14. Quality and Metrics in Requirements Analysis

This chapter describes measures of quality in requirements. Figure 1 reminds the reader of its context. The learning goals of this chapter are shown in Figure 2.

Figure 1

Activity to Which This Chapter Relates

- Planning
- Requirements Analysis
- Emphasis
- Integration and System Testing
- Design
- Implementation and Unit Testing

Figure 2

- Understand the attributes of requirements that lend quality to the application
- Measure each quality attribute

14.1 Introduction to Quality in Requirements Analysis

The more a requirements document expresses what the customer wants and needs, the higher its quality. We usually think of details as being far less important than the “big picture” but a missing requirements detail can seriously impact projects, as numerous case studies show. Recall, for example, the overlooked detail of metric-to-non-metric distance conversion that dispatched a $125 Million spacecraft to oblivion.

To help ensure that the requirements are indeed covered, we focus on qualities that requirements should possess. They should be complete and consistent; each one should be capable of being traced through to the design and the implementation, tested for validity, and implemented according to a rational priority. Figure 3 lists these attributes
and tells what we should look for in good requirements. We can systematically review and inspect requirements based on this list. For example, a set of consistent requirements is far more likely to express what stakeholders want and need than a set with contradictions.

**The Qualities of Requirements Analysis**

<table>
<thead>
<tr>
<th>Quality</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessible</td>
<td>How accessible is each requirement?</td>
</tr>
<tr>
<td>Comprehensive</td>
<td>How comprehensive is the SRS?</td>
</tr>
<tr>
<td>Consistent</td>
<td>How consistent is the SRS?</td>
</tr>
<tr>
<td>Effectively Prioritized</td>
<td>How effectively prioritized are the requirements?</td>
</tr>
<tr>
<td>Secure</td>
<td>How secure is the requirement?</td>
</tr>
<tr>
<td>Self-complete</td>
<td>How self-complete is the SRS?</td>
</tr>
<tr>
<td>Testable</td>
<td>How testable is each requirement?</td>
</tr>
<tr>
<td>Traceable</td>
<td>How traceable is each requirement?</td>
</tr>
<tr>
<td>Understandable</td>
<td>How understandable is each requirement?</td>
</tr>
<tr>
<td>Unambiguous</td>
<td>How unambiguous is each requirement?</td>
</tr>
</tbody>
</table>

This chapter discusses how each of these qualities can be measured.

Projects are greatly improved when the QA organization is involved in the requirements analysis stage. In particular, QA verifies that the intended development process is being executed in accordance with plans. QA should participate in inspections of requirements documents. They tend to have a healthy perspective because they understand that they will have to validate the product based on the requirements. In poorly organized projects, QA may be handed an application with little or no requirements documentation and asked to test it. This begs the question “what is the application supposed to do?”

Security can be a quality of requirements or an actual requirement. For example, “We will make the requirements in our requirements document unlikely to cause security breaches” is not a requirement of the application itself. It is a quality of requirements. On the other hand, “All passwords shall be unavailable except to the system administrator” is an actual requirement of an application.

Agile methods deal with requirements quality through continual contact with the customer: Preferably by having a customer representative working with the team. The desirability of the qualities in Figure 3 is no less important for agile programmers: They can act as a guide and checklist for the interaction with the customer.

Metrics are most useful when their target values are specified *in advance*. For example, we could state that, based on experience, requirements will be considered “complete” when the rate of modification and addition is less than 1% per week.
14.2 Accessibility of Requirements

A project’s requirements change continually throughout its lifecycle. For example, when a programmer tries to implement a requirement and shows it to the customer, the latter frequently finds missing parts. The SRS must then be accessed to ascertain whether these missing requirements were present, and included if they were not. Taking an example from the Video Store case study, the customer (the video store) may question why a DVD’s play time does not appear on the monitor. The developers and the customer will want to know whether this was specified in the SRS. Where in that document should they look? Rummaging through poorly organized documents is time-consuming and therefore expensive.

One accessibility metric is the *average time taken to access a detailed requirement*. To measure this, a sample would be taken of existing and missing requirements, and several people would be timed finding these or ascertaining that they are absent. As outlined in section xx of chapter xx, 150 would be a good sample size. Smaller sample sizes are unreliable but are probably better than nothing. This is summarized in Figure 4.

