5-2003

Categorizing Non-Functional Requirements Using a Hierarchy in UML.

James David Moody
East Tennessee State University

Recommended Citation

Moody, James David, "Categorizing Non-Functional Requirements Using a Hierarchy in UML." (2003). *Electronic Theses and Dissertations.* Paper 763. https://dc.etsu.edu/etd/763

Categorizing Non-Functional Requirements Using a Hierarchy in UML

A thesis
presented to
the faculty of the Department of Computer and Information Sciences
East Tennessee State University

In partial fulfillment
of the requirements for the degree
Master of Science in Applied Computer Science

---

by
James D. Moody, II
May 2003

Keywords: Software Engineering, Requirements Engineering, UML

ABSTRACT

Categorizing Non-Functional Requirements Using A Hierarchy in UML

by

James D. Moody, II

Non-functional requirements (NFRs) are a subset of requirements, the means by which software system developers and clients communicate about the functionality of the system to be built. This paper has three main parts: first, an overview of how non-functional requirements relate to software engineering is given, along with a survey of NFRs in the software engineering literature. Second, a collection of 161 NFRs is diagrammed using the Unified Modeling Language, forming a tool with which developers may more easily identify and write additional NFRs. Third, a lesson plan is presented, a learning module intended for an undergraduate software engineering curriculum. The results of presenting this learning module to a class in Spring, 2003 is presented.

CONTENTS

LIST OF TABLES

CHAPTER 1
INTRODUCTION

This paper is divided into three main parts. First, a broad overview of non-functional requirements (NFRs) is given, along with a survey of NFRs in the software engineering literature. Second, a set of one hundred and sixty one NFRs are categorized using the Universal Modeling Language (UML). The purpose of this categorization is to provide developers with a tool to more easily identify and write effective NFRs. Finally, a learning module covering NFRs is presented. The learning unit is intended for an undergraduate software engineering curriculum. Following the module, a summary of one instance of its application to a class of software engineering students in the Spring of 2003 is given, along with pre- and post-tests which demonstrate its efficacy.

CHAPTER 2

NON-FUNCTIONAL REQUIREMENTS IN SOFTWARE ENGINERING

A survey of NFRs in the software engineering literature

A common problem in software engineering, *defect amplification*, is the tendency of continued software development to exacerbate an uncorrected problem from an earlier stage of the software process [PRE01, 266]. Defect amplification occurs because (1) the customer of a software system neglected to tell the developer of a necessary system requirement, (2) the customer of the software system didn't state a system requirement clearly enough, or (3) the developer of the software system didn't understand the customer's requirement clearly enough. [PRE01, 267]. The defect that the developer unwittingly builds into the system is amplified as the project is built.

For efficiency, it is absolutely essential that the early phases of a software project be completed as accurately as possible. In particular, careful work during the first phase of the standard software development process—the gathering and analysis of user requirements—is absolutely essential for efficient software development. As Brooks notes, "The most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements." [BRO86, 182]

There are two tools used in this specification phase: functional requirements and non-functional requirements (NFRs) [PRE01, 266]. Functional requirements describe, in detail, the exact set of states that the software system may assume. Functional requirements concern the data input and output; they can be easily quantified and tested. Non-functional requirements constrain a system's structure or operation but do not describe any of the system's input, processing, or output. Three examples of non-functional requirements are performance, reliability, and security [CHU99, 211]. A given system may meet the exact needs of the customer/user, but if that system takes too long to process data, exhibits strange behavior, or provides inappropriate access to unauthorized users, the system is deemed unacceptable. Non-functional requirements are harder to test and quantify that functional requirements. As such, extra testing measures are required of the system outside the scope of the customer's software requirements specification.

Functional versus non-function requirements

Pfleeger defines functional requirements by writing that they "describe an interaction between the system and its environment"; her definition of NFRs is that they "describe a restriction on the system that limits the developer's choices for constructing a solution to the problem." [PFL87, 291]

Sage defines a system's functional requirements as directly indicating the quantifiable, measurable aspects of the system. Non-functional requirements specifications, Sage writes, "describe those constraints, alterables, or limitations that relate to the structural or purposeful aspects of system requirements specifications." [SAG90, 88]

Sage writes that the central purpose of functional requirements is to describe all the system capabilities and functionality that the customer/user desires. If there are changes made to the system requirements specification document after the initial requirements gathering phase or later in the software development cycle, Sage writes, then the "functional requirements specifications must be modified to reflect the current system requirements." [SAG90, 88]

The requirements phase

The requirements specification phase is called "concept exploration" by Charette. [CHA86, 313] He contrasts functional and non-functional requirements in three ways. First, he writes that functional requirements are necessary, since they are the "shall" specifications of the software product. The NFRs he deems as optional, since they deal with aspects of the system that are not easily quantifiable. Second, he writes that functional requirements can (and must) be tested, whereas NFRs are "not conclusively testable." [CHA86, 177] Finally, he writes that since functional requirements are quantifiable and must be tested, they are objective, whereas NFRs, not being an absolute precondition for the correct functioning of the system, are subjective. Charette gives two examples of NFRs: maintainability and ease of use. [CHA86, 177]

Davis dedicates two long chapters on specifying functional and non-functional requirements in the software requirements phase. The basis for his chapter on NFRs is Boehm's 1976 Software Quality Characteristics Tree. Davis uses the seven qualities in the third tier of Boehm's tree to categorize NFRs: portability, reliability, efficiency, human engineering, testability, understandability, and modifiability [DAV93, 241].

Portability is "the degree to which software running on one host computer environment can easily be converted to one running on another host computer environment." [DAV93, 244] Davis writes that portability need not be addressed if the application is to be short-lived, impossible to port, or if the system will never be upgraded. Reliability is "the ability of the software to behave consistently in a user-acceptable manner when subjected to an environment in which it was intended to be used." [DAV93, 244] Reliability is key in critical and real-time systems; "99.999 percent reliable" is a term Davis discusses in terms of patient-monitoring systems and military systems. Efficiency refers to "the level at which the software uses scarce system resources." [DAV93, 245] These scarce system resources include memory, internal and external disk space, machine cycles (timing constraints), buffers and communication channels. Of all the non-functional requirements he lists, Davis indicates that efficiency is the NFR that is easiest to quantify.

Human engineering, Davis writes, is that NFR that deals with the user interface, the border where the user and the system meet. Davis traces the evolution of user interfaces, from command languages to menu-based systems to "picking devices" (mouses and joy sticks) to multiple widows. Davis writes that software designers may be tempted to design user interfaces, but this task falls more naturally in the domain of the requirements writer.[DAV93, 251]

According to Davis, "Testability, understandability, and modifiability are closely related." [DAV93, 260] These NFRs, Davis says, are among the most important contributors to the cost of the life cycle, but are the hardest to quantify. Specifying required levels of cohesion (how closely related a module's tasks are) and coupling (the interrelatedness of two modules) in the design is the best a software designer can do with these three NFRs. [DAV93, 260]

Pfleeger begins her discussion of software requirements in the requirements phase by breaking them into two parts: a requirements definition document, which is a document written in language the customer understands, and a requirements specification document, which is written with enough technical detail for the software developer to be able to satisfy the requirements. [PFL87, 190] As an example of the difference in the two types of requirements, she introduces a weekly paycheck printing system. Pfleeger writes that examples of functional requirements are states only of interest to the inner workings of the system (for instance, what input is necessary for a paycheck to be printed?). Examples of NFRs are constraints on the system (for instance, what type of computer must the system be developed on?).

Sage introduces the idea of writing both a functional requirements specification and a non-functional requirements specification. [SAG90, 301] Sage says, "The functional requirements specifications may be expressed in natural language, structured languages (e.g., structured English), or in any of the formal specification languages." [SAG90, 301]

Kotonya and Sommerville write that non-functional requirements "place restrictions on the product being developed and the development process, and they specify external constraints that the product must meet." [KOT98, 287] Because NFRs are restrictions or constraints on system services, Kotonya and Sommerville claim that they are often of "critical importance." They even suggest that sometimes functional requirements may need to be sacrificed to meet the constraints of NFRs.

Unlike other authors of software engineering texts, Kotonya and Sommerville assert that "there is not a clear distinction between functional and non-functional requirements." [KOT98, 112] As an example of this lack of distinction between the two, they offer the following example: a system will ensure that data is protected from unauthorized access.

Using previous definitions, this requirement seems like a non-functional requirement. Unauthorized access would seem to fall under the category of system security. It is an NFR because it "does not specify specific system functionality which must be provided." [KOT98, 112]

However, if the requirement were specified in more detail, such as "the system shall include a user authorization procedure in which users must identify themselves using a login name and password," then the requirement starts to look more functional than non-functional. [KOT98, 112] Kotonya and Sommerville explain that abstract non-functional requirements, when decomposed and explained in greater detail, may become functional requirements.

Kotonya and Sommerville further subdivide non-functional requirements into three categories: process requirements, product requirements, and external requirements. Process requirements are "constraints placed on the development process of the system." [KOT98, 114] These deal with the methods and documentation of the system as it is developed. Product requirements are "requirements which specify the desired characteristics that a system or subsystem must possess." [KOT98, 114] This is the area of specification that the customer/user has the most influence on and interaction with. External requirements are "requirements which may be placed on both the product and the process, and which are derived from the environment in which the system is developed." [KOT98, 115] These requirements may be based on many real-world factors, such as health and safety regulations, organizational considerations, and the system's need to interact with other systems.

There are several significant differences between functional requirements and non-functional requirements. Attributes of functional requirements, such as objective descriptions of system behavior, input, and output make documenting the requirements plausible and clear.

Malan and Bredemeyer divide non-functional requirements into two categories: qualities and constraints. Qualities are "properties or characteristics of the system that its stakeholders care about and hence will affect their degree of satisfaction with the system. Constraints are not subject to negotiation and, unlike qualities, are (theoretically) off-limits during design trade-offs." [MAL01, 327]

Both qualities and constraints, Malan and Bredemeyer write, must have five characteristics, which they call "SMART requirements:"

Specific: requirements are written without ambiguity, using consistent terminology, simple and at the appropriate level of detail.

Measurable: it is possible to objectively verify that this requirement has been met. Tests must be performed, and criteria must be met to verify that the requirement is met.

Attainable: requirements are technically feasible. A professional judgment of the technical "do-ability" of the requirement can be written

Realizable: requirements are realistic, given the resources. The staffing and skill are available. There is access to the development infrastructure and to the run-time infrastructure. There is enough time to complete the project.

Traceable: requirements are linked from their conception through their specification to their subsequent design, implementation, and test. [MAL01, 327]

Though there is seemingly little coverage of non-functional requirements in the software engineering literature, at least one book is completely devoted to the topic: Chung et. al.'s Non-Functional Requirements in Software Engineering . The focus of this book is threefold: defining the framework of non-functional requirements, categorizing non-functional requirements into types, and providing case studies and applications.

Chung uses the NFR framework as a starting point to drive the overall design process. According to the authors, "Most conventional approaches to system design are driven by functional requirements." [CHU99, 2] They assert that most developers concentrate solely on the functionality of the system, leaving decisions on *how* to achieve this functionality as a second thought. As such, the *how* aspect of the design process may go undocumented or ignored.

The purpose of the NFR framework is to "put non-functional requirements foremost in the developer's mind." [CHU99, 2] Chung et. al. identify eight major steps in the design process that relate to non-functional requirements:

- Acquiring or accessing knowledge about the particular domain and the system which is being developed, functional requirements for the particular system, and particular kinds of NFRs and associated development techniques
- identifying particular NFRs for the domain
- decomposing NFRs
- identifying "operationalizations" (possible design alternatives for meeting NFRs in the target system)
- dealing with ambiguities, tradeoffs, priorities, and interdependencies among NFRs and operationalizations
- selecting operationalizations
- supporting decisions with design rationale
- evaluating the impact of decisions [CHU99, 7]

It would be extremely helpful," the authors write, "if at each step in the (software development) process, the developer could draw on available knowledge that is relevant to that step in the process. This is what the NFR framework aims to provide." [CHU99, 7] The sole purpose of the NFR framework is to provide software developers using it a way to represent and analyze non-functional requirements.

To represent the design process visually, the authors introduce a structure they call a "softgoal interdependency graph." "Softgoals" are the cornerstone of the NFR framework. Softgoals are defined as being entities that "represent a goal that has no clear-cut definition and/or criteria as to whether it is satisfied or not." [CHU99, 53]

There are two purposes of this softgoal interdependency graph: to allow a software developer to keep an overall perspective of all softgoals (NFRs) that relate to the current project (the softgoals are connected by arrows to show their interconnectedness), and to allow a developer to impose a hierarchy on all softgoals, implying that some softgoals are dependent on others. [CHU99, 54] After the softgoal interdependency graph is drawn, a developer may use an evaluation procedure provided by the authors to determine if the softgoals are realistic and viable.

The second part of Chung et. al.'s book categorizes non-functional requirements into three major categories: accuracy requirements, security requirements, and performance requirements. They acknowledge that many other non-functional requirements (usability, portability, et cetera) could have been included in the categorization, but that to keep the scope of the NFR framework manageable, they choose the three they deem most important. To emphasize how broad the topic of NFRs is, they present a table of 161 distinct non-functional requirements.

Accuracy requirements, they write, assume that "accuracy is a fundamental semantic attribute of any information item." [CHU99, 188] Thus, every atomic "information item" has a single, absolute state: either it is accurate or it is not. They go on to assert that "the accuracy of information is often regarded as an inherent property of any automated information system," but in reality, accuracy doesn't happen automatically: developers have to consciously design accuracy into any given system.[CHU99, 192] Using the NFR Framework he introduced in the first part of the book, They produce an "Accuracy Requirements Framework" which customizes the NFR Framework to allow developers to instill a higher degree of accuracy into a project. Finally, they create a sample case and uses an evaluation procedure and the softgoal interdependency graph to measure the success of the design decisions made about accuracy.

Security requirements, Chung et. al. write, treat the protection of information "as an asset of an enterprise, just like money or any other forms of property." [CHU99, 214] The "Security

Requirements Framework" is the tool that emerges from their efforts to design security into a system. They categorizes system-related components of security into four categories: confidentiality, integrity, availability, and utility. Confidentiality of a system concerns unauthorized disclosure of information, such as the social security numbers of employees in a company. Integrity of a system concerns guarding against unauthorized updates or tampering, which would corrupt the accuracy requirements described above. Availability concerns guarding against the interruption of service. Utility verifies that the data being used fits into the correct structure; for example, a given integer is supposed to represent dollars, not yen.

Chung et. al. write "Security requirements are often ambiguous and invite many interpretations from different groups of people." [CHU99, 236] To solve this ambiguity, they introduce five decomposition methods that help "analyze security softgoals and clarify their meaning." [CHU99, 236] These decomposition methods divide broad softgoals into concrete, distinct sub-goals that can be implemented. Finally, they create a second sample case and uses an evaluation procedure and the softgoal interdependency graph to measure the success of design decisions about security requirements.

Performance requirements, they write, raise a challenge to software developers. Other non-functional requirements often diminish a system's performance. With performance requirements, developers face having to balance performance with other requirements. They present a "Performance Requirements Framework" to help a developer in "considering and selecting among competing implementation alternatives, justifying and recording decisions, and evaluating their impact on meeting non-functional requirements."[CHU99, 244]

All performance issues in any system, Chung writes, deal with a combination of main memory, secondary memory, response time, and throughput [CHU99, 271]. Performance can be classified as either space-related or time-related. Space-related performance can be further classified as being either main memory or secondary storage. Time-related performance can be classified as being either response time or throughput.

Chung gives three principles for building performance into systems. The first, the "centering principle," states that "efforts to improve performance should focus first on the critical and dominant parts of the workload." The second, the "processing vs. frequency tradeoff principle," states that "minimize the product of the processing time per execution of an operation, and the frequency of executing the operation." The third, the "fixing principle," states that "for good response time for users, 'fixing' (the process of mapping functions and information to instructions and data) is done as early as possible." [CHU99, 261]

Chung et. al conclude with two example case studies and applications of the NFR framework that illustrate the use of the softgoals interdependency graph and how non-functional requirements are satisfied. The two case studies presented are a credit card system and a tax-appeal administrative system.

In the credit card system, performance, security and accuracy requirements are examined, and workload and data flow are examined. The two functional requirements of this system are authorizing transactions and canceling stolen credit cards. They show how security and accuracy are balanced against the system's performance, and how the flow of information relates to system performance.

In the tax-appeal administrative system, performance, security and accuracy requirements are also met, and detailed design decisions for meeting each of the three are explained. The two functional requirements of this system are maintaining the status of tax-appeals and keeping statistics on all appeals. They show how the use of the "centering principle" improves system performance by focusing on the critical and dominant parts of the workload (maintaining status and keeping statistics).

Documenting functional and non-functional requirements

Functional requirements are documented with Use Cases. In the requirements elicitation and analysis phase of software engineering, the documentation of non-functional requirements is an area that has been neglected. [BAR01, 1] Non-functional requirements may be documented using Performance Cases, [BAR01, 2] though this is not yet a standard paradigm for documentation.

Perhaps the most common form of documenting functional requirements, called "Use Cases," was introduced by Cockburn. The purpose of a Use Case is to describe all the major system/user interactions that may take place. Table 1 shows a blank Use Case template.

