Using Guidelines to Improve Quality in Software Nonfunctional Attributes

Malik Hneif and Sai Peck Lee, University of Malaya

// A step-by-step approach to ensuring software quality uses prioritized design guidelines to manage nonfunctional attributes and their interrelationships. //

SOFTWARE ENGINEERING INITIALLY focused on fulfilling a system's functional requirements. As attention to industrial and business considerations increased, the focus gradually shifted toward achieving quality requirements as well.[1] These requirements reflect a system's total behavior and capability not only to fulfill its functionalities but to do so effectively, safely, and productively while also giving users a certain level of satisfaction.[2,3] Software quality has therefore come to encompass system aspects ranging from business considerations such as integration with legacy systems, time to market, and projected lifetime to nonfunctional attributes (NFAs), which include security, availability, performance, usability, and maintainability.[4]

The quality level defines how to implement a system.[5] For example, manufacturing a basic car requires fulfilling functional requirements such as forward and backward propulsion at different speeds. We can design and construct a car simply to achieve its functional requirements, but this wouldn't guarantee the quality that manifests in features such as anti-theft security systems, fuel efficiency, and interior comfort in hot and cold weather. Satisfying such nonfunctional requirements affect a car's design and construction to produce a higher-quality car.

The same applies to software development. There are many ways to develop a software system, but the required quality level restricts the ways of implementing it. We've developed an approach for improving NFA quality by identifying guidelines to help software engineers better meet nonfunctional requirements during system design, implementation, and deployment. (See the sidebar for related work.)

A Guideline-Based Approach

Our approach to achieving NFA quality is preventive, as opposed to curative—that is, it focuses on preventing defects associated with NFAs during the software development life cycle, rather than identifying and correcting defects after testing.[6] The approach is heuristic, based on a framework for nonfunctional requirements defined by Bill Andreopoulos.[5,7] Its practical implementation is through an optimal set of prioritized guidelines that software engineers can identify and apply efficiently throughout system development.

0740-7459/11/$26.00 © 2011 IEEE

RELATED WORK IN NONFUNCTIONAL ATTRIBUTE QUALITY

The research literature includes several approaches to ensure nonfunctional attribute (NFA) quality in software systems.[1–5] However, most approaches focus on individual NFAs, whereas systems usually involve multiple NFAs. Bill Andreopoulos introduced a methodology for evaluating and improving software quality by defining soft goals in a nonfunctional requirements framework.[6] Our work is based on Andreopoulos's heuristic methodology by rejecting candidate guidelines that negatively affect higher-priority nonfunctional quality requirements.

David Budgen categorized software engineering guidelines into three main categories, which we use in our approach: architectural styles, design patterns, and best practices.[7] An architectural style can have positive and negative effects on one or more NFAs. For example, a layered-systems style increases modularity, which improves a system's maintainability, but it negatively affects performance by slowing the data and control flow.[8] The same applies to other guideline categories: secure pipe and message interceptor are design patterns that improve security;[3] an undo feature is a best practice that improves the usability.

A software system's quality is specified in the requirements phase and implemented in the design phase. However, the implementation and deployment phases must preserve the system's quality, so the effort to achieve and preserve NFA quality should span the design, implementation, and deployment phases.[1,9]

Software system stakeholders usually aim for a satisfactory NFA quality level. However, they sometimes require each NFA to achieve a specific quality level. This implies the need to quantify the required quality. Tom Glib and and Lindsey Brodie established a way of doing this using Planguage, a keyword-driven language for writing measurable quality requirements.[10]