![Accessibility](image)

- **Ease of getting to statement** of detailed requirements.
- **Metric:**
  - 0: extremely long average access time (compared with organization’s norm)
  - 10: average access time as fast as can be expected

Figure 4

14.3 Comprehensiveness of Requirements

A quality SRS expresses all of the requirements for a product. By *comprehensiveness*, we mean the extent to which the customer’s wants and needs are included. An appropriate metric would thus be *percentage of the customer’s requirements appearing in the SRS*. An obvious way to ensure this is to have the customer validate it, but this is not a simple matter, as Figure 5 suggests.
Issues in Attaining Comprehensive Requirements

- Not enough resources to satisfy every customer wish
  - Prioritize so that comprehensive within each batch of requirements
- Customer can’t / won’t read entire SRS
  - Make SRS easy to follow
  - Use a standard
  - “Read” SRS to customer
- Limitations of self-inspections
  - Subject to peer inspection
- Contradictory stakeholder requirements need to be satisfied
  - Apply diplomatic skills and expect compromise

Figure 5

“Complete” requirements form an elusive and vague goal; and yet the completeness of requirements is key to the successful completion of a project and the tracking of progress. Each iteration makes the requirements more comprehensive. One way to deal with the evolving set of requirements is to include requirements of future iterations and of all priorities in measuring completeness. An example is shown in Table 1. This perspective helps us to assess how close our plans are to satisfying the customer’s wants and needs.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>#</th>
<th>description</th>
<th>Priority</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>780</td>
<td>780</td>
<td>Every DVD shall …</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>781</td>
<td>781</td>
<td>Customer records shall include …</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>782</td>
<td>782</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1

Typically, everyone wants to know the fraction of the requirements gathered this far. For example, suppose that we have a document with 1,000 detailed requirements. Are we halfway to capturing all of the requirements? Almost done? How do we assess this? One way is to perform random sampling from the customer. Ask various representatives of the customer to provide 150 requirements. Determine the fraction of these that are already present in the SRS. This fractional estimate can be considered a reasonable measure of completion. Although helpful, this is probably and under-estimate, however, because of people’s inability to envision all of what they want and need.

The IEEE defines a measure of completeness in 982.2-1988 A35.1. This is a formula involving 18 observed quantities (e.g., “number of condition options without
processing”) and 10 weights (e.g., the relative importance of “defined functions used”). It measures the degree to which there are loose ends within a set of D-requirements.

### 14.4 Consistency of Requirements

A set of D-requirements is *consistent* if there are no contradictions among them. As the number of D-requirements grows, inconsistency tends to become difficult to detect. This is illustrated by the three inconsistent requirements in Figure 6.

**Example of Inconsistency**

**Requirement 14.** Only basic food staples shall be carried by game characters

......

**Requirement 223.** Every game character shall carry water.

......

**Requirement 497.** Flour, butter, milk and salt shall be considered the only basic food staples.

Figure 6

The object-oriented organization of requirements helps to avoid inconsistencies by classifying D-requirements by class, and by decomposing them into a simple form. This is not a guarantee of consistency, however, and so requirements inspections check for consistency along with the other qualities mentioned.

The following is a consistency metric: *The percentage of detailed requirements partly or wholly contradicted elsewhere.* One would consider a sample of detailed requirements – 150 would be ideal – and investigate each one in turn to determine whether it is contradicted elsewhere in the document. This entails comparing it to all of the remaining detailed requirements. Figure 7 provides another example.
Figure 7

14.5 Prioritization of Requirements

Since quality is ultimately defined by customer satisfaction, the requirements analysis process is continually directed towards the customer’s concept of satisfaction. Teams typically show stakeholders interim accomplishments and stakeholders then influence the course of the work accordingly. Because of this, the priority of requirements – and thus the order in which requirements are implemented, as described in chapter xx – makes a big difference in the customer’s satisfaction. In mathematical language, this is a non-commutative operation since the SRS sequence

\[
\text{implement requirement A then plan to implement requirement B}
\]

-- may well produce a different product from

\[
\text{implement requirement B then plan to implement requirement A.}
\]

There are good and bad prioritizations. For example, giving every requirement the highest priority indicates poor planning. Ranking low-priority requirements is usually a waste of time because they are unlikely to be all implemented: As mentioned, when stakeholders see the implementation of requirements, they tend to change subsequent requirements.