USE CASE #		
Goal in Context		
Scope & Level		
Preconditions		
Success End Condition		
Failed End Condition		
Primary, Secondary Actors		
Trigger		
DESCRIPTION	**Step**	**Action**
	1	
	2	
	3	
EXTENSIONS	**Step**	**Branching Action**
SUB-VARIATIONS		**Branching Action**

Table 1: Blank Use Case form

One goal of writing a Use Case is to be neither too vague nor too specific. For example, a Use Case that has as its purpose "user performs a transaction" or "the system displays information" is too vague. The Use Case writer needs to address the following issues:

- The method in which the user will identify himself/herself to the system
- The method in which the system will verify the identity of the user attempting to use the system
- The resulting state if the system cannot correctly identify the user
- The resulting state if the user selects a transaction that is invalid

At the other end of the spectrum, a Use Case may go into a level of detail that is too great. For example, if the user is attempting to log on to a system, the Use Case does not need to describe every keystroke and mouse click. If the user clicks the mouse cursor when it is outside of the login form, a dialog box does not have to be displayed warning the user that the mouse-click did not apply to the log in attempt. The system is not responsible for checking every possible combination of keystrokes and mouse cursor locations.

Some attributes of non-functional requirements, such as description of system behavior and such hard-to-quantify aspects such as performance, reliability, and security, would seem to make documenting non-functional requirements more difficult. There are currently very few methods of documenting and defining non-functional requirements. One method is proposed by Malan and Bredemeyer [MAL01, 88]. Their method builds on Cockburn's Use Case model (see Table 2). To the modified Use Case form, a new field is added, labeled "Non-functional." This fields contains a list of the non-functional requirements that the use cases must meet. [MAL01, 88] The form they suggest is

<keyword>:<requirement>

where keywords include (but are not limited to) Performance, Reliability, Fault Tolerance, Frequency, and Priority. The requirement is expressed in language that the system's users/stakeholders and developers can understand. Malan and Bredemeyer's motivation for describing non-functional requirements is that the requirements can "drive architectural decisions and be used to validate the architecture." [MAL01, 88]

Two methods of documenting NFRs is to include performance requirements into Use Cases [COC01, 13]and to use the Software Requirements Specification (SRS) to simply list NFRs. According to Barrett, both methods have shortcomings: the first method "hides important details of non-functional requirements inside the Use Cases, possibly leading to their omission, and certainly leading to the impression that an NFR is subsidiary to the corresponding Use Case." With the second, "not every NFR may naturally be associated with a Use Case." [BAR01, 2]

To address these shortcomings, Barrett introduces the Performance Case, which he describes as intending "to raise the awareness of developers for NFRs." [BAR01, 2] Table 2 illustrates an example Performance Case.

PERFORMANCE CASE #		
Goal		
Level		
Preconditions		
Actor		
Type and Subtype		
Action Steps	**Step**	**Action**
	1	
	2	
	3	
EXTENSIONS AND EXCEPTIONS	**Step**	**Branching Action**
Use Case Links		
Design Links		
Rationale Links		

Table 2: Example Performance Case

The structure of the Performance Case is modeled on the structure of the Use Case. The first field is given a descriptive name that succinctly describes the specific NFR. The next field, "Goal," is written using concrete, measurable terms. In this example, a specific time interval is specified: exactly one half of a second.

The "type and subtype" field is what sets the Performance Case apart from the Use Case: here the specific non-functional requirements are described. Performance and average throughput are two candidate NFRs. Other broad NFRs (security, ease of use, maintainability, portability, modifiability, etc.) belong in this field.

Describing the "level" field of the Performance Case, Barrett writes, "As with Use Cases, the level of the case can be primary (describing one basic case) or summary (composed of primary cases)." [BAR01, 4] The "actor" field describes the entity that initiates the interaction with the system. In most cases, the actor is a human identified as the "user," but the actor may also be another system interacting with the system being described.

The "action steps" in the Performance Case are included to illustrate one particular user/system interaction. In the Performance Case, each pair of user/system steps serve to divide the task described in the "goal" field. Extensions and exceptions are handled as they are by Use Cases, with a list of error condition and action pairs. For example, a simple error message may be displayed. These extensions/exceptions, however, do not solve the problem raised by the system. The system developers must correct these problems, not the system itself.

The last section of Table 2 is intended to show the connections with other classes and other Performance Cases. The "Use Case Links" field associates other Use Cases with this Performance Case. The "Design Links" field describes the actual implementation of the system, referencing the names of the classes that objects will be instantiated when this Performance Case is enacted.

Software Metrics

A common complaint of several authors cited herein concerning NFRs is their lack of quantifiability as compared with functional requirements. If NFRs and their Performance Cases could be altered so that they could be more precisely tested and analyzed, the software process would benefit, and developers would be able to produce software of a higher quality with greater efficiency.[CHU99, 114] A metric is, in mathematical terms, a "measure function" (or "measurable function")-- a function that is definable on some definite structure, either abstract or concrete, discrete or continuous. [EJI91, 62]. According to Fenton, metrics are a method of measurement that "numerically characterize simple attributes like length, number of decisions, number of operators (for programs), the number of bugs found, cost, and time for processes." [FEN91, 244]

As mentioned above, Sage describes a system's functional requirements as directly indicating the quantifiable, measurable aspects of the system. These requirements may be measured with software metrics. Sage claims that software process metrics make NFRs more quantifiable. Using metrics, problems in reliability, maintainability, flexibility, and usability can be more easily tracked, corrected, and documented. [SAG90, 78]. Sage lists six potential problem areas: computer resource utilization; software development humanpower; system requirements specification stability; software progress, development, and testing; cost-schedule deviations; and use of software development tools. [SAG90, 78]

Non-functional requirements are requirements of a system that fall beyond the realm of the system being designed. Nevertheless, it is important that they be measured, and metrics may be used to measure NFRs as well as functional requirements.

In Software Metrics: A Rigorous Approach, Fenton argues that, although other software engineers have declared that certain software attributes like dependability, quality, usability, and maintainability are unmeasurable, a rigorous approach to constructing measurement models can make any unmeasurable item measurable. [FEN91, 133]

Fenton calls software metrics an "all-embracing term" that is given to a wide range of activities, including cost and effort estimation models and measures, productivity measures and models, quality control and assurance, data collection, quality models and measures, reliability models, performance evaluation, algorithmic/computational complexity, structural and complexity metrics. [FEN91, 134] Of these nine activities, only the last four seem to fall into the realm of non-functional requirements.

Fenton uses the terms "internal product attributes" and "external product attributes" to describe functional and non-functional requirements, respectively. Of the latter, he examines measurement methods on four attributes: quality, reliability, usability, and maintainability. In a later chapter, he uses the quality attribute as an umbrella term to include all the other non-functional attributes.

Of quality, Fenton says the concept is "too high level to be meaningful or measurable directly", and must therefore be decomposed into lower level attributes. [FEN91, 134] He does this with a tree/hierarchical diagram, where quality and the other high level attributes are the root of the hierarchy. The high-level attributes are divided into sublevels, with each sublevel further divided into its own component metrics (e.g. fault rate, closure time, effort expenditure).[FEN91, 134]

Ejiogu's discussion of software metrics doesn't distinguish between functional and non-functional requirements. Ejiogu, who addresses only "system functionality" [EJI91, 265], discusses metrics of software quality in the scope of Quality Assurance (QA). He writes that "Major corporations are just beginning to set up functional departments with names like metrics groups, metrics council, and process management or software engineering lab." [EJI91, 265] Sound and verifiable measures of evaluation must be defined and adopted, Ejiogu proposes, before Quality Assurance can be addressed; any department involved in quantifying quality must "search for empirical, justifiable, and demonstrably verifiable metric models defined on sound principles and empirical attributes of software behavior." [EJI91, 266]

Ejiogu compares the engineering of quality software with industry counterparts like electronics and automobiles. In the latter two industries, he writes, formal metrics have been in place for years and, as a result, the quality assurance plays a more prominent role in the whole process. [EJI91, 266] He concludes that the so-called "software crisis" might be a result of the relative lack of metrics.

Ejiogu declares that there are three main attributes that determine the ultimate functionality of any system: structuredness, correctness, and reliability. [EJI91, 267] These three attributes seem to have the same level of "unquantifiability" as non-functional requirements. The degree of a software module or software system's structuredness, Ejiogu writes, is "a triad of the factors of refinement, entropy, and functional cohesion." [EJI91, 267]

According to Ejiogu, correctness has to do with "outputting the desired result as determined at requirements definition." [EJI91, 288] Correctness is "built in" during the design phase. Correctness is related to the attribute of reliability in that "correctness in necessary for reliability." [EJI91, 288]

Reliability, Ejiogu writes, is defined as "the degree to which a software system operates for a duration of time without failure." [EJI91, 293] Ejiogu writes that a reliable software system is created by having clear feedback to correct a faulty design during the building of the system.

To illustrate the point that without metrics, software production can be overly complex and even dangerous, Ejiogu presents a module of nine lines of COBOL pseudocode that he purports lacks readability, reliability, and maintainability:

```
MAIN-LINE.
OPEN INPUT DATAFILE.
OPEN INPUT PRNTFILE.
MOVE 'N' TO EOF.
PERFORM READ-FILE UNTIL END-OF-FILE.
DIVIDE TOTAL BY N GIVING MEAN.
WRITE PRNTREC AFTER ADVANCING 2 LINES.
CLOSE DATAFILE PRNTFILE.
STOP RUN.
```

The sample code is unreadable, he asserts, because it lacks comments, although "no amount of commenting or remarking can obliterate the code’s problems." [EJI91, 303]

Ejiogu also finds fault with the code’s content. Several of the process verbs describe several tasks, though to be precise, each should ideally define *one single* task. For example, "PERFORM" involves both an open operation and a write operation. Secondly, the placement of operations is in the wrong format; inequivalent operations are incorrectly interspersed. [EJI91, 304] The clustering together of unrelated ideas, Ejiogu writes, makes code unreadable and extremely difficult to maintain.

Finally, Ejiogu then describes possible corrections to each flaw he identifies and explains why each improvement is critically important in the productions of larger, more complex programs and systems. He suggests grouping related functions together instead of indiscriminately listing them one after the other ; he calls this logical grouping "function cohesion." [EJI91, 317] He also shows how each process verb can be subdivided so that each discrete task is atomic, its name descriptive, and its purpose clear.

CHAPTER 3

NON-FUNCTIONAL REQUIREMENTS CATEGORIZED USING UML

The NFR category hierarchy

Of the one-hundred and sixty one non-functional requirements identified by Chung, et al [CHU99, 160], over ninety percent of them fall easily into eleven broad categories (listed in the next section). The handful of NFRs that cannot be so easily categorized are discussed at the end of this section.

The eleven categories are: performance, cost, security, time, user interface, SRS, end-user, future, intersystem interaction, and system integrity. The categorization of the 161 NFRs will be a useful tool for attacking the problem of selecting representative NFRs to write performance cases for and in the writing of lesson plans to teach them.

Within each separate category, the NFRs can be further categorized easily; some of the NFRs seem to fall naturally together into groups. With the presence of categories and sub-categories, the hierarchical model suggests itself.

At the root of this hierarchy of categories is the abstract notion of the NFR itself, enclosed in a double box. The first tier of the hierarchy of categories represents the eleven categories identified so far. Categories at this level may either be NFRs in their own right or merely abstract categories. The categories are identified by having their names written in a closed rectangle. The categories that are also NFRs are indicated by underlining their name. Thus, all the leaves of this hierarchy must be NFRs and not categories; one cannot have a category that contains no NFRs.

There is one other thing to note about this hierarchy of categories. Some NFRs appear in more than one place: two of them appear in three places and the rest appear in two places. The notation is this: where duplicate entries appear, a number in a circle precedes the name of the NFR.

The justification for this multiple inclusion is that to have allowed an NFR to appear in only one place in the hierarchy, part of the very character of the NFR would have been ignored. For example, the NFR "time performance" is clearly appropriate for both the time and performance categories. To have included it in only one of these two categories would rob it of half its meaning.

For the handful of NFRs that do not fit so easily into this hierarchy of categories, there is a reason for their exclusion. To have included them in the closest-fitting category would dilute the accuracy of the well-fitting NFRs; it would be akin to hammering a square peg into a round hole. For almost all of these excluded NFRs, the difficulty in categorizing them lies in the interpretation of their names. For example, consider "conceptuality." Who is doing the "conceptualizing," i.e., who's "concept" is meant? The system designer? The stakeholder? The client? The end-user? Also consider the NFR "commonality." What is "common"? The system's functions and modules? This system compared with an older system it is replacing? Other, different in-place systems this system is to interact with?

The performance category is one of the "big three" NFRs discussed by Chung, et al in terms of importance (the other two being cost and security). Broadly, performance NFRs describe the speed at which a given system accomplishes its functionality. The performance category can be further subdivided into hardware- and software-related NFRs. Hardware performance concerns the physical parts of the system, including "main memory performance" and "secondary storage performance." The software performance subcategory is the larger of the two, including the performance of buffer space, code space, data space, and throughput, among others.

NFRs in the performance category should not be confused with ones in the time category. Both deal with temporal issues, but they are not the same. Performance concerns the system itself meeting functionality criteria, whereas the time category includes NFRs that deal with issue outside the system's functionality, such as "prototyping time," "planning time," and "development time." These NFRs describe the development of the system in terms of a larger software engineering environment.

The cost category concerns money, both in terms of bringing the system to a finished state and in the cost maintaining the system during its lifetime. The cost category can therefore be divided into two subcategories: before deployment and after deployment. The NFRs that fall into the first category relate to the costs of the requirements, design, coding, debugging, and testing phases of development. Some NFRs in this subcategory include "development cost," "domain analysis cost," and "planning cost." Ideally, these costs should only be incurred once, before final delivery of the system. The second subcategory, "after deployment," contains NFRs that represent the costs associated with the system after it has been delivered to the customer. Some of these NFRs include "maintenance cost," "operating cost," "reengineering cost," and "retirement cost." This final NFR may be linked to costs involved when the early phases of a replacement system are being considered.

The security category is related to the system integrity category, but it is far smaller. Security NFRs may be thought of as concerning two problem areas: malicious outsiders seeking to corrupt or steal data, and modules internal to the system that may inadvertently put the system into a corrupted or invalid state. "Susceptibility" in an NFR that may be used to measure both internal and external threats to the system's integrity. "Robustness" describes the strength of the system's ability to withstand malicious security breaches or inadvertent changes by internal procedures. "Confidentiality" handles the granting of levels of access to the various classes of users of the system.

The structure of one NFR definition and example

The definitions of each of the one hundred and sixty one NFRs will be provided. The definition of the first NFR, accessibility, is given below, along with a brief explanation of its location in the UML diagram. The other one hundred and sixty NFRs, written in the same format of definition and discussion, is given in Appendix A. Some of the NFRs may be interpreted as synonymous with others in the list; this is clarified in said closely-related requirements.

Using the University Library Materials Tracking System (ULMTS) example upon which the in-class NFR lesson plans in chapter four are based, an example of one of the 161 NFRs is also given below. Examples of the other one hundred and sixty NFRs are shown following each definition. For brevity's sake, each item is limited to one or two sentences. The main purpose of each of the examples is to discern it from other similarly-named NFRs within the same top-level category and to clarify its relation with other NFRs in each of the eleven top-level categories.

accessibility – The accessibility requirement describes the degree of access to a system. Without an acceptable amount of access, any given system would be worthless. Both human users and other systems may access a system. The degree of accessibility may be affected by network traffic, hardware failures, and bugs. The notion of access is also closely related to security;

system users may be granted different levels of access based on differing levels of authorization. To allow access to information items in a system, it may be necessary to validate that the user or external system has the appropriate level of access [CHU99, 155]. An example of this validation is an audit trail, "which is input to the access authorization decision, which may be needed for the enforcement of separation of duties"[CLA87, 98].

Accessibility appears in three places in the UML hierarchy: (1) the SystemIntegrity category, Security sub-category, (2) the End User category, and (3) the IntersystemInteraction category. The rationale for accessibility being included in the Security category is described above. Accessibility is included in the EndUser category in situations where a human user is attempting to access a system in the context of being constrained by network traffic, hardware limitations, and the like. Accessibility is included in the IntersystemInteraction category for the same reasons listed in the End User category, except here the entity accessing a system is itself another system.

Example: An administrative user, after successfully completing the login process, shall be able to access the ULMTS system on their first access attempt if their administrative account is valid.

CHAPTER 4

AN IN-CLASS LESSON PLAN FOR TEACHING NFRs

To give students in an undergraduate software engineering class a more thorough and interactive exposure to NFRs, this report recommends that students apply a hands-on technique for learning, using Cockburn's use cases. Specifically, a software engineering class will, after being introduced to both NFRs and sample use case diagrams (perhaps following a lesson on functional requirements analysis), be divided up into groups of no more than four people and each student be given a blank use case form. Next, the students will be given a sample completed use case form based on some real-world computing environment activity. In the lesson plan below, a university library materials tracking system is proposed.

The groups of students will then be given fifteen minutes to complete the use case form for one specific subtask the system will support. While the groups are discussing each scenario and filling in the blank fields of their use cases, the instructor may walk around the classroom, answering questions and ensuring each group stays on task.

At the end of the allotted time, each group will choose a spokesperson, and each group's spokesperson will read their solution to the specific scenario they documented. After each group's use case is presented to the class, the instructor may make some general remarks or recommendations about effective use case strategies that students may have overlooked.