References

1. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley Professional, 2003.
2. B. Andreopoulos, "Satisficing the Conflicting Software Qualities of Maintainability and Performance at the Source Code Level," *Proc. Workshop em Engenharia de Requisitos* (WER 04), 2004; http://wer.inf.puc-rio.br/WERpapers/artigos/artigos_WER04/Bill_Andreopoulos.pdf.
3. D.G. Rosado et al., "Security Patterns and Requirements for Internet-Based Applications," *Internet Research*, vol. 16, no. 5, 2006, pp. 519–536.
4. S. Bode et al., "Software Architectural Design Meets Security Engineering," *Proc. 16th Ann. IEEE Int'l Conf. and Workshop on Eng. Computer Based Systems* (ECBS 09), IEEE CS Press, 2009, pp. 109–118.
5. N. Yoshioka, H. Washizaki, and K. Maruyama, "A Survey on Security Patterns," *Progress in Informatics*, vol. 5, no. 5, 2008, pp. 35–47.
6. B. Andreopoulos, "Achieving Software Quality Using the NFR Framework: Maintainability and Performance," *Proc. 3rd Int'l Conf. Computer Science, Software Eng., Information Technology, e-Business, and Applications* (CSITeA 04), 2004; www.cse.yorku.ca/~billa/MULIC/AndreopoulosCSITeA.pdf.
7. D. Budgen, *Software Design*, 2nd ed., Addison-Wesley, 2003.
8. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
9. B. Meyer, "Practice to Perfect: The Quality First Model," *IEEE Software*, vol. 30, 1997, pp. 102–103.
10. T. Gilb and L. Brodie, *Competitive Engineering*, Butterworth-Heinemann, 2005.

Selecting Guidelines

Two properties characterize an optimal guideline set. First, the selected guidelines should have positive effects on the required NFAs. Second, they should be homogeneous—that is, they should have no negative effects on each other.[8] Three main factors affect the selection of an optimal guideline set.

NFA priorities. Different software systems can have different NFA priorities. For example, a banking system prioritizes security first, followed by availability and performance, whereas statistical analysis systems give priority to reliability, followed by performance and usability.

Prioritizing NFAs helps choose guidelines that improve the quality of higher-priority attributes and reject guidelines that affect those attributes negatively.

Guideline effects on NFAs. NFAs are interrelated, so achieving one attribute can positively or negatively affect the quality of others. In our approach, we categorize the effects of guidelines on NFAs as positive, negative, or neutral. For example, the secure-pipe pattern has a positive effect on security, whereas the layered-architecture style affects system performance negatively, and the proxy design pattern has no effect on usability.

Guideline interrelationships. Because guidelines have different effects on each other, our approach identifies the relationship between them. There are four possibilities:

- *Complementary.* One guideline positively affects another, recommending their use together. For

TABLE 1 Effects of guidelines on nonfunctional software attributes.

Guideline	Guideline type	Nonfunctional attributes			
		Availability	Security	Performance	…
G1: Layered security	Architectural style	Positive	Positive	Negative	…
G2: Packet filter firewall	Design pattern	Neutral	Positive	Neutral	…
G3: Single access point	Design pattern	Negative	Positive	Positive	…
G4: Active redundancy	Best practice	Positive	Neutral	Neutral	…
G5: Checkpoint/rollback	Best practice	Positive	Neutral	Neutral	…
G6: Dynamic scheduling	Best practice	Neutral	Neutral	Positive	…
G7: Lazy initialization	Design pattern	Negative	Neutral	Positive	…
…	…	…	…	…	…

TABLE 2 Relationship matrix between example guidelines.

	Guidelines							
	G1: Layered security	G2: Packet filter firewall	G3: Single access point	G4: Active redundancy	G5: Check-point/rollback	G6: Dynamic scheduling	G7: Lazy initialization	…
G1								
G2	Overlaps							
G3	Complements	Overlaps						
G4	Independent	Independent	Conflicts					
G5	Independent	Independent	Independent	Complements				
G6	Independent	Complements	Independent	Independent	Independent			
G7	Independent	Independent	Independent	Independent	Independent	Independent		
…	…	…	…	…	…	…	…	…

example, with the service-locator and singleton design patterns, the service-locator pattern uses the singleton pattern.

- *Overlapping.* One guideline introduces the same effect as another, so they can replace each other. For example, the dependency-injection and service-locator design patterns overlap. They each solve the same design problem—a class that has dependencies on multiple services—although in different ways.
- *Conflicting.* One guideline negatively affects another one. For example, the single-access-point design pattern and the provide multiple-access guideline introduce opposite effects.
- *Independent.* One guideline has no effect on another, as with implementing the undo feature and using the façade design pattern.

Consideration of these relationships while selecting guidelines ensures a final set that has no overlapping or conflicting relationships.

Using Guidelines to Prevent NFA Defects

The many factors affecting guideline selection complicate the formulation of a homogeneous guideline set. Our approach addresses these complications in two stages.