Assume that each requirement is in one of three priorities. How would we measure the quality of the prioritization? A good quality prioritization categorizes requirements into equal parts, indicating that no category has been neglected. Figure 8 shows a metric for this.
A Prioritization Metric

Assume three prioritizations: high, medium, and low

Metric: Variation from \( \# \text{high} = \# \text{medium} = \# \text{low} \)

Let \( T = \) total number of detailed requirements

\[
\frac{100 \times |T - \frac{T}{3} - \# \text{high}| - |\frac{T}{3} - \# \text{medium}| - |\frac{T}{3} - \# \text{low}|}{T}
\]

0 = worst, 100 = best

Figure 8

For example, if 900 requirements are very well prioritized, each category would contain 300 requirements, and the formula would give \( 100 \times \frac{900 - 0 - 0}{900} = 100\% \). On the other hand, if 700 were classified as high priority, 100 as medium and 100 as low, the metric would yield

\[
100 \times \frac{900 - |300 - 700| - |300 - 100| - |300 - 100|}{900} = 100 \times \frac{100}{900} = 11.1\%.
\]

This indicates poor prioritization.

14.6 Security and High-level Requirements

Non-security requirements specify functionality (e.g., “The application shall allow the retrieval of customer records”) or else they qualify functionality (e.g., “Average retrieval time shall be less than half a second.”). Security requirements are a special case for the following reason.

14.7 Self-Completeness of Requirements

Typically, a requirement depends on other requirements. A set of requirements is self-complete if it contains every part whose inclusion is necessitated by parts already present. Figure 9 illustrates an incomplete set of requirements.
**Self-Completeness:** ("nothing else needed")

**REQUIREMENTS**

1. *The application shall display a DVD in stock when a title is entered at the prompt; otherwise it shall display ‘OUT OF STOCK’.*

2. *The application shall display all of the store’s DVD’s by any director whose last name is entered at the prompt. These shall be displayed one by one. Advancing through the DVD’s shall be controlled by the forward arrow key.*

*Incomplete:* Lacks specification on how to display a video!

**Figure 9**

Without the specification of how a video is to be “displayed,” this set of requirements is incomplete as a unit.

As another example, suppose that the SRS for a calendar application contains the following requirement.

*The application shall retain all information entered by the user for each appointment.*

The presence of this requirement necessitates a requirement describing means for entering appointment information. It also necessitates a requirement explaining means for displaying this information. This is the meaning of “self-completion.” To measure it, we look at each detailed requirement and note any necessary associated requirement that is missing. An appropriate metric is shown in Figure 10.

**Self-Completeness**

When a requirement is present, so must all those necessitated by its presence.

A metric (0 = best; 1 poor; no theoretical upper limit):

\[
\frac{\text{number of missing necessary associated requirements}}{\text{number of detailed requirements present}}
\]
Figure 10

The number of missing requirements is determined by sampling.

14.8 Testability of Requirements

Each requirements must be *testable*: i.e., it must be possible to validate every detailed requirement by testing that it has been properly implemented. Figure 11 provides an example of a non-testable requirement, and shows what it would take to make the requirement testable.

```
The system shall display the difference in salary between the client and the world wide average for the same trade

X -- can't be tested because the average mentioned cannot be determined (event though it exists).

Better:

The system shall display the difference in salary between the client and the estimated world-wide average for the same trade as published by the United Nations on its website www.tbd at the time of the display....
```

Figure 11

Requirements that are not testable are of negligible value: It would be impossible to assess whether such a requirement has been attained. This is an all-or-nothing property: There is little value in the “degree of testability” of a requirement.

Testability can sometimes be used to specify a detailed GUI requirement. For example, instead of specifying a GUI in complete detail we may prefer to provide a test such as the following.

```
The GUI for entering DVD's shall have the fields and buttons listed in figure xx, and its design shall score an average of at least 8.5 out of ten on the user satisfaction questionnaire in Table 2.
```

<table>
<thead>
<tr>
<th>User Satisfaction</th>
<th>Quality</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 = of no value; 5 = average satisfaction; 10 = a pleasure to use</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2004 by E. Braude   Quality in Requirements Analysis   Page 9   1/19/2016
Table 2

An effective way to ensure testability and to ensure the clarity of a requirement at the same time is to include tests with the requirement. Figure 12 and Figure 13 illustrate an example.

**Including Tests with Requirements: Example Requirement**

The video store application shall implement a discount program as follows.

- One DVD: no discount
- Two DVD’s: 20% discount for the second DVD
- Three DVD’s: 40% discount for the second and third DVD
- All video’s beyond the third are discounted at 40%.