By having students interactively document NFRs using use cases, rather than simply be lectured on the subject and shown an example on an overhead projector, hopefully the learning outcomes may be more efficiently taught. When students are asked to think rather than merely listening, the subject material becomes more meaningful and more real than if they were simply presented with plain content.

In any software engineering course, sufficient subject material exists to fill more time than is actually allotted. The above learning activities, depending on the size of the class, represent about thirty minutes in 45 hours of in-class time. The following lesson plan includes a model system (a university library materials tracking system) and three real world scenarios. These three scenarios each represent a requirement in three of the eleven top-level NFR categories identified in the UML hierarchy: Performance, EndUser, and SystemIntegrity. Since each top-level category contains many leaf-node level NFRs, a requirement that represents the broad category is chosen: from the Performance category the time performance requirement is used, from the EndUser category the customizability requirement is used, and from the SystemIntegrity category the security requirement is used. The three NFRs each presuppose some "exterior" resource that a typical system user will not employ. For example, the time performance requirement would use some timing software to measure how long a database lookup takes. The EndUser requirement would use a document describing all options available to a system user to make changes to the system interface and saved in a unique preferences file that is loaded during login. The security requirement would need a human user playing the role of an unauthorized, malicious marauder attempting to access the system.

To quantitatively measure the efficacy of the four learning units, a pre-test and a post-test were written. Each student in the test class was given both tests before and after, respectively, the lesson was administered.

Pre-test

The pre-test was written with two assumptions: the students to whom it will be administered will have already been introduced to the general activities of the requirements phase, and the students have not been taught the distinction between functional and non-functional requirements.

The pre-test was written in the following format: six multiple choice questions, each with three or four incorrect "distractors," purposely incorrect terms taken from other software engineering areas.

This pre-test was administered to one of the undergraduate software engineering classes at East Tennessee State University in January, 2003. A detailed analysis of the raw data is presented in Chapter 4.

Question	
1.	A non-functional requirement is: (select one) a) A document that lists which parts of a software system are not yet working b) A requirement that places constraints on a system rather than explicitly describing what it should do c) A requirement that software developers and customers use as a guide to prevent software from not working correctly
2.	An example of a non-functional requirement is: (select one) a) Assembly language b) A network c) Performance
3.	An example of a non-functional requirement is: (select one) a) Security b) Documentation c) Malfunctions
4.	An example of a non-functional requirement is: (select one) a) High data volume b) Servers and Clients c) Customizability
5.	Non-functional requirements are important because: (select one) a) They allow programmers to fix bugs in the code. b) They allow the customer to easily change the software as it is being produced. c) They describe expectations of what the software should do.
6.	A useful tool to describe non-functional requirements is: (select one) a) Data Flow Diagram b) Use Case c) Sparse Matrix d) Hash Table e) All of the above

Table 3: Pre-test

<u>Three learning units</u>

A software system is to be created for a university library. The central database will contain records for books and library patrons. The records for books will include fields for author, title, publisher, Dewey Decimal ID, cost, and year published. The record for patrons will include fields for name, address, phone number, SSN, student/faculty status, and items currently charged. Authorized system users will be able to search the database on any of these fields and add or remove library items.

USE CASE	Search time for book by author.	
Goal	The average search time to look up whether or not a book is checked out given an author's should be less than one second	
Level	Primary	
Preconditions	List of book records is sorted by author name	
Actor	User	
Type and Subtype	Performance, time performance	
Action Steps	**Step**	**Action**
	1	System prompts for book author
	2	User enters author name
	3	System searches for all books by author and displays charged status
EXTENSIONS AND EXCEPTIONS	**Step**	**Branching Action**
	3a	Author not found – display error message
Use Case Links		Search for book by author
Design Links		class BookList
Rationale Links		...

Table 4: Sample use case given to the students

PERFORMANCE

Using the above Use Case as a guide, write a Use Case for the following NFR:

USE CASE #1	Write an estimate that determines the time to enter a new library item	
Goal	A user will enter a new item in the library database	
Level	Administration; user	
Preconditions	Item does not exist in the database	
Actor	Library employee	
Type and Subtype	Performance, SoftwarePerformance, SoftwareTimePerformance	
Action Steps	**Step**	**Action**
	1	System prompts for user to input unique ID of new item
	2	User enters new item ID
	3	System searches database for ID to determine if duplication
	4	System adds new record
EXTENSIONS AND EXCEPTIONS	**Step**	**Branching Action**
	3a	System finds duplicate item ID, displays error message
Use Case Links		
Design Links		Performance NFR
Rationale Links		

Table 5: Performance Use Case

END USER

Using the above Use Case as a guide, write a Use Case for the following NFR:

USE CASE #2	User changes user interface settings to personalize	
Goal	An authorized user will attempt to customize the system's interface to make interaction with the system more aesthetically pleasing or easier to use	
Level	Primary	
Preconditions	None	
Actor	User; library employee	
Type and Subtype	UserInterface, Objective	
Action Steps	**Step**	**Action**
	1	User selects "Customize" menu option
	2	System displays list of interface elements that can be modified
	3	User selects one element
	4	System displays all possible changes that can be made to that element
	5	User selects one change and hits the "Apply" button
	6	System records change
	7	System writes change to a persistent user preference file loaded after user logs in
	8	System applies change to current user's session
EXTENSIONS AND EXCEPTIONS	**Step**	**Branching Action**
	7a	Error in writing to user preference file—system displays error message
Use Case Links		
Design Links		EndUser NFR, Objective subcategory
Rationale Links		

Table 6: End-user Use Case

SYSTEM INTEGRITY

Using the above Use Case as a guide, write a Use Case for the following NFR:

USE CASE #3	System denies access to unauthorized user	
Goal	A human actor, playing the role of unauthorized intruder, will attempt to gain access by entering a valid user ID but making three guesses about the user's associated password	
Level	Primary	
Preconditions	Valid user account exists	
Actor	User; unauthorized user	
Type and Subtype	SystemIntegrity, Security	
Action Steps	**Step**	**Action**
	1	System prompts for username
	2	User enters valid username
	3	System prompts for password
	4	User enters invalid password
	5	System displays error message
	6	System prompts for password
	7	User enters invalid password
	8	System displays error message
	9	System prompts for password
	10	User enters invalid password
	11	System displays error message
	12	System ends login process
	13	System photographs user, sends picture to FBI, flashes red emergency lights, blares ear-piercing sirens and alarms, emits cloud of noxious knock-out gas
EXTENSIONS AND EXCEPTIONS	**Step**	**Branching Action**
Use Case Links		
Design Links		SystemIntegrity NFR, Security
Rationale Links		

Table 7: System Integrity Use Case

Post-test

To gauge the effectiveness of using the three in-class learning units, a metric to gather hard data was necessary. It was decided that a post-test would fill this need. The post-test was written in a similar format as the pre-test: six multiple choice questions, each with three or four distractors. To prevent any data corruption stemming from any rote memorization on the part of the test subjects (students), the distractors were changed to be different from the pre-test, and the position of the correct answer in each question was randomly changed.

Question	
1.	A non-functional requirement is: (select one) a) A requirement that describes the bugs and errors in previous versions of the software b) A requirement that places constraints on a system rather than explicitly describing what it should do c) A document that lists which parts of a software system that are not yet working
2.	An example of a non-functional requirement is: (select one) a) Performance b) Error log c) Programming language
3.	An example of a non-functional requirement is: (select one) a) Client/server b) Security c) Cache
4.	An example of a non-functional requirement is: (select one) a) A CPU b) Data warehousing c) Customizability
5.	Non-functional requirements are important because: (select one) a) They describe expectations of what the software should do b) They allow the customer to easily change the software as it is being produced c) They let designers easily find errors or duplicate logic in the code
6.	A useful tool to describe non-functional requirements is: (select one) a) Header file b) Graphical user interface c) Data structure d) Use Case e) All of the above

Table 8: Post-test

Analysis of pre-test and post-test data

The following data attempt to rate the effectiveness of the three in-class learning modules.

Question	Pre correct	Pre incorrect	Post correct	Post incorrect
1	11	4	14	1
2	10	5	13	2
3	5	10	15	0
4	10	5	14	1
5	8	7	12	3
6	7	8	12	3

Table 9: Raw class grades

Pre-test result set: { 100, 83, 83, 83, 83, 67, 50, 50, 50, 50, 50, 33, 33, 17, 17}

Pre-test class average: 849/15 = 56.6

Post-test result set: { 100, 100, 100, 100, 100, 100, 100, 100, 83, 83, 83, 83, 67, 67, 67 }

Post-test class average: 1333/15 = 88.87

Question	Number correct increase	% increase
1	3	20
2	3	20
3	10	66.7
4	4	26.7
5	4	26.7
6	5	33.3

Table 10: Grade Increase Percentages

Overall class average grade increase: 32.27%

The preceding data suggest that the goal of the in-class learning units was achieved: to give students in an undergraduate software engineering curriculum a better understanding of the importance of writing correct NFRs and their importance to building software systems as a whole.

Subjects rate the in-class learning units

Though not required to draw conclusions about the quality of the in-class lessons, it was suggested to gauge the test subjects overall satisfaction with the lesson and its presentation. To this end, a questionnaire using the Likert scale was written and administered to the test subjects the week after the in-class lesson was given, as shown in table 11:

(instructions) Circle the appropriate letters, abbreviated as:

AS	Agree Strongly
A	Agree
N	Neutral
D	Disagree
DS	Disagree Strongly

1.	Non-functional requirements are a useful topic in software engineering.	AS A N D DS
2.	Overall, the quality of the presentation was good.	AS A N D DS
3.	Filling out the example use case was helpful.	AS A N D DS
4.	I would have learned the subject better if another teaching method were used.	AS A N D DS
5.	The presenter demonstrated knowledge of his topic.	AS A N D DS
6.	The in-class examples were written well.	AS A N D DS
7.	Overall, the presentation could have been improved.	AS A N D DS
8.	I expect to one day write non-functional requirements in my career.	AS A N D DS

Table 11: Exit questionnaire

To better view the results of this questionnaire, point values were assigned to each of the five choices:

AS	Agree Strongly	5 points
A	Agree	4 points
N	Neutral	3 points
D	Disagree	2 points
DS	Disagree Strongly	1 point

It should be kept in mind that since the questions are written with positive and negative expected results listed in different orders, the five-point questions are not necessarily more desirable than the lesser point values. The numbers only serve to give a more easily understood picture of the class's opinions as a whole. The results are listed in table 12:

Question	Average
1.	4.6875
2.	4.125
3.	4.125
4.	2.6875
5.	4.5
6.	3.9375
7.	3.5625
8.	3.625

Table 12: Exit Questionnaire Results

1. Non-functional requirements are a useful topic in software engineering.

The fact the class average is nearly between "agree strongly" and "agree" suggests the test subjects recognize the usefulness of NFRs.

2. Overall, the quality of the presentation was good.

The class average suggests the test subjects agreed the presentation was good.

3. Filling out the example use case was helpful.

The class average suggests the test subjects agreed filling out the use case was helpful.

4. I would have learned the subject better if another teaching method were used.

The class average falls nearly halfway between “neutral” and “disagree.”

5. The presenter demonstrated knowledge of his topic.

The class average falls exactly halfway between “strongly agree” and “agree.” This writer is flattered.

6. The in-class examples were written well.

The class average suggests the test subjects agreed the examples were written well.

7. Overall, the presentation could have been improved.

The class average falls nearly halfway between “neutral” and “agree.”

8. I expect to one day write non-functional requirements in my career.

The class average falls almost halfway between “neutral” and “agree,” suggesting the test subjects aren’t as strongly passionate they will one day deal with NFRs. Three of the respondents to the questionnaire circled “strongly disagree,” suggesting they have already decided what their career will be and they know it won’t involve NFRs.

BIBLIOGRAPHY

BIBLIOGRAPHY

[ABE98] Abel, Peter, *IBM PC Assembly Language and Programming* (1998), Prentice Hall, Upper Saddle River, NJ.

[AMB02] Ambler, Scott W., "What is Agile Modeling?" http://www.agilemodeling.com (accessed June 14, 2002).

[BAR01] Barrett, Martin, "Putting Non-Functional Requirements to Good Use", publication pending, (2001).

[BOE02] Boehm, Barry, "Get Ready for Agile Methods, with Care," *IEEE Computer* www.computer.org/computer/, (accessed June 14, 2002).

[BOE84] Boehm, Barry, *V & V Software Requirements* (1984), IEEE Computer Society Press, Los Alamitos, CA.

[BOW85] Bowen, T., Wigle, G., and Tsai, J. T., *Specification of Software Quality Attributes*, prepared by Boeing Aerospace Company for Rome Air Development Center, Griffiss AFB, NY, 1985.

[BRO86] Brooks, Frederick, "No Silver Bullet", *Information Processing* (1986), Elseveir Publishers, Holland.

[CHA86] Charette, Robert, *Software Engineering Environments* (1986), McGraw-Hill, New York, NY.

[CHU99] Chung, L., Nixon, B., Yu, E., and Mylopoulos, J., *Non-Functional Requirements in Software Engineering* (1999), Kluwer Academic Publishers, Boston, MA.

[CLA87] Clark, D. D., and Wilson, D. R., "A Comparison of Commercial and Military Computer Security Policies," *Proceedings of the IEEE Symposium on Security and Privacy*, 1987, pp. 184-194.

[COC01] Cockburn, Alistair, *Writing Effective Use Cases* (2001), Addison-Wesley, Boston, MA.

[DAV93] Davis, Alan, *Software Requirements* (1993), Prentice-Hall Inc., Englewood Cliffs, NJ.

[EJI91] Ejiogu, Lem, *Software Engineering With Formal Metrics* (1991), QED Technical Publishing Group, Boston, MA.

[FEN91] Fenton, Norman, *Software Metrics: A Rigorous Approach*(1991),Chapman and Hall, London.

[FIR93] Firesmith, D. G., *Object-Oriented Requirements Analysis and Logical Design* (1993), John Wiley and Sons, New York, NY.

[FLO97] Floyd, Thomas L., *Digital Fundamentals* (1997), Prentice-Hall, Upper Saddle River, NJ.

[FOW99] Fowler, M., Scott, K., *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (1999), Addison-Wesley, Reading, MA.

[IEE93] Institute for Electrical and Electronics Engineers, *IEEE Guide to Software Requirements Specifications*, Standard 830-1993, New York: IEEE Computer Society Press, 1993.

[KIT95] Kit, Edward, *Software Testing in the Real World* (1995), ACM Press, London.

[KOT98] Kotonya, Gerald, and Sommerville, Ian, *Requirements Engineering: Processes and Techniques* (1998), John Wiley and Sons, New York, NY.

[MAL01] Malan, Ruth, and Bredemeyer, Dana, "Defining Non-Functional Requirements", 2001, on http:/www.bredemeyer.com , accessed June, 2002.

[MCC77] McCall, J. A., et al, "Concepts and Definitions of Software Quality" (1977), *Factors in Software Quality*, NTIS, vol. 1.

[MEY90] Meyer, Bertrand, *Object-Oriented Software Construction* (1988), Prentice-Hall, Upper Saddle River, NJ.

[PAG00] Page-Jones, Meilir, *Fundamentals of Object-Oriented Design in UML* (2000), Addison-Wesley, New York, NY.

[PFL87] Pfleeger, Shari, *Software Engineering* (1987), Macmillan Publishing Co., New York, NY.

[PHI00] Phillips, D., *The Software Project Manager's Handbook* (2000), IEEE Computer Society Press, Los Alamitos, CA.

[PRE01] Pressman, Roger, *Software Engineering: A Practitioner's Approach* (2001), McGraw-Hill, New York, NY.

[PUT97] Putnam, L., and Myers, W., *Industrial Strength Software: Effective Management Using Measurement* (1997), IEEE Computer Society Press, Los Alamitos, CA.

[SAG90] Sage, Andrew, and Palmer, James, *Software Systems Engineering* (1990), John Wiley and Sons, New York, NY.

[SIL99] Silberschatz, Avi, and Galvin, Peter, *Operating System Concepts* (1999), John Wiley and Sons, New York, NY.

[STE74] Stevens, W., Myers, G., and Constantine, L., "Structured Design," *IBM Systems Journal*, vol. 13, 1974.

[TAJ81] Tajima, D., and Matsubara, T., "The Computer Software Industry in Japan," *Computer*, May 1981.

[TAN99] Tanenbaum, Andrew, *Structured Computer Organization* (1999), Prentice Hall, Upper Saddle River, NJ.

[TAN02] Tanenbaum, Andrew, and van Steen, Martin, *Distributed Systems: Principles and Paradigms* (2002), Prentice Hall, Upper Saddle River, NJ.

[THA97] Thayer, Richard and Dorfman, Merlin, (editors), *Software Requirements Engineering* (1997), IEEE Computer Society Press, Los Alamitos, CA.

[YOU79] Yourdon, E., and Constantine, L., *Structured Design* (1979), Prentice-Hall, Englewood Cliffs, NJ.

APPENDICES

APPENDIX A

DEFINITIONS AND EXAMPLES OF 160 NON-FUNCTIONAL REQUIREMENTS

accountability – Accountability describes the amount of responsibility the developer has concerning the correctness of the work done according to the SRS (Software Requirements Specification). The SRS is "the final word" when questions of responsibility arise between the developers and the customers. It acts as a final contract, explicitly enumerating every feature the system to be built is to possess. A synonym for accountability is "answerable."