Preparation stage. During this stage, the software engineer collects and categorizes guidelines and then defines their effects on the NFAs and relationships with each other.

Guideline sources include journal articles, conference papers, books, software company brochures, and websites. Because the number of guidelines can be huge, we categorize them according to their type and the development phase to which they apply. We use three category types: architectural styles, design patterns, and best practices.[9] An architectural style applies early in the development life cycle, as do design patterns. On the other hand, the industry has developed many best practices that are useful throughout the development.

Next, we consider the effects each guideline has on NFAs. Table 1 categorizes the effects of seven guidelines on three NFAs in terms of positive, negative, or neutral.

Finally, we specify the relationships between every two guidelines as complementary, contradictory, overlapping, or nonexisting. Table 2 illustrates the output of this step for the seven guidelines listed in Table 1.

Application stage. With a complete dataset of guidelines, effects, and relationships, software engineers can apply our approach to new system development projects. The approach's fundamental idea of pairwise comparisons is similar to Tom Glib and Lindsey Brodie's impact estimation.[10]

To obtain a homogeneous guideline set, the software engineer must apply the following steps to each prioritized NFA:

1. Retrieve the guidelines that positively affect that NFA.

```
function get_guidelines (prioritized_NFA_list) → set of guidelines
  declare guidelines_result_set : set of guidelines
  // include the guidelines for each attribute, and accumulate them in guidelines_result_set.
  For each attribute in prioritized_NFA_list
    Declare attribute_guidelines : set of guidelines
    attribute_guidelines = Get_guidelines_with_positive_relation (attribute)
    while priority (attribute) = priority (attribute + 1) // for multiple attributes with the same priority
      // include the guidelines with positive effects on all the attributes with the same priority
      Append (attribute_guidelines, Get_guidelines_with_positive_relation (attribute+1))
      attribute = attribute + 1
    end while

    // remove duplicates between guidelines fetched previously,
             and the guidelines for the current attribute.
    // for the first attribute, this loop is not executed, since guidelines_result_set is still empty.
    for each g1 in attribute_guidelines
      for each g2 in guidelines_result_set
        if g1 = g2
          Remove (attribute_guidelines, g1)
        end if
      end for
    end for

    // remove guidelines that have negative effect on higher-priority attributes.
    for each guideline in attribute_guidelines
      for each other_attribute in prioritized_NFA_list with higher priority than attribute
        if Relation (guideline, other_attribute) = "negative"
          Remove (attribute_guidelines, guideline)
        end if
      end for
    end for

    // remove conflicting/overlapping guidelines with the previously fetched guidelines.
    // for the first attribute, this loop is not executed, since guidelines_result_set is still empty.
    for each g1 in attribute_guidelines
      for each g2 in guidelines_result_set
        if (Relation (g1, g2) = "conflicts") or (Relation (g1, g2) = "overlaps")
          Remove (attribute_guidelines, g1)
        end if
      end for
    end for

  // resolve overlapping/conflicting guidelines.
  for each g1 in attribute_guidelines
    for each g2 in attribute_guidelines
      if Relation ( g1 , g2) = "overlaps" or Relation ( g1 , g2) = "conflicts"
        declare user_choice from input dialog
        if user_choice = g1
          Remove (attribute_guidelines, g2)
        else
          Remove (attribute_guidelines, g1)
        end if
      end if
    end for
  end for
    // add the new set of guidelines to the result set of guidelines
    Append (guidelines_result_set, attribute_guidelines)
  end for
  return guidelines_result_set
end function
```

FIGURE 1. Pseudocode implementing the algorithm for establishing an optimal guideline set for achieving nonfunctional attributes. Applying this algorithm in a software tool helps in using this approach for building high-quality software systems.

