NFR's: Fact or Fiction?

Janet E. Burge and David C. Brown

Computer Science Department
WPI, Worcester, MA 01609, USA.

Abstract

Requirements, derived from the customers' needs and desires, are used to guide software development and to determine if the completed system is what the customer needs. Requirements Engineering (RE) captures and represents system requirements so that they can be traced through to both implementation and testing to ensure that the resulting system does what the customer has requested.

The primary focus of RE has been on the functional requirements: ensuring that the necessary functionality of the system is delivered to the user. The non-functional requirements (NFRs), however, are also important since they contribute to the overall quality of the resulting system. RE has treated NFRs as something separate and distinct from the functional requirements.

This raises several questions: 1. What does "functional" really mean? 2. Are NFRs really non-functional or are they a form of functional requirements? 3. Are NFRs really requirements? 4. Can NFRs be represented as and by functional requirements?

In this paper, we begin by discussing the definitions of functional requirement, non-functional requirement, software requirement, and function. We then examine NFRs to see if the reasons that normally distinguish them from functional requirements are valid in the context of the definition of function. Finally, we discuss how re-defining NFRs can assist in assuring that they are met and discuss some approaches that can be used.

1 Introduction

Customer requirements are the foundation upon which a software system is built. These requirements, derived from the customers' needs and desires, are used to both guide the development of the system and to determine if the completed system is what the customer requested. Because of its importance, requirement specification has become a research area known as Requirements Engineering (RE) both in Software Engineering [23, 15] and Systems Engineering [9, 5].

The primary goal of Requirements Engineering is to capture and represent system requirements so that they can be traced through to both implementation and testing to ensure that the resulting system does what the customer has requested. Requirements are commonly broken into two types: functional requirements (FRs) that correspond to desired functional capabilities of the system and non-functional requirements (NFRs) that describe desirable overall properties that the system must have (such as being "cost-effective" or "user-friendly"). NFRs generally are not directly related to specific system components and often involve aggregate system behavior [14].

The primary focus of requirements engineering has been on the functional requirements:

ensuring that the necessary functionality of the system is delivered to the user. The NFRs, however, are also important since they contribute to the overall quality of the resulting system. RE research has treated NFRs as something separate and distinct from the functional requirements. This raises several questions:

1. What does "functional" really mean?
2. Are NFRs really non-functional or are they a form of functional requirements?
3. Are NFRs really requirements?
4. Can NFRs be represented as or by functional requirements?

In this paper, we examine if NFRs really are separate and distinct from FRs. First, we begin by discussing the definitions of "functional requirement", "non-functional requirement", "software requirement", and "function". We then examine NFRs to see if the reasons that normally distinguish them from functional requirements are valid in the context of the definition of function. Finally, we look at how re-classifying NFRs can assist in ensuring that they are satisfied.

2 Requirement Definitions

In many cases, terms such as function, functional requirements, and non-functional requirements are used without being defined. Before discussing NFRs, it is useful to describe what they are and give some examples. The following sections present definitions of functional requirements and non-functional requirements.

2.1 Functional Requirements

Functional requirements (FRs) define what the system must do [22, 20]. Roman [17], describes FRs as capturing "the nature of the interaction between the component and its environment". These are both somewhat vague definitions since there are likely to be many things that a system "does", not all of which may be needed by the user. Since the FRs are required things, they must be desired by the customer, and intended by the designer.

If the system being designed was an air traffic control system, a functional requirement might be that aircraft positions must be reported with an accuracy of within two nautical miles. The requirement does not specify how to achieve that accuracy, only that the accuracy is necessary. Another important feature of the requirement is that it is "testable" (i.e., it is possible to demonstrate that the requirement has been met).

2.2 Non-Functional Requirements

While functional requirements describe what the system or device should do, non-functional requirements are concerned with the manner in which the system or device should accomplish that function given "the constraints of a non-ideal world" [20]. Roman [17] describes NFRs as restricting the types of solutions under consideration. Roman also refers to the NFRs as "constraints". For example, there are likely to be any number of ways in which a given functional requirement can be met. NFRs provide guidance on differentiating between these solutions. For example, if the NFR concerns performance, solutions that are faster will be preferred and solutions with a poor performance should be rejected.

NFRs are typically described as attributes of the system or device that contribute to the overall quality of the product. These attributes are not confined to one portion of the product's functionality. In a software system, these include the "ilities" such as reliability, security, scalability, extensibility, manageability, maintainability, interoperability, composability, evolvability, survivability, affordability, understandability, and agility [10], as well as other system wide properties, such as performance, security, availability, modifiability, adaptability, nomadicity, survivability, evolvability, and responsiveness [8].