**Figure 12**
Including Tests with Requirements Accompanying Tests

1. Customer Owen Jones rents one DVD. He is charged the regular rental of $5.00.

2. Customer Teresa Edwards rents two DVD’s. She is charged the regular rental of $5.00 for the first and $4.00 (20% discount) for the second.

3. Customer Theodore List rents three DVD’s. He is charged the regular rental of $5.00 for the first, $4.00 (20% discount) for the second, and $3.00 (40% discount) for the third.

4. Customer Fred Harari rents five DVD’s. He is charged the regular rental of $5.00 for the first, $4.00 (20% discount) for the second, and $3.00 (40% discount) for the third. The fourth and fifth DVD’s are charged at $3.00 (40% discount).

Figure 13

Agile programming applies this test orientation but short-circuit’s the prose by encoding the test up front, using it as a requirement, in effect. A reasonable testability metric is the following.

The percentage of detailed requirements that are accompanied by a test

Even when a requirement is testable, executing a test may be difficult or time-consuming. To be complete, we sometimes need to consider the cost of testing as part of the consideration. A very high cost influences our choice and priority of requirements. For example, suppose that we consider adding to our online video service the following requirement.

For each movie entered by the customer, the application shall display a number from 0 to 5 that provides the system’s estimate of how much he will like the movie. This estimate is based on the customer’s past viewing ratings.

This can be tested, but the expense of doing so properly may dissuade us from making it a high priority.

14.9 Trace-ability of Requirements

Trace-ability was defined in section xx of chapter xx. Our concern here is how to measure a set of requirements in this respect. A requirement is traceable if it maps in a completely clear manner to the design element that accommodates it; the code that implements it; and the unit tests that test it. When the OO (or “class”) organization of requirement writing is used, the mapping to the class within the design and to the code can be completely clear. It requires work and clear thinking to maintain such clarity, however. For example, a Customer paragraph that specifies the rentals that each customer can have could compromise trace-ability: The design would probably have the
classes Customer, DVD, DVDRental, and DVDRentals. The requirements organization should be reflect this.

Organizing classes by functionality, GUIs, use cases, etc. has various advantages, as discussed in section xx of chapter xx; but trace-ability is not a strength for most of them. One would have to peruse each detailed requirement and ask whether it will clearly map to a class. Trace-ability can be measured as in Figure 14. Using 2 as a measure allows the application of triage for a give requirement, using $0 = \text{untraceable}; 2 = \text{fully traceable}; 1$ otherwise.

**A Trace-ability Metric**

(Range: 0 to 100)

\[
100 \times \sum [\text{trace-ability of each detailed requirement (0-2)}] \\
\text{2 \times [number of detailed requirements]} \\
\]

0 = un-trace-able; 2 = clearly trace-able to a specific class

**Figure 14**

### 14.10 Understandability of Requirements

Understandability seems to be a highly subjective quality because it depends on peoples’ opinion: However, understandability can be measured. For example, a set of people can be asked to express on a form their opinion of a requirements document. Table 2 is an example of an opinion form – in this case applied to a user interface

### 14.11 Un-ambiguity of Requirements

Unless a D-requirement is written clearly and unambiguously, we won’t be able to determine whether it has been properly implemented. Figure 15 illustrates an example of an ambiguous requirement, followed by an improved version.
The player can decide the qualities of Encounter characters.

\[ \times \text{ At any time? Probably not. Would have to test under all circumstances, many not intended, incurring unnecessary expense, and producing a wrong result.} \]

\[ \checkmark \text{ Better version:} \]

Whenever all foreign players are absent from the area containing the player’s main character, the player may change the quality values of this character, keeping the sum total of the quality values unchanged. The PlayerQualityWindow, (see section tbd) is used for this purpose. Changes take effect four seconds after the “OK” button is pressed.

\[ \text{Figure 15} \]

Figure 16 shows a metric for ambiguity that depends on a triage measurement: Decide whether the detailed requirement has exactly one clear meaning (score of 2); otherwise many meanings (score 0); otherwise give it a score of one.