Accountability implies that a developer is accountable for work he or she performs. Specifically, it may mean a developer is held to some error limit. For example, the SRS sometimes states that the software shall have no fewer than x number of bugs per thousand lines of code. If this number is exceeded, a developer may be held accountable. For this reason, accountability falls most naturally into the SRS category in the NFR UML hierarchy.
Example: The ULMTS system developer(s) shall be held responsible if the number of acceptable errors as specified throughout the SRS (Software Requirements Specification) is exceeded.

accuracy – Accuracy is "the precision of computations and control" [MCC77, 182]. Accuracy is a broad term, and it may be applied to many facets of a software system. In Chung, et al's classification of NFRs, accuracy is one of the three main divisions (security and performance being the other two). They define accuracy as being an "inherent property of information;" just as a physical object may have size and weight, an information item has accuracy. [CHU99, 159].

Accuracy is a key component of any system. If a user detects the slightest degree of inaccuracy in one place, it may call into question the veracity of any other part of the software, and where doubt arises, a user may lose trust in the whole system. Though arguably, accuracy could have been applied to any of the categories in the NFR hierarchy, it falls most naturally under the SystemIntegrity category, since it is the very integrity of the system that is compromised by inaccurate data.
Example: The ULMTS shall make no more than one transaction error per 5000 materials entry transactions.

adaptability – The adaptability requirement is defined by "the rate at which the software solution can adapt to a new requirement" [DAV93, 211]. Adaptability also refers to the degree to which a system may be changed based on a pre-existing system or an unalterable constraint. Using this sense, adaptability implies that a given system is not completely adequate to be put in place; modifications must be made to it to accommodate its implementation. The adaptability requirement differs from modifiability in that adjustments to the system must be made before it is thoroughly reconciled with everything that is already in place, whereas with the modifiability requirement, a system has been deployed before modifications are made to it.

As it relates to software's adaptation to a new requirement, the adaptability requirement may be found in the SRS category of the UML hierarchy. As it relates to alterations made to a system based on pre-existing constraints, the adaptability requirement may be found in the Future category, FutureChange sub-category of the UML hierarchy.

Example : Changes in user needs shall allow changes to the functionality of the ULMTS possible.

additivity – Additivity refers to the ability of a given system to continue to function when newer system elements are added to the environment. Additivity is a difficult requirement to satisfy, since time is unidirectional and the future cannot be so easily predicted. There are two models of integrating system elements through time. In one model, a system is built with the future in mind, and all possible additions to it are anticipated and handled accordingly. In the second model, components that are to be added to existing systems are created with the past in mind; they are written so that they may be added to systems that have previously been built and continue to exist.

The additivity requirement falls most naturally into the Future category, FutureChange sub-category of the UML hierarchy, since the issue is one of compatible additions being made to a system throughout its lifetime.
Example: The ULMTS shall to continue to function correctly when new classes are added to the system.

adjustability – The adjustability requirement is almost synonymous with modifiability. There is a very slight semantic difference: adjusting an in-place system implies that there were problems or incompatibilities to begin with (perhaps due to inadequate specifications), and after the system was implemented, these shortcomings presented themselves, necessitating "tweaking" to create correct functionality. Modifiability implies the system was completely correct to begin with when it was implemented, but external changes in the environment are what dictate changes in the system.

The adjustability requirement falls most naturally into the Future category, FutureChange sub-category of the UML hierarchy since the issue is one of changes being made to a system after deployment and throughout its lifetime.
Example: Changes in user needs shall allow changes to the functionality of the ULMTS possible.

affordability – The affordability requirement simply refers to the capacity of the customer to pay for the system the developer created. For this reason, affordability belongs in the Cost category, under the sub-category "CostPredeployment": it is standard practice for the customer to pay for work products before they are delivered. Unlike other types of cost, such as maintenance cost, which extends over the lifetime of a system, affordability concerns just a one-time consideration: whether it is feasible to allow system development on a project to begin based on an agreed-upon, strict dollar amount.
Example: ULMTS system development shall proceed with the understanding the University's budget has allotted $875,000 for the total project (development, installation, and maintenance).

agility – The agility requirement relates to a system's performance; it can be found in the Performance category, SoftwarePerformance sub-category of the UML hierarchy. With the recent rise of agile programming and agile modeling, developers now have a new paradigm to work within. Four basic tenets of agile programming are:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation

- Customer collaboration over contract negotiation
- Responding to change over following a plan

[AMB02, 298].

But if these define agile programming, what does the agility requirement describe? According to the second tenet above, it is more important that agile software run than have "comprehensive documentation." One simple strategy for rendering documentation redundant is to make a program's variable names self-explanatory.
Another is to make the structure of system components (such as classes) uniform so the amount of duplicated documentation may be reduced.
Example: The ULMTS shall be agile in the sense that the structure of system components (classes, modules) will be uniform; in so doing, the amount of duplicated documentation shall be reduced.

auditability – The auditability requirement describes "the ease with which conformance to standards can be checked" [MCC77, 54]. Customers do not have a part in auditing; it is done throughout the project's development process by the developers and, in the case of independent audits, by an outside agency or team that is separate from the organization responsible for the project to "assess compliance with requirements, specifications, baselines, standards, procedures, instructions, codes, and contractual and licensing requirements" [THA97, 182].

The counterpart to auditing is acceptance testing, which is done by customers to insure the system meets the requirements as specified in the SRS. Since auditing is checking the conformance of the system to its specifications, and since it is done by the developers, the auditability NFR belongs in the SRS category of the UML hierarchy.
Example: ULMTS system administrators shall have the ability to perform system audits at any point, and the results of these audits shall be recorded in log files.

availability – Pressman defines availability as the "probability that a program is operating according to requirements at a given point in time, and is defined as

Availability = [MTTF / (MTTF + MTTR)] * 100%"

where MMTF is mean-time-to-failure and MTTR is mean-time-to-repair [PRE01, 408].

The availability requirement is similar to accessibility, since the degree of availability may be affected by network traffic, hardware failures, and bugs. Unlike accessibility, availability has no security implications; it is assumed that the system be as available as possible to any user without checking any permissions or access levels. Since this requirement concerns access to a system after it has been deployed, it can be found in the Future category, FutureAccess sub-category of the UML hierarchy.
Example: The ULMTS shall be available to a minimum of 200 concurrent users (including users and administrators).

buffer space performance – A buffer is a "memory area that stores data while they are transferred between two devices or between a device and an application" [SIL99, 117]. Buffers are used to even the mismatch of speeds of devices when doing input and output.

Although the term "buffer" is also applied to a "circuit that prevents loading of an input or output" [FLO97, 285], the buffer space performance requirement falls into the SoftwareSpacePerformance sub-category of the Performance category because of the more general sense of a buffer representing an area of memory rather than a particular circuit.
Example: The buffer space shall not exhibit noticeably detrimental behavior after buffer overflows.

capability – "Capabilities are the fundamental requirements of the system and represent the features or functions of the system needed or desired by the customer" [IEE93, 119].
Because this requirement suggests simply all the features a system is required to have, it can be found in the SRS category.

Capability is also used to describe a software engineering task that an organization may or may not possess. The Capability Maturity Model (CMM) was developed by the Software Engineering Institute to accurately qualify an organization's current state of process maturity [PRE01, 381] . Used in this sense, capability refers to the developers themselves, so it is not included in this hierarchy of non-functional requirements.
Example: The ULMTS shall possess all features and functions of the system described in the SRS.

capacity – The term "capacity" describes the maximum amount or number that can be contained. Like the capability requirement, a system's "capacity" is vague. It may refer to a number of measures: the size of memory, the size of secondary storage, the bandwidth of a network connection, the maximum number of users that may access the system simultaneously, et cetera. Because these quantities should be determined by the customer and included in the SRS for development, the capacity requirement may be found in the SRS category of the UML hierarchy.
Example: The ULMTS shall support the following capacity requirements:
*total size of main memory – 40 gigabytes
*total size of secondary storage – 120 gigabytes
*maximum number of users that may access the system simultaneously – 200
*total bandwidth: 1024 baud

clarity – The clarity requirement describes the degree to which a system and all its components are unambiguous. A system is unambiguous "if and only if every requirement in the SRS has only one possible interpretation" [IEE93, 71].

The clarity requirement can be found in the UserInterface category of the UML hierarchy, in the Learnabilty sub-category and ContinuedUse sub-category. As it relates to the end user, clarity describes both how easily a user can learn a software system's interface, and how easily a user can continue to understand the interface after it is learned (see also the learnability and trainability NFRs).
Example: The ULMTS system documentation shall be legible and understandable by a person of the eighth-grade reading level.

code-space performance –In IBM PC assembly language, code space is referred to as the *code segment* and is addressed via the CS register [ABE98, 177]. Maximum optimization of a system may be achieved by writing all code in machine language, but due to extreme difficulty for humans to manipulate machine instructions, this might be an unrealistic goal. Performance tradeoffs are made by writing code in assembly language and high-level languages; performance may suffer due to a larger code space but debugging the code is much easier and more natural since the logic and instructions are human-readable.

Because this requirement concerns the performance of code in a system's hardware (memory areas), the code-space performance NFR may be found in the HardwarePerformance category of the UML hierarchy, SpacePerformance subcategory.
Example: It shall take no longer than 70 milliseconds to instantiate objects from any class in the ULMTS architecture.

cohesiveness – The cohesiveness requirement describes *cohesion*, "the degree to which each module in a system does one task and does it well" [YOU79, 113]. Cohesion also refers to "the relatedness of elements that constitute an encapsulated unit" [PAG00, 204].

Constantine writes that cohesion may be viewed as a spectrum, with high cohesion being most desirable and low cohesion being least desirable and to be avoided, where possible [YOU79, 113]; his scale of cohesion divides cohesiveness into seven categories. They are, from the highest to lowest degree of cohesiveness:

*Functional cohesiveness – a component's elements contribute to one and only one task.

*Structural cohesiveness – a component's elements are involved in tasks where output from one task serves as input to the next.

*Communicational cohesiveness – a component's elements concentrate on one area of a data structure.

*Procedural cohesiveness – a component's processing elements are related and must be executed in a specific order.

*Temporal cohesiveness – a component's elements are related by the fact that all must be executed within the same span of time.

*Logical cohesiveness – a component's elements perform a set of tasks whose functions are related logically.

*Coincidental cohesiveness – a component's elements perform a set of tasks that are loosely related to each other, if at all [YOU79, 88].

Since this requirement relates to the inter-relatedness of a system's components, the cohesiveness NFR can be found in the Design category of the UML hierarchy.
Example: The modules of the ULMTS system shall exhibit a high degree of cohesion, as described by Constantine.

commonality – The commonality requirement refers to "the use of standard data structures and types throughout the program" [MCC77, 191]. Commonality may also refer to not merely the data primitives and familiar data structures, but also to the system components (such as structs and classes) which they comprise. The reuse of previously written system components and modules, either implemented in-house or bought off the shelf, help promote commonality; for this reason, the reusability requirement is closely related to this NFR. (see reusability)

The counterpart to the commonality NFR is diversity, which describes the degree of difference between a system's components and modules (see also diversity). Although commonality is increased when system components are reused, diversity is not necessarily detrimental. A system's components may need to be diverse if there is functionality that existing components cannot accomplish. Diversity, like commonality, may also refer to data type and data structures throughout the program.

Because this NFR refers to the components that comprise a given software system, the commonality requirement can be found in the Design category of the UML hierarchy.
Example: The classes in the ULMTS design shall use the standard data structures and types of the STL (Standard Template Library) throughout the program.

communication cost – The communication cost requirement describes the monetary amount required for the system developers and customers to exchange information. For a system to be successfully built, the developer and customer should communicate continuously throughout the development process. The developers may elicit requirements from the customers, but they should also ask further questions for absolute clarification of the specifications. Prototypes (see the prototyping time and cost requirements) are a good first step to ensure that the developers are producing a product that accurately reflects the customers' expectations. Acceptance testing allows the customers an opportunity to "test-drive" the completed system to make sure it has all the necessary functionality.

Because this requirement concerns the cost of interaction between developer and customer before a system is delivered, the communication cost NFR may be found in the Cost category of the UML hierarchy, in the CostPredeployment sub-category.
Example: It shall cost no more than $350 to conduct one meeting where ULMTS developers and representatives of the University exchange information.

communication time – The communication time requirement describes the amount of time necessary for the system developers and customers to exchange information about the software that is to be produced (for a definition of communication, see the communication cost requirement). Two-way communication between the two sides is crucial if a software system that fully satisfies the customers' requirements and specifications is to be built.

Because this requirement concerns the length of time spent by both developer and customer exchanging information about a system to be built, the communication time NFR can be found in two places in the UML hierarchy: in the Time category, in the TimePredeployment sub-category, in both the Developer and Customer sub-subcategories.
Example: It shall take no more than one business day for ULMTS developers and representatives of the University to respond to a question from the other.

compatibility – The compatibility requirement describes "the ability of two or more systems to exchange information" [THA97, 182]. When a system is being deployed to replace an earlier version of that system, it is imperative that it be compatible with everything that it is replacing is compatible with. Compatibility is especially important in distributed systems and client/server architectures where other systems may be running on other platforms.

Compatibility is nearly synonymous with inter-operability, a requirement that describes "the relative effort to couple the software of one system to the software of another system" [THA97, 183]. (see also the inter-operability NFR)

Because the compatibility requirement describes the interaction (information exchange) between two or more systems, it can be found in the IntersystemInteraction category of the UML hierarchy.

Example: The ULMTS shall be fully backwards compatible with PLMTS (the University's previous library materials tracking system).

completeness – The completeness requirement describes "the degree to which full implementation of required functionality has been achieved" [MCC77, 137]. Such required functionality is usually spelled out in the SRS. The authors of the IEEE standard for the SRS list four criteria to determine whether the SRS satisfies the completeness requirement:

*Everything that the software is supposed to do is included in the SRS.

*Responses of the software to all realizable classes of input data in all realizable classes of situations is included.

*All pages are numbered; all figures and tables are numbered, named, and referenced; all terms are defined; all units of measure are provided; and all referenced material is present.

*The SRS contains no sections marked "To be determined" [IEE93, 154].

As it relates to the Software Requirements Specification, the completeness requirement may be found in the SRS category of the UML hierarchy.
Example: ULMTS will have full implementation of the required functionality as is explicitly described in the SRS.

component integration cost – The component integration cost requirement describes the financial amount of combining and testing the modules that comprise a system. In an object-oriented language these components might be structs and classes, or, in the case of C++, these components might be a number of .h and .cpp files.

Integration testing is "the process of combining and testing multiple components together; the primary objective of integration testing is to discover errors in the interfaces between the components" [KIT95, 254]. Generally, the more thorough the testing, the higher the cost to satisfy this requirement.

Because this requirement concerns the cost of combining components, the component integration cost NFR may be found in the Cost category of the UML hierarchy, in the CostPredeployment sub-category.
Example: The integration of the ULMTS components (classes) shall not cost more than $7,500 (note: the total cost of ULMTS to the University shall not exceed $875,000).

component integration time – The component integration time requirement describes the amount of time required to combine and test the modules that comprise a system
(for a definition of component integration, see the component integration cost requirement).

There are a number of integration testing methods that may determine how much time this requirement describes. In one method, "big bang testing," all components are combined at once and the integrated result is tested" [KIT95, 193]. At first glance this strategy might seem to take the least amount of time, but because debugging is difficult (since an error can be associated with any component) big bang testing may require more time than the developers may have. Another strategy, "bottom-up integration," uses incremental testing of components and additivity testing. Though it takes longer to test each component separately and then add it to the existing system, it is easier to debug since the exact location of each error is clear; this may allow for a shorter component integration time.

Because this requirement concerns the length of time required to combine components, the component integration time NFR may be found in the Time category of the UML hierarchy, in the TimePredeployment sub-category, in the Developer sub-subcategory.
Example: It shall take no longer than 30 days for system developers to successfully integrate all of the components (classes) of ULMTS.

composability – This requirement is the opposite of decomposability. It describes "the degree to which a design method ensures that program components (modules), once designed and built, can be reused to create other systems" [MEY90, 88]. (see also the reusability NFR)

Since this requirement concerns the construction of multiple systems, it more affects the work of system developers than the requirements and specifications of customers and end-users of one system. For this reason, the composability requirement can be found in two places: in the Cost category, CostPredeployment subcategory, and the Time category, TimePredeployment subcategory, Developer sub-sub-category. If developers write system components that are reusable for development of other systems, both development time and development cost may reduced.
Example: The ULMTS shall be composed of classes; these class interfaces are to be implemented using C++.

comprehensibility – The comprehensibility requirement describes "the ease with which a program component can be understood without reference to other information or other modules" [PRE01, 262]. A requirement is comprehensible if its readers, which include "customers, users, project managers, software developers, and testers" can easily understand the requirement's meaning with a minimum of explanation [DAV93, 186]. If users and customers do not fully understand a requirement in the SRS, they may erroneously approve it, leaving the developer to introduce a flaw into the system. If developers and testers do not fully understand a requirement in the SRS, they will be unable to build or test the system. "The burden of creating an understandable SRS falls on the shoulders of the writers; it is not the readers' responsibility to learn everything writers know in order to digest the SRS" [DAV93, 186].