Other types of NFRs include quality of service parameters (QoS) such as performance parameters [14] and system attributes (also referred to as properties) such as "ease-of-use" that apply to the system in general but cannot be phrased as a task that the system performs [13].

Roman views NFRs as constraints (referring to them as both non-functional require-

ments and non-functional constraints). Roman lists several constraint categories: interface constraints, performance constraints, operating constraints (such as personnel availability), life cycle constraints (maintainability, enhanceability, etc.), economic constraints (such as cost), and political constraints (such as avoiding use of a competitor's device).

NFRs tend to be very general and are likely to be desirable to varying degrees in different systems. For example, in a military system, user safety may be extremely important, while in a banking system there are few user safety concerns. There are often conflicts between the NFRs that result in tradeoffs being made. Typical examples would be trading off cost versus safety or resources used versus performance.

2.3 Software Requirements

When designing software, the consideration of requirements falls into a sub-field of Software Engineering known as Requirements Engineering. There are several RE areas where research is being done [15]: eliciting requirements, modeling and analyzing requirements, communicating the requirements, reaching agreement on requirements, and evolving requirements. There is also work done specifically on NFRs. For example, the NFR Framework [6] represents the NFRs as goals that must be satisfied by the system. The system design consists of a goal-graph giving the NFRs, alternative ways of satisfying them, and claims for and against these alternatives.

One aspect of software engineering that differs from the design and manufacture of other systems is that one of the largest costs is not the development, but instead is the maintenance. That is because software systems are mutable and can change over time, either due to finding and correcting defects in the original system or by responding to the customer's changing needs.

One way to keep software costs down is to reuse or modify existing systems for new needs. This makes the ability to keep track of requirements even more crucial. Requirements for the new system (both FR and NFR) will consist of some or all the requirements met by the reused system (if not, then it is likely that the cost saved by reuse and modification will be minimal) plus some additional ones. It is important to ensure that the modifications result in the system meeting the new requirements but do not cause the system to violate any prior requirements.

3 Functional versus Non-Functional

Before examining more closely what makes a requirement functional, we will first look at the meaning of "function". Researchers have studied the definition and representation of function for a variety of engineered artifacts, and have also studied how to reason using those representations [21].

3.1 Function

Chandrasekaran and Josephson [3, 4] define function by viewing it as a set of effects that an entity has on its environment. These effects must be desired by some agent in order for them to have meaning as a function. Otherwise they will be spurious, just as a clock's ticking does not help someone know the time.

Functions are usually utilized by agents in the environment (e.g., users) to achieve goals. Typically, designers intend effects and users desire them. Effects may be behaviors or properties of the functioning entity: e.g., a clock's second hand moving, or a chair's flat seat.

Chandrasekaran and Josephson refer to the "mode of deployment" as the way that causal interactions between the entity and the environment are instantiated. Some things only function when in certain physical relationships to the environment (e.g., "plugged in") or when the environment acts on them in some way (e.g., "press button to start").

If a set of behavioral constraints on the environment are satisfied when the entity is correctly deployed then that entity can be said to play a "role" in that environment. Behavioral constraints are any constraints on the behavior of the environment, such as a required voltage being achieved, or some condition producing some action.

If the entity "plays a role" in the environment and that role is desired by some agent in the environment, then the entity has (or performs) a function. The actual function is defined by the constraints being satisfied. The agent desiring that role can be thought of as the "user", but it might also be some other entity.

This comprehensive and general view of functionality is highly compatible with Roman's [17] statement that functional requirements involve the interaction between the object and its environment.

3.2 Function and Software

If functional requirements can be described by their effect on the environment, then in order to examine functional requirements we must first look at what environments are important for software. With its long life cycle, software exists in a number of different environments either defined by the sequence of time (first *development*, then *operation*) or by who is interacting with the software (the *developer/maintainer* or the *user*). There are many ways that the environments could be delineated, but for this paper we will look at three different environments: development, operational, and maintenance.

The development environment may have its own requirements (functional and nonfunctional) that are not directly related to the task that the system performs. A typical development requirement would be one that requires the development team to write the software in a particular language. For example, in the early 1980s, the Department of Defense mandated that government software be written in Ada. There may also be requirements on the hardware platform, especially if the software is going to be run on equipment already owned by the customer.