**Un-ambiguity**

A metric (range: 0 to 100)

\[
100 \times \frac{\sum [\text{un-ambiguity of each detailed requirement (0-2)}]}{2 \times [\text{number of detailed requirements}]}
\]

0 = could have many meanings; 2 = clearly one meaning

\[ \text{Figure 16} \]

### 14.12 Measuring Completion

Measuring the extent to which requirements have been implemented is not complicated. Figure 17 shows two useful completion metrics.
Comprehensiveness Metrics

Let T = total number of documented detailed requirements (all priorities; all iterations)

\[
\text{METRIC: } \% \text{ Requirements implemented} = \frac{100 \times \text{# requirements implemented}}{T}
\]

\[
\text{METRIC: } \% \text{ Requirements Currently Targeted} = \frac{100 \times \text{(# requirements implemented)}}{\text{T}} + \frac{\text{(# top priority requirements in current iteration)}}{\text{T}}
\]

Figure 17

14.13 Alternative and Additional Metrics for Requirements Analysis

The following list of quality assurance metrics includes requirements analysis metrics in IEEE Standard 982.2-1988 (“IEEE Guide for The Use of IEEE Standard Dictionary of Measures to Produce Reliable Software”). Some measure the qualities we have already discussed in a different manner.

- Percentage of unambiguous specific requirements (IEEE metric 6)
- Degree of completeness (IEEE metrics 23 and 35)
- Percentage of misclassified D-requirements (in the object-oriented style, this measures the percentage allocated to the wrong class)
- Trace-ability (IEEE metric 7)
- Degree of atomicity (indivisible into smaller parts)
- Consistent with the remaining requirements (IEEE metrics 12 and 23)
- Measures of the effectiveness of requirements inspection
- Percentage of missing or defective requirements found per hour of inspection
- Measures of the effectiveness of the requirements analysis process
- Cost per D-requirement
  - On a gross basis (total time spent / number of D-requirements)
  - On a marginal basis (cost to get one more)
- Rate at which specific requirements are …
  - modified
  - eliminated
  - added
- Measure of the degree of completeness of the requirements. This can be estimated from the rate, after the official end of D-requirements collection, at which specific requirements are …
14.14 Inspecting D-Requirements

The reader is referred to Chapter xx for a description of the inspection process in general. Specific requirements (or D-requirements) are the first software process documents which can be inspected against prior documentation (the C-requirements). Inspectors prepare for the inspection by reading over the C-requirements and comparing the specific requirements with them. It can be very productive to inspect requirements against each of the qualities and metrics listed above.

15.14.1 Example of Un-inspected D-Requirements

This section provides as example of a D-Requirements inspection. Here is an un-inspected version of D-requirements upon which we will perform an example inspection, entering the results in a table (see Table 3). We employ the technique of automatically adding the [not inspected yet] comment to each. It is removed when the inspection takes place. The final version of these requirements, resulting from the inspection, is shown in the case study on page xx.

Area Requirement 1 (“Area name”). [Not inspected yet] Every area shall have a name of up to 15 characters.
Area Requirement 2 (“Area image”). [Not inspected yet] There shall be an image in gif form to display each Area object.
Area Requirement 3 (“Display area method”). [Not inspected yet] Whenever a player character enters an area, that area and the characters in it shall be displayed.
Area Requirement 4 (“Courtyard object”). [Not inspected yet] There shall be an Area object with name “courtyard.” Its image shall be that shown in Figure xx on page xx.
Area Requirement 5 (“Dressing room object”). [Not inspected yet] There shall be an Area object with name “dressing room” and blank background image. The dressing room shall be adjacent to the courtyard area.

Encounter Requirement 1 (“Engaging a foreign character”). [Not inspected yet] When an engagement takes place, the following computation is performed: The sum of the values of qualities of a game character relevant to the area in question shall be referred to as the character’s area value. [In this release, all qualities will count as equal.] In an engagement, the system compares the area values of the characters and transfers to the stronger, half of the points of the weaker. For example, suppose the player engages a foreign character in an area requiring stamina and attention span, and ps is the value of the player’s stamina, etc. Assuming ps + pa > fs + fa, we would have ps’ = ps + fs/2, pa’ = pa + fa/2, fs’ = fs/2, fa’ = fs/2 where x’ is the value of x after the transaction.
EncounterCharacter Requirement 1 (“Name of game character”). [Not inspected yet] Every game character in the Encounter video game shall have a unique name of up to 15 characters.

EncounterCharacter Requirement 2 (“Qualities of game characters”). [Not inspected yet] Every game character has the same set of qualities, each having a floating point value. These are initialized to 100/n, where n is the number of qualities. The qualities are attention span, endurance, intelligence, patience, and strength.