Comprehensibility is a major requirement affecting the testing phase. Testing will likely be more effective if:

*"The design is well understood.
*Dependencies between internal, external, and shared components are well understood.
*Changes to the design are well communicated.
*Technical documentation is instantly accessible.
*Technical documentation is well organized.
*Technical documentation is specific and detailed.
*Technical documentation is accurate." [PRE01, 404]

Because the comprehensibility requirement describes the clarity of the specification to both the developers and the customers, the requirement can be found in the SRS category of the UML hierarchy. A system's comprehensibility is also affected by the conciseness NFR (see also conciseness).
Example: Every ULMTS program component shall be able to be reasonably understood by developers without reference to other information or other modules.

conceptuality – The conceptuality requirement "represents the concepts in the domain under study ... a conceptual model should be drawn with little or no regard for the software that might implement it" [FOW99] (see also the domain analysis cost and domain analysis time requirements). With a conceptual perspective, developers may conceive of what the customer requires, not how. The conceptual level is more abstract than the implementation level, in which the details of how the requirement is to be met is manifested in the code itself [KIT95, 166].

Because this requirement describes the efforts of the developers to conceive of the customer's specifications as the system is being designed, the conceptuality NFR may be found in the Design category of the UML hierarchy.
Example: The classes that comprise the ULMTS shall accurately represent concepts of and elements in a library materials tracking domain.

conciseness – The conciseness requirement describes the compactness of the program in terms of lines of code [MCC77, 32]. In addition to the number of lines of code, conciseness may also refer to how the system's specification is written. A verbose, wordy specification is one that is not precise and may affect the accuracy to which the developers are able to implement the requirements. A concise specification is "as short as possible without adversely affecting" any other specification in the SRS [DAV93, 182].

Since the conciseness requirement describes both the number of lines of code that comprise the system and the system's specification, factors based on the work of the developers, this requirement can be found in the Time category, TimePredeployment sub-category, Developer sub-sub-category. The conciseness NFR directly affects the understandability requirement (see also understandability).
Example: The ULMTS system documentation shall not exceed 500 pages or 1,500,000 words.

confidentiality – The confidentiality requirement describes the degree of protection that information sent and received within a system cannot be intercepted and read by unauthorized users. Confidentiality may be established by encrypting messages before they are sent; encryption "can take place through a secret key shared with the [message] receiver or by using the receiver's public key" [TAN02, 311]. The use of digital signatures may further enhance confidentiality; a public-key cryptosystem such as RSA (Rivest-Shamir-Adelman algorithm) has become a common way to protect the security of information being transmitted [TAN02, 311].

Because this requirement concerns the degree of security of information against unauthorized use, the confidentiality NFR may be found in the SystemIntegrity category of the UML hierarchy, in the Security sub-category.
Example: The sensitive user information (passwords, date of birth, addresses) stored in ULMTS shall be available only to the system administrator(s).

configurability – The configurability requirement describes "the ability to organize and control elements of the software configuration" [MCC77, 102]. A system's software configuration is defined as "the items that comprise all information produced as part of the software process" [PRE01, 323]. Each of these items is referred to as a software configuration item (SCI), and as the software process progresses, the number of SCIs quickly grows. The branch of software engineering that audits, accounts for and controls these SCIs is called configuration management, "the discipline of identifying the configuration of a hardware/software system at discrete points in time with the purpose of systematically controlling changes to the configuration and maintaining the integrity and traceability of the configuration throughout the system lifecycle" [THA97, 261]. (see also the integrity and traceability requirements).

Because the configurability requirements are handled by the developers during the design and implementation of the system, this NFR may be found in the Design category of the UML hierarchy. (see also reconfigurability)
Example: The ULMTS developers shall be able to control changes to the configuration of the hardware/software system at discrete points in time to maintain the integrity and traceability of the configuration throughout the system lifecycle.

consistency – The consistency requirement describes two aspects of a system's design and development. Consistency may refer to "the use of uniform design and documentation techniques throughout the software development project" [MCC77, 108] and it may also describe the system's specifications as listed in the SRS. As it relates to the latter definition, a specification is consistent "to the extent that its provisions do not conflict with each other, other specifications, or objectives [DAV93, 314].

The consistency requirement, as it describes specifications, may be subdivided into internal consistency, which states that no subset of individual requirements within an SRS conflict with each other, and external consistency, which states that no requirements contained in an SRS conflict with any already baselined project documentation [DAV93, 171]. (see also internal consistency and external consistency)

Consistency may be found in two places in the UML hierarchy: in the Design category and in the SRS category.
Example: No ULMTS record shall vary from the format provided by the University as provided in the prototype acceptance document.

controllability – The controllability requirement describes the degree of control a system's developers have over the writing of the software. Controllability is closely related to the testability requirement, since the more control developers have over the construction of the software, "the more testing can be automated and optimized" [PRE01, 438]. Pressman describes five facets of controllability:"

*All possible outputs can be generated through some combination of input.

*All code is executable through some combination of input.

*Software and hardware states and variables can be controlled directly by the test engineer.

*Input and output formats are consistent and structured.

*Tests can be conveniently specified, automated, and reproduced." [PRE01, 438]

As the controllability requirement is most closely related to a system's testability, it can be found in the Testing category of the UML hierarchy.
Example: The developers shall practice change control when reconfiguring the hardware and software components of ULMTS.

coordination cost – The coordination cost requirement describes the monetary amount necessary for the developers and customers to exchange information about a system as it is being developed. Co-ordination between the two sides is similar to communication (see also the communication cost and communication time requirements) in that for a software system to be successfully built, the developer and customer should coordinate continuously throughout the development phase. The developers may elicit requirements from the customers, but they should also ask further questions for absolute clarification of the specifications. Prototypes (see also the prototyping time and cost requirements) are a good first step to ensure that the developers are producing a product that accurately reflects the customers' expectations. Acceptance testing allows the customers an opportunity to "test-drive" the completed system to make sure it has all the necessary functionality.

Because this requirement concerns the cost of interaction between developer and customer, the coordination cost NFR may be found in the Cost category of the UML hierarchy, in the CostPredeployment sub-category.
Example: It shall cost no more than $350 to conduct one meeting where ULMTS developers and representatives of the University exchange information.

coordination time – The coordination time requirement describes the amount of time necessary for the system developers and customers to exchange information about the software that is to be produced (for a definition of coordination, see the communication cost requirement). Co-ordination between the two sides is similar to communication (see also the communication cost and communication time requirements) in that two-way co-ordination between the two sides is crucial if a software system that fully satisfies the customers' requirements and specifications is to be successfully developed.

Because this requirement concerns the length of time spent by both developer and customer exchanging information about a system to be built, the coordination time NFR can be found in two places in the UML hierarchy: in the Time category, in the TimePredeployment sub-category, in both the Developer and Customer sub-subcategories.
Example: It shall take no more than one business day to reply to a question posed by ULMTS developers to representatives of the University and vice versa.

correctness – This requirement describes "the extent to which a program satisfies its specification and fulfills the customer's mission objectives" [MCC77, 133]. If a Software Requirements Specification (SRS) document is used, it acts as the final word on what constitutes correctness. An SRS is composed of "a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements" [PRE01, 395]. Pressman writes that one metric for measuring correctness is the number of defects per KLOC (one thousand lines of code).

Since correctness concerns the customer's specifications and the accuracy with which the developers implement them, the correctness NFR can be found in the SRS category of the UML hierarchy.
Example: The ULMTS deliverables shall not vary from the system described in the SRS; developers will be held liable for any variance in said deliverables.

cost – The cost requirement is a very broad one, including costs incurred both before and after the software system has been deployed. Predeployment costs include domain analysis cost, prototyping cost, and development cost. Post-development costs may also be further categorized into installation costs, maintenance costs, and redo costs, each of which are comprised of several requirements.

Generally, the cost requirement concerns money, and may be defined as the total price of a system's deliverables. Calculating this total system cost is a complex task; there are many cost tradeoffs that must be considered as the development proceeds. Cost tradeoff is "the approach to problem solving that compares and evaluates alternatives (technical) solutions, especially where advantages and costs cannot be accurately measured in numbers, by considering costs of alternatives in comparison with benefits derived" [THA97, 182].

Because this NFR can be subdivided into so many other requirements, "Cost" is one of the primary categories in the UML hierarchy.

Example: The total cost of the development, installation, and maintenance of ULMTS to the University shall not exceed $875,000.

coupling – The coupling requirement describes the degree to which the modules and components of a given system rely on and interact with other modules and components of that system [PAG00]. Pressman writes that coupling may be viewed as a spectrum, with loosely coupled components being more desirable than tightly coupled ones [PRE01, 266]. Constantine's scale of cohesion divides coupling into seven categories [YOU79, 133]; they are, from the lowest degree to the highest:

*Null coupling – No data flows between two components.

*Data coupling – There is a one-to-one correspondence of data items between two components' methods.

*Stamp coupling – A portion of a cohesive, aggregate data structure is passed between two components.

*Control coupling – A variable that controls decisions in a subordinate or superordinate component is passed between these components.

*Common coupling – Two components communicate via a named, shared repository of global data.

*Global coupling – Communication takes place via global variables.

*Content coupling – One component makes use of data or control information maintained within the boundary of another component or branches into that component.

Since this requirement relates to the inter-relatedness of a system's components, the cohesiveness NFR can be found in the Design category of the UML hierarchy.

Example: No class in the ULMTS design document shall possess an undue dependence on any other class in the ULMTS design; no object in the ULMTS shall access any private member data of any other object in the ULMTS system.

customer evaluation time – In system/software engineering, a customer is "an individual or organization who specifies the requirements for and formally accepts delivery of a new or modified hardware/software product and its documentation; the customer may be internal or external to the parent organization of the project and does not necessarily imply a financial transaction between customer and developer" [IEE93, 88]. The customer evaluation time NFR describes the amount of time taken for "the tasks required to obtain customer feedback based on evaluations of the software representations created during the engineering stage and

implemented during the installation stage" [PRE01, 412]. The customer evaluates the prototypes of the system to be built, not the final system itself.
(see also prototyping cost, prototyping time)

The feedback of these prototype evaluations are used to aid the developers in clarifying their interpretation of the requirements in the SRS and how they should be implemented.(see the correctness requirement) The SRS acts as a contract, the "final arbiter" of what the customers require and what the developers are to produce.

Because it concerns system costs, the customer evaluation time requirement may be found in the Cost category of the UML hierarchy, in the CostPredeployment sub-category.
Example: The University shall not take more than 14 days to evaluate the ULMTS after it has been installed.

customer loyalty – The customer loyalty requirement describes the degree to which buyers and/or users of a program or software system are satisfied with the developers' product and will continue to use other products produced by the developers.

The customer loyalty NFR is affected by many other requirements: accuracy, affordability, compatibility, cost, dependability, efficiency, intuitiveness, learnability, performance, reliability, safety, security, user-friendliness, and versatility. It is difficult to accurately quantify total customer loyalty, since one system may be used by many (hundreds or thousands) of users. It is also difficult to measure because the users are human, and their perceptions are subjective, and likely to change from day to day. Customer loyalty may be measured by personal interviews, which is time-consuming, or questionnaires, which may not accurately quantify loyalty if all system users do not complete and return them.

Because this requirement concerns the subjective emotions that users have toward a software system, the customer loyalty NFR may be found in the EndUser category of the UML hierarchy.
Example: If, after 60 days, the University is adequately satisfied with the ULMTS system, the University shall provide positive testimony upon request to the competence of the ULMTS system developers.

customizability – The customizability requirement describes the degree of change that an end-user may make to a system in order to make it easier to use and more pleasant to work with. By having the ability to make enhancements to a software system (see also the enhanceability requirement), a human user may have more positive feelings towards a potentially complex system; "personalizing" one's interface with a system also relates to its user-friendliness (see also the user-friendliness requirement). Having a high degree of customizability in a system may be important to a customer since the end-users' personalizing of their interaction with the software may lead to greater productivity.

Because this requirement concerns end-user related changes made to a software system after it has been deployed and is in normal operation, the customizability NFR may be found in two places in the UML hierarchy: first, in the Future category, in the FutureChange sub-category; and second, in the EndUser category.
Example: ULMTS system administrators shall be able to customize the interface of the ULMTS in three menus: user profile, user settings, and audit file settings.

data-space performance – The data-space performance requirement describes the performance of the system as it handles the data used by the program, namely the program's variables as they are read from disk or a database and maintained in volatile memory while the system is in use. The data-space is independent of the code-space (see the code-space requirement); both use separate areas of memory while the program is executing.

The data-space performance requirement can be found in the SoftwarePerformance category, SpacePerformance subcategory. In IBM PC assembly language, data-space is referred to as the *data segment* and is addressed via the DS register [ABE98, 164].
Example: The ULMTS system shall be able to retrieve no fewer than 500 records per second from the primary hard disk.

decomposability – This requirement describes the degree to which a system's components and modules can be separated from the system for testing or restructuring. By controlling the scope of testing and restructuring, problems can be more easily isolated and "smarter retesting" can be performed [PRE01, 422]. Pressman gives two descriptions of a system's decomposability: the system is built from independent modules, and these software modules can be tested independently from one another [PRE01, 422].

As the decomposability requirement is most closely related to a system's testability, it can be found in the Testing category of the UML hierarchy.
Example: The software modules (classes) that comprise the ULMTS system shall be independently testable from one another, which is to say, no class will be required to be present when testing any other class.

degradation of service – Degradation of service is a type of denial of service attack, in which unauthorized or malicious hackers attempt to stop or slow network use with a barrage of network requests. Often hackers use "zombies," network computers belonging to others who are tricked into working for the hacker. Under a degradation-of-service attack, the intended victim's servers are not completely overwhelmed, but are instead stunned with a heavy concentration of significant amounts of network activity. As a result, authorized users may experience long delays when requesting network connections and slow transmission speeds of information.

Because this requirement concerns the authentication of bona fide users and prevention of malicious barrages of network requests by hackers, the degradation of service NFR can be found in the SystemIntegrity category of the UML hierarchy, in the Security sub-category.
Example: Authorized ULMTS users shall experience a delay no longer than ten seconds as a result of attempted denial-of-service attacks.

dependability – The dependability requirement describes the degree to which software can be expected to perform its intended function with required precision, and do so without failure. (see also the precision NFR) Dependability is a very important non-functional requirement: if a software system is not dependable, it is virtually worthless.
The dependability NFR is synonymous with the reliability NFR (see also the reliability requirement).

Because the dependability requirement describes the lack of failure shown by a system that has already been deployed, it may be found in the Future category of the UML hierarchy, in the FutureOperation sub-category.

Example: The ULMTS software shall be expected to perform its intended function with a MTBF (mean time between failures) of thirty days.

development cost – The development cost requirement describes the total monetary amount of developing a software system for a customer (for a definition of development, see the development time requirement). The development cost may be affected by many factors: the size and scope of the project, the number of developers, the length and detail of requirements elicitation, the number and complexity of the prototypes, the amount of developer-customer communication and feedback (see also the communication cost requirement), the method and accuracy of testing, and the cost of deploying the final
product to the customer.

Because this requirement concerns the total cost of all development activities, the development cost NFR could have been placed in the Design and Testing categories of the UML hierarchy, but for simplicity's sake, it can be found in the Cost category, under the CostPredeployment sub-category.
Example: The cost of the development phase of ULMTS to the University shall not exceed $175,000 (note: the total cost of ULMTS shall not exceed $875,000)

development time – This very broad requirement describes the interval of time between the moment the first phase of software development begins and the delivery of the final software system to the customer. *Development* is an overarching term; it may entail eliciting requirements from the customer, analyzing and clarifying specifications, design, constructing prototypes, eliciting customer feedback, coding, testing, debugging, and performing customer acceptance testing.

Because this requirement concerns the sum time of all development activities, the development time NFR could have been placed in the Design and Testing categories of the UML hierarchy, but for simplicity's sake, it can be found in the Time category, under the TimePredeployment sub-category, in the Developer sub-subcategory.
Example: It shall take no longer than 120 days for system developers to successfully design, code, and test all of the components (classes) of the ULMTS system.

distributability – The distributability requirement describes the "degree to which software meets interoperability standards for systems distributed across multiple platforms" [PAG00, 44]. Perhaps the best example of getting many major software vendors to agree on interoperability standards is the Common Object Request Broker Architecture (CORBA). CORBA allows objects (instances of classes) to communicate, "not only across similar machines, but also across machines that are different models, running different operating systems, and connected by an assortment of different networks" [PAG00, 107]. As such, the distributed nature of these different platforms becomes "transparent" to software system developers.

Because this requirement describes software that may be run on and communicate with different systems, the distributability NFR may be found in the IntersystemInteraction category of the UML hierarchy.
Example: The ULMTS shall meet interoperability standards for systems distributed across multiple platforms as described in ISO-9000, section 2.11.301.

diversity – The diversity requirement describes the degree of difference between a system's components and modules. It may also refer to the degree of difference between the data structures and data types throughout a program. The counterpart requirement of diversity is commonality, which refers to "the use of standard data structures and types throughout the program" [MCC77, 344]. (see commonality)

Diversity of a system's components is important when there is required functionality that existing components cannot accomplish. Component reuse increases the level of a software system's commonality; design and implementation of new components increases a system's diversity.

Because this NFR refers to the components that comprise a given software system, the diversity requirement can be found in the Design category of the UML hierarchy.
Example: The components that make up the ULMTS (classes) shall have a minimum amount of diversity; this will be accomplished by having all classes use the standard data structures and types of the STL (Standard Template Library) throughout the entire system.

domain analysis cost – The domain analysis cost requirement describes the total monetary amount of performing domain analysis. Domain analysis is "the identification, analysis, and specification of common requirements from a specific application domain ... object- oriented domain analysis is the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks" [FIR93, 86].