The operational environment is the one where most customer requirements apply. Most requirements defined as functional apply to this environment—these requirements specify what services the software must provide. An example would be a requirement for an air traffic control system that specifies the target tracking accuracy.

The maintenance environment must also be considered. This may not always be the same as the development environment. There may be requirements that affect maintenance specifically. An example would be a requirement to allow expansion of the software capacity in some way, like responding to additional data storage needs or additional users. These expansions would take place after the software is deployed.

If the functional requirements define how the system must interact with its environment, then all of the different environments the system exists in must be considered. Many of the requirements normally referred to as nonfunctional (such as scalability, extensibility, and others) refer to the development (or maintenance) of the system rather than its deployment. When the environment under consideration is development or maintenance, these requirements *do* have an effect on that environment. This suggests that many of the NFRs can be considered as functional when looked at from the point of view of the environment to which they apply.

4 What are NFRs?

So what differentiates an NFR from an FR? A wide variety of different types of NFRs have been described—what do they have in common?

4.1 NFR Characteristics

There are a number of characteristics (or missing characteristics) of NFRs that are used to distinguish them from FRs:

1. NFRs do not describe something that the system "does", i.e. they are not requirements that describe a function that the system performs;
2. NFRs do not relate to a specific system component, instead they "cross-cut" software functionality;
3. NFRs can not be evaluated without looking at the system as a whole—this particularly applies to NFRs that involve end-to-end performance.

The question is: do these characteristics automatically mean that the requirement is non-functional? For the first characteristic, if this is combined with Chandrasekaran's definition of function, then many NFRs do perform a function. For the second characteristic, the cross-cutting functionality may actually imply that the NFR is really an abstract description of many more specific functional requirements that apply to the various functions that the software performs. For the third item, the need to have the entire system in place is not unique to performance-based NFRs, there are probably many functions of the system that can not be evaluated until the system is complete.

4.2 Abstract FRs or Preferences

One way to handle an NFR is to break it down into the associated FRs that together would satisfy the requirement. For example, the NFR of affordability can be translated into specific requirements on the cost of the system if the range of acceptable costs can be made known. An NFR regarding accuracy could be broken into a number of more specific requirements that indicate what needs to be accurate and what "accurate" means in each case.

In some cases, however, this is not possible. For example, if the requirement was that the system be "as cheap as possible" there would not be a way to translate that into a specific dollar amount that can be tested. In this case, there may be a range of acceptable costs but the requirement cannot be expressed in terms of a specific one. The only way to determine if the system was "as cheap as possible" would be to compare the costs of alternative designs. Even such a comparison is no guarantee unless *all* possible alternatives are known.

In this case, and others, the *degree* to which the NFR is met may vary. This is different from more concrete, functional requirements where testing the requirement results in a pass/fail answer. Indicating that the system should be "as cheap as possible" is *not* a requirement. If it were a requirement then there would need to be a way to ensure that it has or has not been met. Instead, this is a *preference* indicating that when given a choice between a solution with a higher cost and one with a lower cost, the lower cost alternative is preferable. Chung, et. al. [7] discuss the notion of satisficing [19], looking for solutions that are "good enough" although not necessarily optimal.

This means that some NFRs may actually be expressed as both requirements and preferences. They may be refinable to functional requirements that specify the threshold of acceptability and may also include additional preferences that indicate that solutions that "beat" the threshold should be preferred.

5 An NFR Portfolio

If something is considered a requirement, whether functional or non-functional, there needs to be some way to verify that it has been met. Listing the NFRs that apply to a given system can be a useful aid to defining design and development priorities but there needs to be sufficient rigor applied to ensure that these priorities hold. Here we list a portfolio of NFR techniques that can be used to assist in following and testing NFRs.

One method, already mentioned, is to translate the NFR into a set of FRs that together will ensure that the NFR is met. The new FRs can then be added to the initial set of FRs and used and tested in a similar fashion. Since NFRs are general, the refinement to more specific FRs could be aided by a catalog of knowledge describing likely refinements.

Chung, et. al. [7] have designed an NFR Framework that uses NFRs as the basis for system design. In this approach, Softgoal Interdependency Graphs (SIGs) are used to refine the NFRs and show relationships between them. The softgoals in the NFR framework describe how the more general NFRs apply to a specific system. As part of the framework, catalogs are provided and used that provide information about specific NFRs, development techniques for meeting requirements, and correlations and tradeoffs among softgoals. While they do not go so far as to state that their refined NFRs are actually FRs, many of them could be described as such.