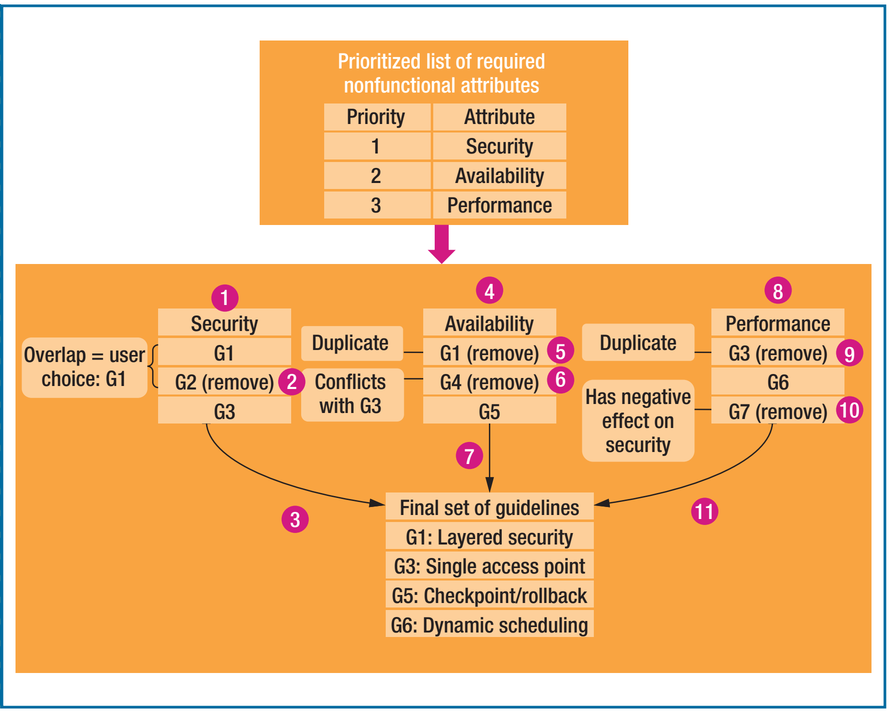

FIGURE 2. Example application of the approach to a banking software system. The numbers show the sequence of pairwise comparisons to identify a homogeneous guideline set for implementing NFA quality requirements.

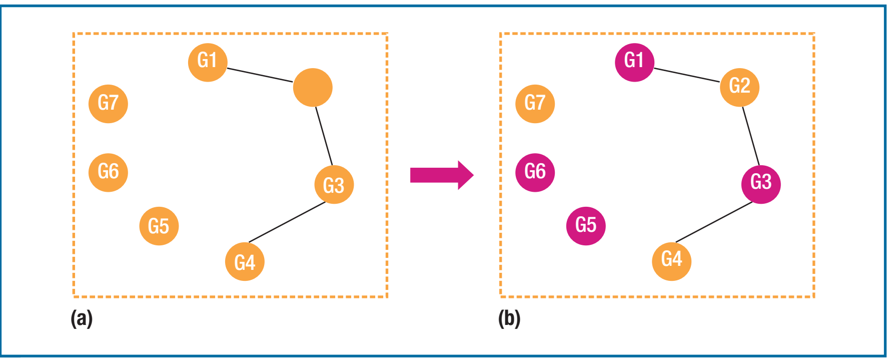

FIGURE 3. Graph proof of guideline set homogeneity. (a) Seven nodes represent the guidelines in Table 2, and lines between nodes indicate an overlap or conflict relationship between two guidelines. (b) The magenta nodes resulted from the application of our approach in the banking system example. None of these four guidelines are connected directly to another.

2. Reject any guideline that has a negative effect on NFAs with higher priority.
3. For two overlapping or conflicting guidelines between two NFAs, when the two attributes have different priorities, reject the guideline that supports the lower-priority attribute; and when the two attributes have the same priority, reject one according to software engineer's preference.
4. For two overlapping guidelines for the same NFA, reject one of them according to the software engineer's preference.

An engineer's preferences in steps 3 and 4 would likely reflect his or her experience.

Because applying these steps manually can be difficult and error prone, we developed an algorithm as a basis for a software tool that can assist the process. Figure 1 presents a pseudocode implementation of the algorithm.

Example Application

To demonstrate our approach, we describe how it applies to a banking software system. Figure 2 illustrates the process, which begins with a prioritized list of NFAs and adopts the guidelines, effects, and relationships illustrated in Tables 1 and 2.