According to Pressman, "the goal of domain analysis is straightforward: to find or create those classes that are broadly applicable, so that they may be reused" [PRE01, 218]. The cost of performing domain analysis may be affected by several factors: the complexity of the domain, the number of pre-existing classes that are available for reuse, and the familiarity of the developers with the domain (perhaps from previous software projects).

Because this requirement concerns the cost of analyzing a business domain before any system design activities even begin, the domain analysis cost NFR may be found in the Cost category of the UML hierarchy, in the CostPredeployment sub-category.
Example: The cost of performing domain analysis by the developers of the system shall not exceed $17,500.

domain analysis time – The domain analysis time requirement describes the total length of time required to perform domain analysis (for a definition of domain analysis, see the domain analysis cost requirement).

Like the cost of domain analysis, the amount of time required to perform domain analysis may be affected by several factors: the complexity of the domain, the number of pre-existing classes that are available for reuse, and the familiarity of the developers with the domain (perhaps from previous software projects).

Because this requirement concerns the time needed to analyze a business domain before any system design activities even begin, the domain analysis time NFR may be found in the Time category of the UML hierarchy, in the TimePredeployment sub-category, in the Developer sub-subcategory.

Example: It shall take no longer than 21 days for ULMTS system developers to conduct the necessary amount of domain analysis (domain feature gathering) prior to starting the design phase of ULMTS.

efficiency – The efficiency requirement describes "the degree to which software makes optimal use of system resources as indicated by the following subattributes: time behavior and resource behavior" [IEE93, 36].

The time for a system to carry out a task is closely related to system resources. Faster processors allow for faster reads and writes, and multiprocessors allow for concurrent processing, increasing efficiency. Larger areas of memory allow for larger buffers and double buffering, which also make system functionality more efficient. The efficiency non-functional requirement also describes the extent to which the system's software performs its intended functions with a minimum consumption of the system's resources [IEE93, 37].

Because this requirement describes the usage of system resources after the system has been deployed and is in normal operation, the efficiency NFR may be found in the Future category, in the FutureOperation sub-category of the UML hierarchy. (see also the performance requirement)
Example: The ULMTS shall support no fewer than 500 database lookups per second.

elasticity – The elasticity requirement describes the effort required to modify or add to a software system. A system may need to be "elastic" if there will be known changes in its operating environment after it has been deployed and is in normal operation. Also, the probability that unseen environment changes and/or future updates may dictate a system be designed and built with elasticity in mind.

There are a number of other requirements that are related to the elasticity NFR. The more modifiable a system is, the higher amount of elasticity it will have. An elastic system is likely to have more adjustability, enhanceability, evolvability, extensibility, flexibility and scalability (see these requirements).

Because it relates to the ease of making modifications to a software system after it has been deployed and is in normal operation (either for anticipated or unknown future changes in the operating environment), the elasticity requirement can be found in the Future category of the UML hierarchy, in the FutureChange sub-category.
Example: The ULMTS shall be elastic to the extent that changes in client needs shall allow the functionality of the ULMTS to be updatable.

enhanceability – "As software is used, the customer/user will recognize additional functions that will provide benefit. Perfective maintenance extends the software beyond its original functional requirements" [PRE01, 222]. The enhanceability requirement describes the degree of this "perfective maintenance." Increasing performance or capability by enhancing current functions or by adding new functions or data is natural reaction of a customer using a software system and deciding it should provide functionality that was not agreed upon in the SRS; it is a natural human tendency to "want more" than what is currently available.

The enhanceability requirement is closely related to seven other requirements: adaptability, adjustability, customizability, extensibility, flexibility, modifiability, and reconfigurability.

Because this requirement concerns maintenance-related changes made to a software system after it has been deployed and is in normal operation, the enhanceability NFR may be found in two places in the UML hierarchy: in the Future category, in the FutureMaintenance and FutureChange sub-categories.
Example: ULMTS system administrators shall be able to enhance/make changes to the interface of the ULMTS in three menus: user profile, user settings, and audit file settings.

evolvability – The evolvability requirement describes the effort required to modify an operational software system. A system may be required to be evolvable if there will be known changes in its operating environment after it has been deployed and is in normal operation. Also, the probability that unseen environment changes and/or future updates may dictate a system be designed and built with evolvability in mind.

There are a number of other requirements that are related to the evolvability NFR. The more modifiable a system is, the more likely it will be able to evolve to meet the changing required functionality. An evolvable system is likely to have more adjustability, enhanceability, extensibility, flexibility, and scalability (see these requirements).

Because it relates to the ease of making modifications to a software system after it has been deployed and is in normal operation (either for anticipated or unknown future changes in the operating environment), the evolvability requirement can be found in the Future category of the UML hierarchy, in the FutureChange sub-category.
Example: The ULMTS shall be evolvable to the extent that changes in client needs shall allow the ULMTS to able to support additions to its core functionality for a period of seven years after installation (see also the maintenance section of the SRS, section 3.1.1.4).

execution cost – The execution cost requirement describes the monetary amount of executing a software system once it has been deployed and is in operation. Execution costs include both the cost of system maintenance (such as error correction and system upgrades) and operating cost (such as payroll and hardware costs); it does not include any costs associated with retiring or replacing the system (see also the replaceability and retirement cost requirements).

Because this requirement concerns the cost of operating a software system after it has been delivered to the customer and installed, the execution cost NFR may be found in the Cost category of the UML hierarchy, in the CostPostdeployment sub-category, in the MaintenanceCost sub-subcategory.
Example: The total cost of the hardware of ULMTS (one server, 40 terminals, and five printers) to the University shall not exceed $55,000 (note: the total cost of UMLTS shall not exceed $875,000).

extensibility – The extensibility requirement involves extending both the design of the system and the software system itself. In regards to design extensibility, the requirement describes the "degree to which architectural, data, or procedural design can be extended by adding variations to an already stated theme" [PAG00, 177]. It is much easier to make alterations and additions during the design phase of the development process before any code has been written; time and money are wasted when an implemented system must be extended or corrected when those changes could have been made during the design phase.

In regards to software system extensibility, this requirement describes making changes to a system after it has been deployed. Altering software, though, runs the risk of introducing problems. Meyer addressed this situation in what he called the "open-closed principle" which dictates that the modules, methods, and classes of a system should be open for extension, while closed for modification [MEY90, 329]. In other words, the software should be designed so that its capabilities can be extended without changing it; new implementations can be added (*extending* the software) without changing any existing classes.

The extensibility requirement can be found in two places in the UML hierarchy: in the Design category, and in the Future category, in the FutureChange sub-category.
Example: Changes in user needs shall allow the functionality of the ULMTS to be extended by the developers as described in the maintenance section of the SRS.

external consistency – The external consistency requirement describes the specifications within a system's SRS. To be externally consistent, no requirements contained in an SRS conflict with any already baselined project documentation [DAV93, 205]. To ensure external consistency among a system's requirements, "one must create and maintain full cross-references between all requirements and relevant statements made in other documents [DAV93, 205].

External consistency is one specialization of the more general consistency requirement; internal consistency is the other. The internal consistency NFR states that no subset of individual requirements within one SRS may conflict with any other subset [DAV93, 206]. (see also consistency and internal consistency)

The external consistency requirement may be found in the SRS category of the UML hierarchy.
Example: The individual modules (classes) of ULMTS shall be externally consistent in that they follow the standard variable, function, and documentation format as described in the ISO-9000 C++ standard, section 1.31.19.

fault-tolerance – The fault-tolerance requirement describes the damage that occurs when the program encounters a fault [MCC77, 143]. A fault is an "incorrect step, process, or data definition in a computer program ... in common usage, the terms *error* and *bug* are used to express this meaning" [IEE90, 33]. Fault tolerance is a more specialized version of the tolerance requirement (see also tolerance).

A system's fault tolerance is closely related to its survivability. Whereas fault tolerance describes the amount of damage a system can handle, survivability describes how well the software will perform after experiencing this damage.

A system may be designed to have a higher degree of fault tolerance. As with the survivability requirement, the more independent a software system's components are, the less their correct operation depends on other components. This component independence leads to a more fault tolerant system as a whole, since damage to one component is not likely to affect others.

Because this requirement concerns the damage that may affect a software system's ability to correctly perform its functionality, the fault-tolerance NFR may be found in the SystemIntegrity category of the UML hierarchy, in the Error sub-category.

Example: The ULMTS will be fault-tolerant to the degree that, upon encountering an incorrect step, process, data definition, or thrown exception, ULMTS will allow the user to revert to a previous state before failing gracefully.

feasibility – A requirement is feasible "to the extent that the lifecycle benefits of a system specified will exceed lifecycle costs and that the system will satisfy the system requirements. It also implies validating that the specified system can be sufficiently maintainable, reliable, and human-engineered to keep a positive lifecycle balance" [BOE84, 196].

Putnam and Myers list four dimensions of software feasibility:

*"Technology – Is a project technically feasible? Is it within the state of the art? Can defects be reduced to a level matching the application's needs?

*Finance – Is it financially feasible? Can development be completed at a cost the software organization, its client, or the market can afford?

*Time – Will the project's time-to-market beat the competition?

*Resources – Does the organization have the resources needed to succeed?" [PUT97, 155]

Feasibility of a software system is a "yes or no" question; it cannot be partially feasible. At the center of the feasibility of a system is whether it is practical and reasonable to build based on a host of constraints: time, budget, existing hardware, personnel, maintenance, future changes, et cetera.

Because this requirement concerns whether or not a system will fulfill its necessary functionality in an acceptable manner as described by its requirements, the feasibility NFR may be found in the SRS category of the UML hierarchy.

Example: The delivery of the ULMTS to the University shall be feasible to the degree that the ULMTS is sufficiently maintainable, reliable, and human-engineered as specified in the SRS.

flexibility – The flexibility requirement describes the "effort required to modify an operational program" [MCC77, 208] or system. A software system may be required to be flexible if there will be known changes in its operating environment after it has been deployed and is in normal operation. Also, the probability that unseen environment changes and/or future updates may dictate a system be designed and built with flexibility in mind.

There are a number of other requirements that are related to the flexibility NFR. The more modifiable a system is, the higher amount of flexibility it will have. A flexible system is likely to have more adjustability, enhanceability, evolvability, extensibility, and scalability (see these requirements).

Because it relates to the ease of making modifications to a software system after it has been deployed and is in normal operation (either for anticipated or unknown future changes in the operating environment), the flexibility requirement can be found in the Future category of the UML hierarchy, in the FutureChange sub-category.

Example: Changes in user needs shall allow changes to the functionality of the ULMTS to be possible.

formality – The formality NFR describes a software requirement that "can either be proven 'correct' through proof-of-correctness methods or can be translated directly into an operational application program that implements the requirements" [THA97, 316]. The counterpart to formality is the naturalness requirement (see naturalness). A natural requirement is one written in language more likely to be spoken and understood by the customer, using terms that demonstrate domain knowledge. A formal requirement more benefits the developers, as it is more likely to be more precise and descriptive to one who lacks the domain knowledge familiar to the customer. The SRS is more likely to be written with more formality than naturalness.

Because it deals with the language of the requirements in the SRS, the formality NFR can be found in the SRS category of the UML hierarchy.

Example: The requirements written in the SRS shall either be able to be proven 'correct' through proof-of-correctness methods or can be translated directly into an operational application program that implements the requirements.

generality – The generality requirement describes the "breadth of potential application of program components" [MCC77, 244]. These components, such as structs and classes in object-oriented languages, have the capacity for reuse (see also the reusability requirement). If one of these components may be reused many times in different contexts and different projects, it may be described as a *general* component. Generality is a desirable requirement, since code reuse saves design, implementation, testing, and maintenance time.

The counterpart NFR to generality is specificity (see the specificity requirement). Just the opposite of generality, a component that is specific has little capacity for reuse and application in software systems other than the one it was designed and implemented for. A requirement with a high degree of specificity is not necessarily undesirable; it is just that there is no time saved because the component is not reused in other systems.

Because this NFR refers to the degree of potential application of the components that comprise a given software system, the generality requirement can be found in the Design category of the UML hierarchy.

Example: The components that make up the ULMTS (classes) shall general to the degree that they be written in such a manner that they may be reused when similar functionality is required (see the code reuse section of the SRS, 2.4.1.1)

guidance – The guidance requirement refers to the amount of assistance an end-user may obtain from the system while using it. Typically, assistance comes in the form of accessing online help files or, less conveniently, user manuals. Guidance is useful for both novice users, who may not know much about the system's functionality, or experienced users, who may need a reminder how to perform a complex task. A high level of guidance is especially important in large, complex systems that may lack sufficient user-friendliness (see also the user-friendliness, learnability, and trainability requirements).

Because this requirement concerns the degree of help a system provides to its users through the system's interface, the guidance NFR may be found in the UserInterface category, ContinuedUse sub-category.

Example: The ULMTS shall provide assistance to users while using ULMTS in the form of online help files.

hardware cost – This straightforward requirement refers to the cost of a system's hardware, the "mechanical, magnetic, and electron design, structure, and devices of a computer" [THA97, 188]. A hardware system can be viewed as being composed of interacting hardware configuration items (HCIs), entities that "exist where functional allocations have been made that clearly delineate the separation between equipment functions and software functions, and the hardware has been established as a configuration item" [THA97, 188] (see also the configurability requirement). Generally speaking, a hardware system must be in place before a software system may be deployed upon it.

The hardware cost requirement may be found in the Cost category of the UML hierarchy, in the CostPredeployment sub-category.

Example: The total cost of the hardware of ULMTS (one server, 40 terminals, and five printers) to the University shall not exceed $55,000 (note: the total cost of UMLTS shall not exceed $875,000).

impact analyzability – Impact analysis is one part of the broader process called analysis, the "process of studying a system by partitioning the system into parts (functions or objects) and determining how the parts relate to each other to understand the whole" [IEE83, 64].

Impact analysis takes place during the control activity in configuration management (CM). Control refers to the management of changes made to a system during development, particularly as they relate to the baseline. Impact analysis, therefore, "looks forward to the design and development baselines" [PHI00, 265], which is to say that changes to the software system as it currently exists will take into account how these changes will impact the system in the future.

Because this requirement describes the analysis of design changes made to the system during the development phase, the impact analyzability NFR can be found in the Design category of the UML hierarchy.

Example: The developers of ULMTS will practice impact analysis
by managing changes made to the system during development; the developers will also take into account how these changes will impact ULMTS after it has been installed.

independence – There are two types of independence requirements: hardware independence and software system independence.

Hardware independence is defined as "the degree to which the software is decoupled from the hardware on which it operates" [MCC77, 311]. An example of this is the Java language. Proponents of Java tout it's "write once, run anywhere" capability, meaning that a Java program can be run on many different platforms. This is made possible by the use of the Java Virtual Machine (JVM), an interpreter that converts the Java bytecode to code that is understandable by the operating system on which the program is running.

Software system independence is "the degree to which the program is independent of nonstandard programming language features, operating system characteristics, and other environmental constraints" [MCC77, 311]. The more independent a system is, the more likely it will be portable (see also the portability requirement) and have a high degree of mobility (see also the mobility requirement).

The independence NFR can be found in the Future category of the UML hierarchy, FutureOperation sub-category.

Example: Since the ULMTS software will run on the library's existing hardware and not ported to other platforms, it is not necessary that the program be independent of nonstandard programming language features, operating system characteristics, or any other environmental constraints.

informativeness – The informativeness requirement describes the degree of knowledge a system may impart about itself. A related requirement is the guidance NFR. Typically, knowledge about a system is provided by online help files or user manuals. A high level of informativeness is important for both novice users, who may not know much about the system's functionality, or experienced users, who may need a reminder how to perform a complex task. Informativeness is especially useful in large, complex systems that may lack sufficient user-friendliness (see also the user-friendliness, learnability, and trainability requirements).

Because this requirement concerns the degree of help a system provides to its users through the system's interface, the informativeness NFR may be found in the UserInterface category, ContinuedUse sub-category.
Example: The ULMTS will be informative to users while using the system by providing assistance in the form of online help files.

inspection cost – A software inspection is "a semiformal to formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the originator to detect faults, violations of development standards, and other problems" [THA97, 411].

An inspection is just one activity that makes up a formal technical review, which is performed by the software engineers developing a system. According to Pressman, this formal technical review has five objectives:

*"to uncover errors in function, logic, or implementation for any representation of the software

*to verify that the software under review meets its requirements

*to ensure that the software has been represented according to predefined standards

*to achieve software that is developed in a uniform manner

*to make projects more manageable." [PRE01,m 183]

Because this requirement describes the cost of performing inspections of the software system before it has been delivered to the customers and deployed, the inspection cost NFR can be found in the Cost category of the UML hierarchy, in the CostPredeployment sub-category.
Example: The total cost of inspection and verification of ULMTS to the University shall not exceed $5,000 (note: the total cost of ULMTS to the University shall not exceed $875,000)

inspection time – The inspection time requirement describes the amount of time it takes the software engineers to perform inspections and the formal technical review of a system before it is delivered to the customers (for the definition of an inspection, see the inspection cost NFR).