Testing the NFRs is very difficult, but necessary if they are to have any real meaning in the system. One approach is to associate each

NFR with a battery of tests that could indicate either compliance with the NFR or non-compliance. Tests from this set would be selected (and even refined) for the specific system being tested based on knowledge of the related FRs and the system architecture. The tests would not *prove* NFR satisfaction but they would serve as an indication of the degree to which the NFR is likely to be satisfied.

Another possible way to test NFRs is by taking advantage of known tradeoffs between them. If it is known that a particular tradeoff takes place in a system between two NFRs, and only one of these is considered important for the system under development, it might be easier to test for the lack of the less desirable NFR and use that as a possible indicator that the other is satisfied. The success of this approach hinges on how strong the relationship is between the two NFRs. It may be better at pointing out non-compliance than compliance but could still be a useful addition to a battery of standard NFR tests.

These approaches look at NFR testing and verification, it is also important to ensure that the NFRs are met during development (or maintenance). One way would be to develop an NFR critiquing system [18] that would make use of compiled experiential design knowledge [1] to detect design decisions and software characteristics that tend to lead to lack of satisfaction for an NFR.

Some critiquing systems make use of design rationale. One example of a critiquing system using rationale is Janus, [11] which looked at kitchen designs and their conformance or non-conformance to rules describing good design. InfoRat [2] looked at design rationale to test for inconsistency and for tradeoff violations.

If the NFRs that guided the design decisions are clearly documented along with the decisions (i.e., as rationale), it is then possible to look at what the priorities were for the design and if they are consistent with the requirements. Consistent thought about an NFR during development, as revealed by rationale, is a good sign, while decisions without adequate rationale are bad. The rationale-driven approach also has the advantage that the NFRs can have relative priorities assigned to assist in conflict resolution.

Another use of rationale is in selecting between different pre-existing design and architecture structures. Attribute Based Architectural Styles (ABAS) [12] offer a mechanism to design for various quality attributes at the architectural level. Each style is designed to address a specific quality attribute. This associates the architecture with the reason, or rationale, for using it. The intention is to have a collection of styles so that the architect can combine them as needed to form the complete system architecture. The rationale, which includes the NFRs, can be used to select between the different styles. This applies to other design elements as well. For example, rationale was used by Vadhavkar and Peña-Mora [16] to select between different design patterns.

6 Summary and Conclusions

In this paper we have addressed these questions: what does "functional" mean, whether NFRs are really non-functional, whether NFRs are requirements at all, whether NFRs be represented by FRs, and what techniques might be employed to ensure and indicate NFR satisfaction.

Every requirement affects the system in some way, otherwise it would not be worth requiring. This applies both to requirements typically classified as functional and those typically classified as non-functional. If something is required, there should be some way to determine if that requirement has or has not been satisfied.

This does not imply that un-testable NFRs such as "minimize cost" are not valuable—such factors are important to consider so that the system provides the maximum degree of user satisfaction. Such requirements, however, are not really requirements but are actually preferences.

Viewing NFRs as being separate and distinct from FRs has been an obstacle in the way of discovering methods to ensure that these requirements are met. In this paper, we look at how function has been defined for other design domains and use that definition to show that a requirement being defined as functional or non-

functional is a matter of how it is viewed.

A significant danger of keeping NFRs in a class by themselves is that it allows the developer to avoid looking beneath them to see how they affect the functional requirements of the system. In many cases, the NFRs can be refined into FRs that together satisfy the original NFR.

The other issue that must be addressed is how to verify NFRs that cannot be refined. In this case, the NFRs can be treated as preferences and the developer should look toward avoiding conflict as well as ensuring compliance.

In the previous section we described some possible ways of ensuring that NFRs, whether abstract FRs or preferences, could be addressed. Since many NFRs describe the overall quality of the system, rigorously ensuring that they are followed by using critiquing, rationale, or an approach similar to the NFR Framework [7], will help to build a higher quality system.

Supported by information describing tradeoffs, NFR to FR refinements and relationships between the NFRs, a portfolio of techniques will increase the chance that the system meets the NFRs and that customer satisfaction results.

Acknowledgements

The authors would like to acknowledge the assistance and support provided by Prof. George T. Heineman.

About the Author

Janet Burge is a PhD student and instructor at Worcester Polytechnic Institute doing research on using Design Rationale in Software Maintenance. She has B.S. and M.S. degrees in Computer Science and is a student member of the ACM. Her master's thesis was on "Knowledge Elicitation for Design Task Sequencing Knowledge." She also works part-time as a Senior Research Associate at Charles River Analytics, a small AI R&D company.