EncounterCharacter Requirement 3 (“Image of game character”). [Not inspected yet] Every game character will be shown using an image that takes up no more than 1/8 of the monitor screen.

EncounterCharacter Requirement 4 (“Engagement with foreign character”). [Not inspected yet] Whenever an Encounter game character enters an area containing another game character, and one of them is player-controlled, the player character may either choose or be obliged by the game to engage the other character. Whether there is a choice or not is controlled by the game in a random way on a 50% basis.

EncounterGame Requirement 1 (“Encounter game object”). [Not inspected yet] There shall be a single EncounterGame object.

ForeignCharacter Requirement 1 (“Freddie foreign character object”). [Not inspected yet] There shall be a foreign character named “Freddie,” all of whose qualities have equal values and whose image is shown in Figure 4.57 on page 242.

PlayerCharacter Requirement 1 (“Configurability”). [Not inspected yet] Whenever all foreign players are absent from an area, the player may set the values of his qualities using the PlayerQualityWindow, as long as the sum of the quality values remains the same.

PlayerCharacter Requirement 2 (“Main player character”). [Not inspected yet] The player shall have complete control over a particular game character called the main character.

PlayerCharacter Requirement 3 (“Living points”) [Not inspected yet]. Encounter shall produce the sum of the values of the character’s qualities, called its living points.

We will show typical results of an inspection of these requirements. One inspection comment about this set as a whole is that the requirements do not support enough expansion of the game into a competitive product. A more particular defect is that the requirements do not properly specify the delay involved in setting a player’s quality values; during the delay the player is subjected to an engagement in an unprepared state. (If the delay is too small, the player simply sets the qualities required for the area as high as possible and the game is not much of a challenge.) Let’s inspect the list of proposed specific requirements one at a time.

Table 3 is an example of a form that can be used for the inspection of D-requirements, applied to the above list. The properties in Table 4.3 are defined above in Section 3 on page 187 (adapted from Ross [Ro]). Most of the metrics described in that section can be computed from this table.
The table contains “Notes” and “No” notes.

Here are the “No” notes.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Traceable forward</th>
<th>Complete</th>
<th>Consistent</th>
<th>Feasible</th>
<th>Non-ambiguous</th>
<th>Clear</th>
<th>Precise</th>
<th>Modifiable</th>
<th>Testable</th>
<th>Traceable backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area Requirement 1</td>
<td>Note 2</td>
<td>Yes</td>
<td>Yes</td>
<td>No 1</td>
<td>Yes</td>
<td>No 2</td>
<td>No 1, 2</td>
<td>Yes</td>
<td>Note 14</td>
<td></td>
</tr>
<tr>
<td>Area Requirement 2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No 3</td>
<td>Yes</td>
<td>No 3</td>
<td>Note 3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Area Requirement 6</td>
<td>Yes</td>
<td>Note 5</td>
<td>Note 5</td>
<td>Yes</td>
<td>No 3</td>
<td>No 3</td>
<td>No 5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Area Requirement 3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Note 6</td>
<td>Note 3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Area Requirement 4</td>
<td>Yes</td>
<td>Note 7</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Engagement Requirement 1</td>
<td>Note 2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Note 3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EncounterCharacter Requirement 1</td>
<td>Yes</td>
<td>Note 1</td>
<td>Yes</td>
<td>Yes</td>
<td>No 1</td>
<td>Yes</td>
<td>No 1</td>
<td>No 2</td>
<td>No 1, 2</td>
<td>Yes</td>
</tr>
<tr>
<td>EncounterCharacter Requirement 2</td>
<td>Yes</td>
<td>No 11</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Note 8</td>
<td>Yes</td>
<td>No 6</td>
<td>Yes</td>
<td>Note 9</td>
</tr>
<tr>
<td>EncounterCharacter Requirement 3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Note 3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EncounterCharacter Requirement 4</td>
<td>Note 11</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Note 12</td>
<td>Note 12</td>
<td>No 7</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EncounterGame Requirement 1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Note 13</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Foreign Character Requirement 1</td>
<td>Note 11</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No 8, 9</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Player Character Requirement 1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No 10</td>
<td>Yes</td>
<td>Note 15</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Player Character Requirement 2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No 12</td>
<td>No 12</td>
<td>No 3</td>
<td>No 12</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Player Character Requirement 3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Note 10</td>
<td>Note 10</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3

The table contains “Notes” and “No” notes.

