build B, and Build A is 6 "good units" better than Build C, then you would think Build B should be exactly 3 "good units" better than Build C, but I rated Build B as 4 relative to Build C.

The analytic hierarchy process has a neat way to estimate a so-called inconsistency index for each comparison matrix. Showing you how to compute an inconsistency index estimate is outside the scope of this column, but it is quite easy to calculate. For the comparison matrix shown earlier, the inconsistency index is 0.05—which is below the generally accepted threshold value of 0.10. And, in fact, my comparison of the three builds based on network functionality is very inconsistent and in a production scenario I would investigate and revise those comparisons. Some variations of the analytic hierarchy process modify the attribute ranking vector values based on the corresponding inconsistency index.

Another disadvantage of the analytic hierarchy's subjective scale is that it is vulnerable to human psychology—there is a tendency to want to see improvement in builds and so you must be careful not to subconsciously inflate rankings of more recent builds. As always, your final metrics will only be as good as your input data.

In my opinion the biggest disadvantage of using the analytic hierarchy process is that the number of comparison tables can become very large if you use a lot of comparison attributes. This can lead to a tendency to exclude valid comparison attributes in order to keep the number of calculations manageable. In this column I used Excel to manage the analytic hierarchy process, but there are several excellent commercial tools available that simplify the data input process, perform calculations automatically, and

have nice reporting capabilities.

Conclusion

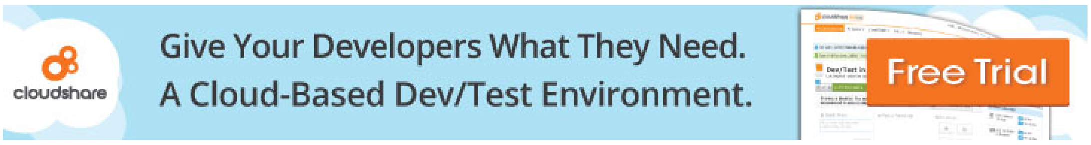
Although the analytic hierarchy process has been the subject of many research papers and the general consensus is that the technique is both technically valid and practically useful, there are critics of the method. My view on the matter is that you should use the analytic hierarchy process as just one component of your overall build quality analysis. It's a valuable supplement to other objective and subjective techniques that measure product quality. In general you should not rely completely on any single software system quality metric or technique.

Here I've shown you how easy and powerful the technique is when used to monitor build quality. Based on my experience, using the analytic hierarchy process to measure overall software quality represents a significant enhancement to traditional techniques like bug count metrics and milestone checklists.

Furthermore, you can use the technique in other areas. On one project I used it for competitive analysis to assess the overall quality of my product relative to major competing products. I've heard of other testers using similar methodologies to report on the health of a new build of a product based on weighting the results of automated build verification tests. As the software development environment continues to mature, techniques like the analytic hierarchy process will become increasingly important components of your software engineering skill set.

Send your questions and comments for James to testrun@microsoft.com.

James McCaffrey works for Volt Information Sciences Inc., where he manages technical training for software engineers working at Microsoft. He has worked on several Microsoft products including Internet Explorer and MSN Search. James can be reached at jmccaffrey@volt.com or v-jammc@microsoft.com.


cloudshare
Give Your Developers What They Need.
A Cloud-Based Dev/Test Environment.
Free Trial