David Brown is Professor of Computer Science at Worcester Polytechnic Institute. He has B.Sc., M.Sc., M.S. and Ph.D. degrees in Computer Science, and is a member of the ACM, IEEE Computer Society, AAAI, and IFIP WG 5.2. He is the Editor of the Cambridge UP journal AIEDAM: AI in Engineering, Design, Analysis and Manufacturing; and is on the Editorial Boards of several Journals, including: Concurrent Engineering: Research and Application; Research in Engineering Design and the International Journal of Design Computing.

References

[1] D.C. Brown. Compilation: The hidden dimension of design systems. In H. Yoshikawa and F. Arbab, editors, *Intelligent CAD*, volume III. North-Holland, 1991.

[2] J.E. Burge and D.C. Brown. Inferencing Over Design Rationale. In J. Gero, editor, *Artificial Intelligence in Design '00*, pages 611–629. Kluwer Academic Publishers, 2000.

[3] B. Chandrasekaran and J.R. Josephson. Representing Function as Effect. In *Proc. of the Functional Modeling Workshop*, Paris, France, 1997.

[4] B. Chandrasekaran and J.R. Josephson. Function in Device Representation. *Engineering with Computers, Special Issue on Computer Aided Engineering*, pages 162–177, 2000.

[5] B. Chandrasekaran and H. Kaidal. Representing Functional Requirements and User-system Interactions. In *Proc. of the AAAI Workshop on Modeling and Reasoning about Function*, pages 78–84, Portland, Oregon, 1996.

[6] L. Chung, B.A. Nixon, and E. Yu. Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design. In *Proc. of the ICSE-17 Workshop on Architectures for Software Systems*, pages 31–43, Seattle, WA, USA, 1995.

[7] L. Chung, B.A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in*

Software Engineering. Kluwer Academic Publishers, 2000.

[8] L. Chung and E. Yu. Achieving System-Wide Architectural Qualities. In *Proc. of the OMG-DARPA MCC Workshop on Compositional Software Architectures*, Monterey, CA, USA, 1998.

[9] M. Dorfman. System and software requirements engineering. In R.H. Thayer and M. Dorfman, editors, *System and Software Requirements Engineering*, pages 4–16. IEEE Computer Society Press, Los Alamitos, CA, 1st edition, 1990.

[10] R.E. Filman. Achieving Ilities. In *Proc. of the Workshop on Compositional Software Architectures*, Monterey, CA, USA, 1998.

[11] G. Fischer, A. Lemke, R. McCall, and A. Morch. Making argumentation serve design. In T. Moran and J. Carroll, editors, *Design Rationale Concepts, Techniques, and Use*, pages 267–294. Lawrence Erlbaum Associates, 1995.

[12] M. Klein and R. Kazman. Attribute-based architectural styles. Technical Report CMU/SEI-99-TR-22, CMU/SEI, 1999.

[13] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, 1997.

[14] F. Manola. Providing Systematic Properties (Ilities) and Quality of Service in Component-Based Systems. Technical report, Object Services and Consulting, Inc., February 1999.

[15] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *Proc. of the Conference on The Future of Software Engineering*, pages 35–46, Limerick, Ireland, 2000.

[16] F. Pena-Mora and S. Vadhavkar. Augmenting design patterns with design rationale. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, pages 93–108, 1996.

[17] G. Roman. A Taxonomy of Current Issues in Requirements Engineering. *Computer*, pages 14–22, April 1985.

[18] B.G. Silverman. Building a Better Critic: Recent Empirical Results. *IEEE Expert*, pages 18–25, April 1992.

[19] H.A. Simon. *The Sciences of the Artificial*. MIT Press, 1981.

[20] R.H. Thayer and M. Dorfman. Introduction, issues, and terminology. In R.H. Thayer and M. Dorfman, editors, *System and Software Requirements Engineering*, pages 1–3. IEEE Computer Society Press, Los Alamitos, CA, 1st edition, 1990.

[21] Y. Umeda and T. Tomiyama. Functional Reasoning in Design. *IEEE Expert*, pages 42–48, April 1997.

[22] R.T. Yeh and P.A. Ng. Requirements analysis – a management perspective. In R.H. Thayer and M. Dorfman, editors, *System and Software Requirements Engineering*, pages 440–461. IEEE Computer Society Press, Los Alamitos, CA, 1st edition, 1990.

[23] P. Zave. Classification of research efforts in requirements engineering. *ACM Computing Surveys*, pages 315–321, December 1997.