Here are the “No” notes.
1. Can a game character or area have a name with no characters?
2. The number 15 is rigid.
3. Only one?
4. If the player controls several characters, are all of their areas to be displayed or does this have to do only with the main player character?
5. Filling the entire monitor screen?
6. It should be easier to add new qualities or remove them.
7. When is there a Freddie? When does he appear?
8. In future releases characters may mutate.
9. Clarify what stays the same.
10. Can the value of a quality be negative?
11. Ambiguous because the player can’t control everything that happens to the main character at all times.
12. Refine “complete control.”

The “Notes” are as follows:

1. Is any keyboard character acceptable?
2. Check validity with customer.
3. It is unclear how modifiable this should be.
4. It is hard to answer “complete” because it is unclear. See the note referenced in the clear column for the issue.
5. We assume that the customer has some leeway in exactly what “courtyard” will look like.
6. Are there dressing room exits to any other area?
7. This is somewhat clumsily written: could lead to misunderstanding.
8. It is usually preferable to have a single requirement match each attribute.
This does not appear necessary, as the qualities will be treated alike.
9. Produce at any time? On request? Show at all times?
10. These details are not mentioned in the C-requirements: check with customer.
11. Clarify “50% basis,” if possible.
12. For Internet versions it may become necessary to have more than one instance of an EncounterGame object. We will not exclude this possibility in future iterations.
13. It is not clear in what directions this could be modified.
14. Is the requirement written in such a way that it will be possible to trace it through to the code that implements it?

### 14.15 Summary of Quality in Requirements

Figure 18 and Figure 19 summarize the process that can be followed in expressing a single requirement.
Writing a Detailed Requirement 1

1. Classify requirement as functional or non-functional
   - IEEE SRS prompts for most non-functional
   - select method for organizing functional requirements
2. Size carefully
   - a functional requirement corresponds ± to a method
   - too large: hard to manage
   - too small: not worth tracking separately
3. Make trace-able if possible
   - ensure suitable for tracking through design and implementation
4. Make testable
   - sketch a specific test that establishes satisfaction

Figure 18

Writing a Detailed Requirement 2

5. Ensure not ambiguous
   - ensure hard to misunderstand intention
6. Give the requirement a priority
   - e.g., highest (“essential”); lowest (“optional”); neither (“desirable”)
7. Check that requirement set self-complete
   - for each requirement, ensure all other necessary accompanying requirements are also present
8. Include all error conditions
   - state what’s specifically required for non-nominal situations
   - include programmer errors for critical places
9. Check for consistency
   - ensure that each requirement does not contradict any aspect of any other requirement

Figure 19

Most of the steps outlined in Figure 18 and Figure 19 were described in this section as desirable qualities for requirements. The following are additional notes keyed to the numbering in Figure 18 and Figure 19.

Additional to Note 3: Assessing whether or not a requirement is traceable amounts to imagining a design for the application and imagining how the requirement would have to be satisfied by the design. This is easiest if the requirement corresponds directly to a method.

Additional to Note 8: Many requirements depend on particular data and we need to indicate how the requirement is to operate in case the data is wrong or inconsistent. For critical requirements, this should include errors due to bad design or programming (routine requirements may not account for defective design or defective programming of the application). For example:
An acceptable requirement:
   When the on button is pressed, the high-intensity X-ray shall turn on if the parameters satisfy the following conditions …
The application itself should not permit illegal parameters – specified elsewhere – but we specify this redundantly because the operation is a critical one.

An unacceptable requirement:
   The tic-tac-toe positions shall be displayed if no player has had three moves more than the other player.
The application itself should not permit illegal plays, and redundancy here is overkill.

Figure 20 summaries the main points made in this chapter. A set of requirements satisfying this list is more likely to express the wants and needs of the stakeholders compared with a set of requirements deficient in any one of them.

<table>
<thead>
<tr>
<th>Chapter Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following lend quality to requirements</td>
</tr>
<tr>
<td>Accessibility</td>
</tr>
<tr>
<td>Comprehensiveness</td>
</tr>
<tr>
<td>Consistency</td>
</tr>
<tr>
<td>Prioritization</td>
</tr>
<tr>
<td>Metrics are most useful when targets are specified in advance</td>
</tr>
<tr>
<td>Use historical data to set goals</td>
</tr>
</tbody>
</table>

**Figure 20**