1. The first NFA is security. Three guidelines {G1, G2, G3} have positive effects on security, but G1 and G2 overlap, so the software engineer must choose between them.
2. In this case, the engineer prefers G1.
3. As a result, the final guideline set includes {G1, G3} and excludes G2.
4. The second NFA is availability. Again, three guidelines {G1, G4, G5} have positive effects on availability.
5. G1 is already in the final guideline set.
6. G4 conflicts with G3, which is also already in the final set, so G4 is rejected.
7. Consequently, only {G5} is added to the final guideline set.
8. Performance has three supporting guidelines: {G3, G6, G7}.
9. G3 is already in the final set.
10. G7 has a negative effect on security, which has higher priority than performance.
11. Only {G6} is added to the final set.

The final guideline set for the banking system is {G1, G3, G5, G6}.

We can use graph theory to prove the final set's homogeneity. The nodes in Figure 3a represent the seven guidelines in Table 2, and the lines indicate an overlapping or conflicting relationship. We want to find a set that includes neither of these relationships. With this condition, a valid set comprises the nodes that aren't directly connected to each other.

Figure 3b highlights the guideline set resulting from the banking example. None of these four guidelines is connected directly to another. The final set is therefore homogeneous.

Our approach manages the potential overlaps and conflicts in applying multiple guidelines to improve NFA quality. It leads software engineers step by step in selecting a suitable guideline set to apply throughout system development, enabling a preventive approach to quality improvement. Because NFAs are one aspect of software quality, improving their achievement likewise improves the system's overall quality.

Our approach's effectiveness depends on the strength and validity of the dataset used. Collecting more guidelines during the preparation stage generates more opportunities for applying them. Correctly determining a guideline's effect on NFAs and its relationship to other guidelines is essential to success.

The software engineer's experience also affects the approach's effectiveness. Having a field expert assign a weight to each guideline relative to each NFA might help guarantee an optimal decision between overlapping guidelines during the application stage, but its cost might be high.

By default, our approach aims to achieve the highest quality NFAs according to their priority. However, some NFAs might require a specific quality level. Quantification techniques, such as those developed by Glib and Brodie,[10] could enable achievement of a targeted NFA quality level—though not necessarily the highest level.

ABOUT THE AUTHORS

MALIK HNEIF is a postgraduate student at the University of Malaya, currently serving as a research assistant on a project to improve the quality of nonfunctional software attributes. His research interests include software development methodologies and quality. Hneif received his master's degree in software engineering from University of Malaya. He's a member of IEEE. Contact him at malik.hneif@gmail.com.

SAI PECK LEE is a professor in the Department of Software Engineering at the University of Malaya. Her research interests include object-oriented techniques, computer-aided software engineering tools, software reuse, requirements engineering, and software quality. Lee received her PhD in computer science from Université Panthéon-Sorbonne (Paris I). She's a member of IEEE and a founding member of Informing Science Institute. Contact her at saipeck@um.edu.my.

References

1. S.H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed., Addison-Wesley, 2002.
2. *ISO/IEC TR 9126, Software Engineering–Product Quality*, Int'l Organization for Standardization, 2000.
3. G.M. Weinberg, *Quality Software Management: Anticipating Change*, Dorset House, 1997.
4. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley Professional, 2003.
5. B. Andreopoulos, "Satisficing the Conflicting Software Qualities of Maintainability and Performance at the Source Code Level," *Proc. Workshop em Engenharia de Requisitos* (WER 04), 2004; http://wer.inf.puc-rio.br/WERpapers/artigos/artigos_WER04/Bill_Andreopoulos.pdf.
6. R.G. Dromey, "Software Quality—Prevention versus Cure?" *Software Quality J.*, vol. 11, 2003, pp. 197–210.
7. B. Andreopoulos, "Achieving Software Quality Using the NFR Framework: Maintainability and Performance," *Proc. 3rd Int'l Conf. Computer Science, Software Eng., Information Technology, e-Business, and Applications* (CSITeA 04), 2004; www.cse.yorku.ca/~billa/MULIC/AndreopoulosCSITeA.pdf.
8. M. Hneif and S.P. Lee, "Guideline-Based Approach for Achieving Non-functional Attributes of Software," *Proc. Int'l Conf. Computer Eng. and Technology* (ICCET 10), IEEE Press, 2010, pp. 305–308.
9. D. Budgen, *Software Design*, 2nd ed., Addison-Wesley, 2003.
10. T. Gilb and L. Brodie, *Competitive Engineering*, Butterworth-Heinemann, 2005.