Because this requirement describes the time to perform inspections of the software system before it has been delivered to the customers and deployed, the inspection time NFR can be found in the Time category of the UML hierarchy, in the TimePredeployment sub-category, in the Developer sub-subcategory.
Example: It shall take no longer than eight business days for representatives of the University to inspect the ULMTS systems prior to formally accepting it for installation.

integrity – The integrity requirement describes "the extent to which access to software or data by unauthorized persons can be controlled" [MCC77, 137]. The system's ability to withstand both

accidental and intentional attacks from hackers may be measured by metrics written by testers intentionally trying to compromise the system; attacks can be made "on all three components of software: programs, data, and documents" [PRE01, 308].
Pressman defines two additional attributes of integrity: threat and security. Threat is "the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time; security is the probability (which can be estimated or derived from empirical evidence) that the attack of the specific type will be repelled" [PRE01, 308]. He provides a formula for integrity:

integrity = summation [(1 – threat) * (1 – security)]

where threat and security are summed over each type of attack.

The integrity requirement can be found in the SystemIntegrity category of the UML hierarchy, Security sub-category.
Example: Access to all ULMTS software and data by unauthorized persons can be controlled by using testing metrics written by testers intentionally trying to compromise the system as described in the testing section of the SRS (5.1.1.4).

inter-operability – The inter-operability requirement is defined as "the ability of two or more systems to exchange information and to mutually use the information that has been exchanged" [IEE83, 99]. More broadly, inter-operability is "the effort required to couple one system to another" [MCC77, 114]. Inter-operability is nearly synonymous with compatibility, a requirement that describes "the ability of two or more systems to exchange information" [THA97, 318]. Inter-operability is especially important in distributed systems and client/server architectures where other systems the current system must interact with may be running on different platforms.

Because the inter-operability requirement describes the interaction (information exchange) between two or more systems, it can be found in the IntersystemInteraction category of the UML hierarchy.
Example: The ULMTS shall be fully interoperable with the UNMTS (the University’s nuclear materials tracking system).

internal consistency – The internal consistency requirement describes the specifications within a system's SRS. To be internally consistent, no subset of individual requirements within one SRS may conflict with any other subset [DAV93, 281]. Internal consistency may be achieved with CASE tools, most of which do consistency error reports and consistency checking for data flow diagrams [THA97, 311].

Internal consistency is one specialization of the more general consistency requirement; external consistency is the other. The external consistency NFR states that no requirements contained in an SRS conflict with any already baselined project documentation [DAV93, 97]. (see also consistency and external consistency)

The internal consistency requirement may be found in the SRS category of the UML hierarchy.
Example: The individual modules (classes) of ULMTS shall be internally consistent in that they follow the standard variable, function, and documentation format as described in the ISO-9000 C++ standard, section 1.31.19.

intuitiveness – The intuitiveness requirement describes the degree to which a user can learn and use a software system without any training specific to that particular system. Since intuition is the faculty to know without having to consciously reason, a system may be used intuitively if it makes use of features common to the human experience. One way of accomplishing this is to hide the low-level functionality of a system from a user, instead providing a more user-friendly interface; this interface may be further made more intuitive to use by employing features familiar to most users. An example of this is icons. The human brain more easily remembers colors and visual images than strings of characters and digits. By representing files and directories as icons, the user may more easily navigate a complex directory-structure-oriented system.

Since the intuitiveness NFR refers to a description of a system's user interface, it can be found in two places in the UML hierarchy; first, in the UserInterface category, Learnability sub-category, and second, in the UserInterface category, ContinuedUse sub-category.
Example: The ULMTS system's user interface shall model the look-and-feel of the Microsoft Windows operating system family.

learnability – The learnability requirement describes the ease with which a system can be learned by a user. A system's learnability may be affected by the following conditions:

* The end-users' previous experience with earlier versions of the system
* The end-users' previous experience with different systems that had similar functionality
* The system's user interface
* The system's user-friendliness
* The intuitiveness of the system's components
* The understandability of the system's functionality

Learnability is a more specific variation of the closely-related usability requirement (see also usability). It is also a counterpart to the trainability requirement, since an end user who is learning how to operate a new system or an updated version of a previous system is being trained (see also trainability). The learnability requirement can be found in the UserInterface category of the UML hierarchy, Learnability sub-category.
Example: It shall take no longer than seven days for employees of the University library to be trained in the use of ULMTS to be able to satisfactorily perform the four main functions as described in the usability requirement portion of the SRS.

main-memory performance – As its name suggests, this requirement describes the performance of a system's main memory, also called primary memory. Memory in a computer is the area where programs and data are stored. Computers use binary digits (ones and zeros) in their memories, and all memories are made up of addressable locations (sometimes called cells).

A system's main-memory performance may be affected by the cell size; an eight-bit (one byte) cell can store only half the data a sixteen-bit cell can. Another factor affecting performance is the rate of transfer between main memory and the CPU. Historically, "CPUs have always been faster than memories; as memories have improved, so have CPUs, preserving the imbalance" [TAN99, 232]. One strategy to improve performance is to use a cache, a high-speed area of memory.

Because this requirement concerns the performance of main-memory, typically a hard disk, the main-memory performance NFR can be found in the Performance category of the UML

hierarchy, in the HardwarePerformance sub-category, in the HardwareSpacePerformance sub-subcategory.
Example: The main memory of the ULMTS system shall support no fewer than 75 reads and/or writes per second of records no smaller than 5k.

maintainability – The maintainability requirement describes the effort required to locate and fix an error in a program [MCC77, 143]. Put another way, is it "the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements" [PRE01, 322]. Pressman writes that there is no way to measure maintainability directly [PRE01, 322], but Hitachi developed an indirect metric called spoilage to measure maintainability. Spoilage measures the cost of fixing errors after the system has been deployed [TAJ81, 95]. When the ratio of spoilage to total project cost becomes large, the customer may choose to take appropriate action
[TAJ81, 95].

Because this requirement describes changes or enhancements made to a system after it has been deployed, the maintainability requirement can be found in the Future category, FutureMaintenance subcategory.
Example: ULMTS shall be maintained by the University library system administrator at a time cost of no more than two hours per work week.

maintenance cost – The maintenance cost requirement refers to the price of locating and fixing an error in a program (see the maintainability NFR). Taken together, the maintenance cost requirement and the maintenance time requirement comprise a system's maintainability.

Other maintenance tasks besides correcting system faults are to improve performance or other attributes, or to adapt the system to changes in the environment [IEE83, 61]. Because maintenance of a system occurs after the product has been to delivered to the customer and deployed, the maintenance cost requirement can be found in two places in the UML hierarchy: once in the Future category, FutureMaintenance sub-category, and again in the Cost category, CostPostDeployment sub-category, MaintenanceCost sub-subcategory.
Example: The total cost to the University of maintaining ULMTS for a ten-year period following its deployment shall not exceed \$5,000 (note: the total cost of ULMTS to the University shall not exceed \$875,000).

maintenance time – The maintenance time requirement refers to the time required to locate and fix an error in a program (see the maintainability NFR). Taken together, the maintenance cost requirement and the maintenance time requirement comprise a system's maintainability.

Other maintenance tasks besides correcting system faults are to improve performance or other attributes, or to adapt the system to changes in the environment [IEE83, 13]. Because maintenance of a system occurs after the product has been to delivered to the customer and deployed, the maintenance time requirement can be found in two places in the UML hierarchy: once in the Future category, FutureMaintenance sub-category, and again in the Time category, TimePostDeployment sub-category.
Example: It shall take no longer than three business days for developers of (or a representative of developers of) ULMTS to respond to a maintenance request by a University employee after ULMTS has been successfully installed.

maturity – The maturity requirement describes the degree to which a software system is mature. A system is said to be mature when it has attained a final, desired state of full development. Ideally, maturity is the goal of all systems, but often a system will never reach maturity, instead being replaced by another system (see also the replaceability requirement).

Several other requirements affect the likelihood a system will reach full maturity: adjustability, customizability, enhanceability, evolvability, modifiability and reconfigurability. The easier a software system may be changed or updated after it has been deployed, the more likely it may attain maturity. In some cases a system may be deemed mature only because the customers have accepted the fact that their software has shortcomings but is not willing to expend the time or cost to correct them.

Because this requirement describes the point in a system's lifetime that it will operate without any further alterations or improvements, the maturity NFR may be found in the Future category of the UML hierarchy, in the FutureOperation sub-category.
Example: The ULMTS system shall take no longer than six months after deployment to attain a final, desired state of full development.

mean performance – The mean performance requirement describes the average performance of a software system, taking all of its combinations of disparate parts into account. For example, the average of peak-period performance and off-peak-period performance comprises a system's mean performance (see also the peak-period performance and off-peak-period performance requirements). So does the average of the main-memory performance and the secondary performance (see also the main-memory performance and the secondary performance requirements), and time and space performance averages. Knowing a software system's mean performance is useful when one wants to get a "big picture" of the system, an idea of its total performance without being concerned with the details of its individual hardware components or particular times of the day it is being used.

Because this requirement concerns the performance of both time and space, hardware and software, the mean performance NFR may be found in all four sub-categories that are permutations thereof (HardwareTimePerformance, HardwareSpacePerformance, SoftwareTimePerformance, and SoftwareSpacePerformance).
Example: The ULMTS system shall attain a mean performance of no fewer than 150 transactions per second, whether during times of peak usage or off-peak usage.

measurability – The measurability NFR describes how easily measured a requirement is. A requirement (or specification) is measurable "when some finite process exists to verify that the product meets the requirements" [THA97, 64]. The terms measure and metric are very similar, except a metric is more closely associated with *attributes*: "a metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute" [IEE93, 51].

The measurability requirement is important in system testing, since it can only be determined that a specification is met by having a concrete, tangible number to compare with the SRS requirement. A closely related NFR to measurability is the repeatability requirement (see repeatability). A specification that can be measured once, and have the results repeated to yield the same measure is a good indication the specification has been successfully met.

Because this requirement is concerned with the quantifiable attributes of requirements and components, the measurability NFR may be found in the SRS category of the UML hierarchy.
Example: To facilitate testing of the ULMTS system, all requirements shall be quantitatively measurable using the metrics listed in SRS section 3.1.5.1.

mobility – The mobility requirement describes the ease with which the software can be transposed from one environment to another. Mobility will be an important NFR if the customer knows that before it is retired, the system will at some point have to be transferred to another operating environment.

Unlike the portability requirement, which is nearly synonymous with mobility, this NFR does not assume that the new environment the system will be moved to will have a different hardware configuration or operating system. The mobility requirement is synonymous with the nomadicity NFR. (see portability, nomadicity)

The mobility requirement may be found in the Future category of the UML hierarchy, FutureOperation subcategory since the ability to move the system to different operating environments assumes the system has been deployed in an initial environment to begin with.
Example: Since the ULMTS software will run on the library's existing hardware and not be ported to other platforms, it is not necessary that the software be independent of any environmental constraints.

modifiability – The modifiability requirement may describe both the SRS and the final, delivered system the SRS specifies.

An SRS is modifiable if "its structure and style are such that any changes can be made easily, completely, and consistently" [IEE84, 61]. Davis lists two reasons an SRS should be modifiable: first, a customer's needs are always evolving, and second, the SRS, like all complex software-related documents, will contain errors [DAV93, 224]. The SRS will be modified as new needs are discovered, old needs must be altered, or if obsolete needs must be abandoned.

In regards to the modifiability of the software system as a whole, making modifications is related to the degrees of cohesion exhibited by its components and the degree of coupling existent between components [YOU79, 361]. (see also the cohesiveness and coupling requirements)

Concerning system modifiability, the modifiability requirement differs from adaptability in that adjustments to the system are made after the system is put into place, whereas with the adaptability requirement, modifications are made to the system before it is implemented so that it satisfies the constraints of all elements already in place.

As it relates to the changeability of a system's requirements as the customer's needs evolve, the modifiability NFR can be found in the SRS category of the UML hierarchy. As it relates to the changeability of the system itself, the modifiability requirement can be found in the Future category, FutureChange sub-category.
Example: Changes in user needs shall allow changes to the functionality of the ULMTS possible.

modularity – When software is described as being *modular*, it is "divided into separately named and addressable components, often called modules, that are integrated to satisfy problem requirements" [PRE01, 321]. It is by using the principle of modularity that a large, monolithic problem or program can be broken into more manageable, easy to comprehend parts. Meyer [MEY90, 177] lists five reasons a system might benefit from being modularized:

*"Modular decomposability: If a design method provides a systematic mechanism for decomposing the problem into subproblems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

*Modular composability: If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

*Modular understandability: If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.

*Modular continuity: If small changes to the system requirements result in changes to individual modules, rather than system-wide changes, the impact of change-induced side effects will be minimized.

*Modular protection: If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized."

Because modularity concerns the grouping of functions and logic into independent components (such as classes and structs), the modularity requirement may be found in the Design category of the UML hierarchy.
Example: The ULMTS system shall be written in modules (classes) to allow for code reuse and facilitate testing before the final system is deployed.

naturalness – A natural requirement is one written in language more likely to be spoken and understood by the customer, using terms that demonstrate domain knowledge. The naturalness requirement is the counterpart to the formality requirement (see formality); it describes software requirements that can be translated directly into an operational application program that implements the requirements. A natural requirement is more beneficial to the customers, as it is more likely to be more written in language more familiar to the customer than the developer. The initial, first draft of the software system's specifications is more likely to be written with more naturalness than formality.

Because it deals with the language of the requirements in the SRS, the naturalness NFR can be found in the SRS category of the UML hierarchy.
Example: The system requirements of the ULMTS shall be written in language that the customer can understand; all technical language and acronyms will be annotated.

nomadicity – The nomadicity requirement describes the ease with which the software can be transposed from one environment to another. Nomadicity will be an important NFR if the customer knows that before it is retired, the system will at some point have to be transferred to another operating environment.

Unlike the portability requirement, which is nearly synonymous with nomadicity, this NFR does not assume that the new environment the system will be moved to will have a different hardware configuration or operating system. The nomadicity requirement is synonymous with the mobility NFR. (see portability, mobility)

The nomadicity requirement may be found in the Future category of the UML hierarchy, FutureOperation subcategory since the ability to move the system to different operating environments assumes the system has been deployed in an initial environment to begin with.

Example: Since the ULMTS software will run on the library’s existing hardware and not be ported to other platforms, it is not necessary that the software be independent of any environmental constraints.

observability – This requirement describes the visibility of a system's functionality, particularly as it relates to its testability. Pressman describes observability in eight ways:"

*Distinct output is generated for each input.
*System states and variables are visible or queriable during execution.
*Past system states and variables are visible or queriable (e.g., transaction logs).
*All factors affecting the output are visible.
*Incorrect output is easily identified.
*Internal errors are automatically detected through self-testing mechanisms.
* Internal errors are automatically reported.
*Source code is accessible." [PRE01, 209]

Because of its close relationship with a system's testability, the observability requirement can be found in the Testing category of the UML hierarchy.

Example: The ULMTS system shall be observable in the following three ways:

a) Distinct output shall be generated for each input.
b) Current system states and variables shall be visible and queriable during execution.
c) Past system states and variables shall be visible and queriable using transaction logs.

off-peak-period performance – The off-peak-period performance requirement describes the performance of a software system during the time interval that it is being used the least. Systems in different domains have different peak periods, though for many business systems in the United States with which humans interact, the off-peak occurs at night and in the early morning hours when most of the population is asleep. It may be useful to know the performance of a software system when the off-peak period is occurring; large batch jobs that might slow other, more time-critical transactions can be run. Also, if it is known when a system is being used the least, hardware components may be taken offline for repair, or software updates and reinstallations can be undertaken with minimal interruption. The counterpart to the off-peak-period performance NFR is the peak-period performance NFR (see also).

Because this requirement concerns a particular time that a system's performance is measured, the off-peak-period performance NFR can be found in two places in the UML hierarchy: in the Performance category, in both the HardwareTimePerformance and SoftwareTimePerformance subcategories.

Example: During off-peak periods of operation, the ULMTS system shall perform no fewer than 50 transactions (database lookups) per second.

operability – This requirement describes the ease of operation of a program by users[MCC77, 113]. Operability is almost synonymous with the "usability" NFR (see usability), except the connotation of usability relates more to a system's user-friendliness whereas operability describes a slightly more specialized meaning, a "quality metric that can be used to measure those operational characteristics of the software that provide useful inputs and outputs that can be assimilated" [BOW85, 87].

The operability requirement falls into two categories in the UML hierarchy: EndUser and Testing. The justification for inclusion within the EndUser category is that it is the end users who are subjected to a system's operational features throughout its lifetime. The operability requirement relates to testability in that "the better it works, the more efficiently it can be tested." Pressman [PRE01, 276] offers three examples:

*The system has few bugs (bugs add analysis and reporting overhead to the test process).

*No bugs block the execution of tests.

*The product evolves in functional stages (allows simultaneous development and testing).

Example: The end users who are subjected to the ULMTS system's operational features throughout its lifetime shall experience an acceptable level of ease of use when interacting with the system.

operating cost – The operating cost requirement describes the monetary amount of operating a software system once it has been delivered to the customer and deployed. Operating costs include both the cost of system maintenance (such as error correction and system upgrades) and execution costs; it does not include any costs associated with retiring or replacing a system (see also the replaceability and retirement cost requirements).

Because this requirement concerns the cost of operating a software system after it has been delivered to the customer and installed, the operating cost NFR may be found in the Cost category of the UML hierarchy, in the CostPostdeployment sub-category, in the MaintenanceCost sub-subcategory.

Example: The total cost to the University of operating ULMTS for a ten-year period following its deployment shall not exceed $5,000 (note: the total cost of ULMTS to the University shall not exceed $875,000).

peak-period performance – The peak-period performance requirement describes the performance of a software system during the time interval that it is being most heavily used. Systems in different domains have different peak periods; for example, a Wall Street stock trading software system may have its heaviest usage for the first ten minutes after the opening bell, a drive-through ATM may have its heaviest usage for the hour during which workers take their lunch breaks, or a sporting event ticket-selling system may have its peak period once a year when it is determined which two teams will meet in the annual playoffs. The counterpart to the peak-period performance NFR is the off-peak- period performance NFR (see also).

Because this requirement concerns a particular time that a system's performance is measured, the peak-period performance NFR can be found in two places in the UML hierarchy: in the Performance category, both in the HardwareTimePerformance and SoftwareTimePerformance subcategories.

Example: During peak periods of operation, the ULMTS system shall perform no fewer than 150 transactions (database lookups) per second.

performability – The performability requirement is synonymous with performance (see the performance NFR).

performance – The performance requirement is a "measure of process speed, response time, resource consumption, throughput, and efficiency" [MCC77, 104]. Performance is a very broad non-functional requirement; in fact, it is one of the top-level categories in the UML hierarchy. The numerous subcategories are first divided into hardware performance and software performance; these in turn have been further categorized into time performance and space performance. The HardwareTime sub-category contains requirements such as peak period performance, off-peak period performance, and throughput. The HardwareSpace sub-category contains requirements such as main memory performance and secondary-storage performance. The SoftwareTime sub-category contains requirements such as efficiency and time performance. The SoftwareSpace sub-category contains requirements such as buffer space performance, code space performance, and data space performance.
Example: The ULMTS system shall exhibit the process speed, response time, resource consumption, throughput, and efficiency as described in section 3.1.7.3 of the SRS.

planning cost – As its name suggests, the planning cost requirement describes the monetary amount of all activities undertaken in the planning phase of software system development. Planning entails all of the tasks required to "define resources, timelines, and other project-related information" [PRE01, 161]. Planning is crucial for any software project because it produces a general schedule for all work that is to be completed in the future. Planning allows developers to identify and avoid any potential problems that might arise, thus preventing wasteful "dead-ends", unforeseen component conflicts, and missed deadlines.

Developers plan not only the design of the finished product, but also system prototypes (themselves a form of a "plan") and testing. By planning the complexity and functionality prototypes, developers can more economically demonstrate to the customer the system that is being built without incurring the costs of erroneously writing code that implements specifications the customer did not intend. By planning the tests that are to be performed on a software system, developers may more efficiently identify and correct bugs and performance problems without wasting money conducting redundant and unnecessary tests.

Because this requirement describes the cost of planning the activities that will be performed before a system is built and delivered to the customer, the planning cost NFR may be found in the Cost category of the UML hierarchy, in the CostPredeployment sub-category.
Example: The cost of all activities associated with the planning phase of ULMTS to the University shall not exceed $15,000 (note: the total cost of ULMTS to the University shall not exceed $875,000).

planning time – As its name suggests, the planning time requirement describes the length of time to perform all activities undertaken in the planning phase of software system development (for a definition of planning, see the planning cost requirement). If a project has a strict timeline it must follow, this requirement is very important. Spending time planning before design and implementation begins allows developers to more clearly understand the scope of the project, and allows them to structure a reasonably accurate timeline which they may follow throughout the development phase. Failure to plan makes it impossible to know how long any of the development tasks will take; therefore the project's deadline will almost certainly be missed.

Because this requirement describes the amount of time that software system developers spend planning activities that will be performed before a system is built and delivered to the customer, the planning time NFR may be found in the Time category of the UML hierarchy, in the TimePredeployment sub-category, in the Developer sub-subcategory.
Example: The planning phase of ULMTS shall not exceed 45 days (note: the total length of the development of ULMTS shall not exceed 240 days).

plasticity – The plasticity requirement refers to how "plastic" (changeable) both the SRS and the final, deployed software system are. An SRS is plastic if its structure and style are such that any changes can be made easily, completely, and consistently" [IEE84, 61] (see also the modifiability requirement). Davis lists two reasons an SRS should be modifiable: first, a customer's needs are always evolving, and second, the SRS, like all complex software-related documents, will contain errors [DAV93, 188]. The SRS will change as new needs are discovered, old needs must be altered, or if obsolete needs must be abandoned.

In regards to the plasticity of the software system as a whole, making modifications is related to the degrees of cohesion exhibited by its components and the degree of coupling existent between components [YOU79, 93] (see also the cohesiveness and coupling requirements).

As it relates to the changeability of a system's requirements as the customer's needs evolve, the plasticity NFR can be found in the SRS category of the UML hierarchy. As it relates to the changeability of the system itself, the plasticity requirement can be found in the Future category, FutureChange sub-category.
Example: – Changes in user needs shall allow changes to the functionality of the ULMTS possible.

portability – The portability requirement describes "the ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, and replaceability" [IEE93, 61].
Portability will be an important NFR if the customer knows that before it is retired, the system will at some point have to be transferred to another environment. The new environment may be a different hardware configuration or another platform/ operating system.

Two closely related requirements to portability are mobility and nomadicity (see mobility, nomadicity). Mobility refers to the ease of moving the system, but unlike portability, it has no implication that there will be environmental changes that the developers of the software system will have to take into account when designing and implementing it. Nomadicity is a synonymous requirement for mobility.

The portability requirement may be found in the Future category of the UML hierarchy, FutureOperation subcategory since the ability to move the system to different operating environments assumes the system has been deployed in an initial environment to begin with.
Example: Since the ULMTS software will run on the library's existing hardware and not be ported to other platforms, it is not necessary that the software be independent of any environmental constraints.

precision – The precision requirement may describe both the SRS and the finished software system.

An SRS is precise "if and only if (a) numeric quantities are used whenever possible, and (b) the appropriate levels of precision are used for all numeric qualities" [DAV93, 22]. For example, "The system shall process one hundred database look-ups per second" is a precise requirement; "The database will perform database look-ups" is not precise. Many times the exact number of tasks a system is to perform per unit of time or how quickly one task will take is not known; in these cases full precision is not possible to incorporate into these requirements.

As it relates to software systems, precision is defined as "a measure of the ability to distinguish between nearly equal values; for example, four-place numerals are less precise than six-place numerals; nevertheless, a properly computed four-place numeral may be more accurate than an improperly computed six-place numeral" [IEE83, 51]. In computer languages, precision almost always refers to the manipulation and comparison of floating-point numbers. Precision is nearly synonymous with the accuracy requirement (see also accuracy).

The precision requirement may be found in two places in the UML hierarchy: first, in the SRS category, and second, in the Future category, FutureOperation sub-category.
Example: The ULMTS software shall be expected to perform its intended function with a MTBF (mean time between failures) of thirty days.

predictability – The predictability requirement describes the accuracy that the testers of a system may have when describing the results of a test before it is actually conducted. Predictability is a good indicator of system correctness, since if testers can accurately predict a software system's behavior, it is likely that its functionality has been correctly implemented. A related requirement is repeatability (see the repeatability NFR). If a particular test case is repeated a number of times and it yields the same result after each test, the developers can be confident that conducting the test one more time will again yield the same result; therefore, the level of predictability is high.

Because this requirement concerns the accuracy of anticipating a system's behavior during the testing phase, the predictability NFR may be found in the Testing category of the UML hierarchy.
Example: During the testing phase of the development of the ULMTS system, testers shall be able to repeat a particular test three times, and the test shall yield the same result after each test, enabling testers to predict the outcome of each test.

process management time – This requirement describes the efforts of the system developers during the design and implementation phases of the software project before the system is delivered to the customer. The term *process* is used in a variety of contexts in software engineering, but in terms of this requirement may be defined as the methods and activities the developing organization gets its work done. Some organizations actually have process groups, software engineers and computer specialists who are concerned with the development process. The process group's typical functions include "defining and documenting the process, establishing and defining metrics, gathering data, assisting projects in analyzing data, and advising management on areas requiring further attention" [THA97, 111].

Because this NFR refers to the time spent by the developers on the system before it is delivered to the customer and deployed, the process management time requirement may be found in the Time category of the UML hierarchy, in the TimePredeployment sub-category, in the Developer sub-subcategory.

Example: It shall take no longer than 30 days for the ULMTS developers to accurately determine the amount of time each phase of system development will take.

productivity – This requirement refers to both the productivity of system developers as they build the project and the productivity of the end users.

Pressman writes "when reusable components are applied throughout the software process, less time is spent creating the plans, models, documents, code, and data that are required to create a deliverable system;" as a result, the productivity of the developers has increased since the same level of functionality is delivered to the customer with less input effort [PRE01, 227] (for a discussion of what makes components reusable, see the reusability requirement).

The productivity of end-users is affected by several factors. Systems with high levels of learnability and trainability allow the users to get more information about operating the system. Software that is modifiable and customizable allows users to change the software to reflect their personal styles of working. Systems with a high level of transparency allow users to focus on their work without seeing and being distracted by the technical details of how the system operates (see also the learnability, trainability, modifiability, customizability, and transparency requirements).

Because this requirement concerns both how the system is designed and the end-users that ultimately use the system to do work, the productivity NFR may be found in two places in the UML hierarchy: in the Design category and in the EndUser category.
Example: The ULMTS developers shall use reusable components throughout the software lifecycle to increase productivity, in that less time will be spent creating the plans, models, documents, code, and data.

project stability – The project stability requirement describes the degree to which the software project will change as the software system is built during the development phase. A project may be broadly defined as "a temporary activity characterized by having a start date, specific objectives and constraints, established responsibilities, a budget and a schedule, and a completion date" [THA97, 114].

It is inevitable that changes will occur as development progresses; instead of resisting these changes, a reasonable response would be to try to manage it, thereby increasing stability. If project changes are carefully recorded and analyzed, their impact on the overall project will be predictable and will not disrupt development and the ultimate completion of the project (see also the impact analysis requirement). Pressman lists four features of stability:

*"Changes to the software are infrequent.
* Changes to the software are controlled.
* Changes to the software do not invalidate existing tests.
* The software recovers well from failures." [PRE01, 164]

The project stability requirement, since it describes the degree of change or volatility of the whole software project during the development phase, can be found in the Time category of the UML hierarchy, in the TimePredeployment sub-category, in the Developer sub-subcategory.
Example: The ULMTS system development start date, schedule, and completion date shall not stray more than 10 days from the original estimate; he ULMTS system development budget shall not stray more than three percent from the original estimate.

project tracking cost – As its name suggests, this requirement describes the cost of tracking the software project during the development phase. (for a definition of *project*, see the project stability requirement) It is imperative all the tasks that comprise the whole project be tracked, since errors and changes have the potential to cause a ripple effect, possibly delaying completion of the project and causing budget overruns. Typically, the
project manager oversees project tracking, though every developer should be aware of changes made in other developers' areas of the work since change in one part of a system usually affects other parts.

Pressman lists six ways tracking can be accomplished:

*Conduct periodic project status meetings in which each team member reports progress and problems.

*Evaluate the results of all reviews conducted throughout the software engineering process.

*Determine whether formal project milestones have been accomplished by the scheduled date.

*Compare the actual start-date to the planned start-date for each project task listed in the resource table.

*Meet informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.

*Use earned value analysis to assess progress quantitatively. [PRE01, 229].

Because this requirement concerns the cost of tracking a project during the software development phase before the system is released tot he customer, the project tracking cost NFR can be found in the Cost category of the UML hierarchy, in the CostPredeployment sub-category.
Example: The total cost of all project tracking-related tasks of the ULMTS system shall not exceed $4,000 (note: the total cost of ULMTS to the University shall not exceed $875,000).

promptness – The promptness requirement describes the punctuality of the developers; more specifically, it describes the interval of time between the moment the first phase of software development begins and the delivery of the final software system to the customer. This time interval entails eliciting requirements from the customer, analyzing and clarifying specifications, design, constructing prototypes, eliciting customer feedback, coding, testing, debugging, and performing customer acceptance testing. A related requirement to promptness is timeliness, which describes the degree of appropriateness of the software to its intended environment when it is delivered (see also the timeliness requirement).

Because this requirement concerns the sum time of all development activities, the promptness NFR could have been placed in the Design and Testing categories of the UML hierarchy, but for simplicity's sake, it can be found in the Time category, under the TimePredeployment sub-category, in the Developer sub-subcategory.
Example: The interval of time between the moment the first phase of software development begins and the delivery of the final software system to the customer shall not exceed eight months.

prototyping cost – As its name suggests, the prototyping cost requirement describes the cost to produce a prototype during the design phase of system development. A prototype is a model of the software built, constructed for customer and developer assessment. Two main paradigms of prototyping are *throwaway prototyping* and *evolutionary prototyping* [PRE01, 213]. A throwaway prototype is constructed as a rough demonstration of the system requirements then discarded, and an evolutionary prototype is build and added upon, eventually "evolving" into the finished product. Neither of the two prototype paradigms is more cost-effective than the other in all situations; there are a number of factors that determine which paradigm is more appropriate for system development, such as application complexity, the application domain, and whether the customer is certain of basic system requirements [PRE01, 213].

Because this requirement concerns both cost and overall system design, the prototyping cost non-functional requirement can be found in two places in the UML hierarchy: first, in the Cost category, under the CostPredeployment sub-category, and second, in the Design category.
Example: The total cost of creating a working prototype of the ULMTS system shall not exceed $2,000 (note: the total cost of ULMTS to the University shall not exceed $875,000).

prototyping time – As its name suggests, the prototyping time requirement describes the length of time to produce a prototype during the design phase of system development (for the definition of a prototype, see the prototyping cost requirement). As is the case for determining the cost of prototyping a system, no one paradigm (throwaway prototype versus evolutionary prototype) is the best choice for all systems. If a throwaway prototype is to be used, the length of development time must include not only the time required to construct the initial prototype, but must also include the "reconstruction" of what that prototype represented after any design changes have been made. An evolutionary prototype, in some cases, may save some development time since the prototype will eventually "evolve" into the final product.

Because this requirement concerns both prototyping time and overall system design, the prototyping time NFR can be found in two places in the UML hierarchy: first, in the Time category, under the TimePredeployment sub-category, in the Developer sub-subcategory; and second, in the Design category.
Example: The time required to construct a working prototype of the ULMTS system shall not exceed 30 days (note: the total length of the development of ULMTS shall not exceed 240 days).

reconfigurability – The reconfigurability requirement describes "the ability to organize and control elements of the software configuration" [MCC77, 183]. A system's software configuration is defined as "the items that comprise all information produced as part of the software process" [PRE01, 249]. Each of these items is referred to as a software configuration item (SCI), and as the software process progresses, the number of SCIs quickly grows. The branch of software engineering that audits, accounts for and controls these SCIs is called configuration management, "the discipline of identifying the configuration of a hardware/software system at discrete points in time with the purpose of systematically controlling changes to the configuration and maintaining the integrity and traceability of the configuration throughout the system lifecycle" [THA97, 401]. (see also the integrity and traceability requirements).

Because the reconfigurability requirements are handled by the developers during the design and implementation of the system, this NFR may be found in the Design category of the UML hierarchy (see also the configurability requirement).

Example: The ULMTS developers shall be able to control changes to the configuration of the hardware/software system at discrete points in time to maintain the integrity and traceability of the configuration throughout the system's lifecycle.

recoverability – The recoverability requirement describes the extent to which a software system can recover gracefully after a failure occurs. This NFR is almost synonymous with the recovery requirement (see also the recovery requirement), except that it denotes a measure of the ability a given system is able to recover from a fault or error; recoverability is a metric of the degree to which a system may recover.

Because this requirement describes the ability of an application to resume normal operation after a fault or error has occurred, the recoverability NFR may be found in the SystemIntegrity category of the UML hierarchy, in the Error sub-category.
Example: The ULMTS system shall take no longer than three hours to return to normal operation after one of the faults listed and described in the SRS, section 4.1.2.2 occurs.

recovery – A system is said to *recover* when a fault occurs and it resumes processing. The recovery requirement describes the ability of a software system to return to normal operation after one of these faults occurs. In some cases, requirements dictate that the system must recover within a specified period of time. Recovery is especially important in applications where errors and faults would translate directly into safety hazards and loss of human life, such as in medical and health monitoring systems, weapons-guidance, and radioactive materials handling.

Because this requirement describes the ability of an application to resume normal operation after a fault or error has occurred, the recovery NFR may be found in the SystemIntegrity category of the UML hierarchy, in the Error sub-category.
Example: The ULMTS system shall take no longer than three hours to return to normal operation after one of the faults listed and described in the SRS, section 4.1.2.2 occurs.

reengineering cost – As its name suggests, the reengineering cost requirement describes the monetary amount of performing reengineering on a software system after it has been deployed and is in use. The motivation for reengineering comes from the desire to change one's current system as opposed to retiring the system and building a completely different one (see the retirement cost requirement). Reengineering involves adding functionality, improving performance and reliability, and enhancing maintainability [PRE01, 76]. Software reengineering is important because the environments in which software exists are constantly changing; these systems must also evolve to reflect these changes.

Because this requirement describes the cost of changing a software system after it has been put into use, the reengineering cost NFR may be found in the Cost category of the UML hierarchy, in the CostPostdeployment sub-category, in the Redo sub-subcategory.
Example: The total cost of reengineering the ULMTS system (in the form of adding functionality, improving performance and reliability, and enhancing maintainability) shall not exceed $8,500 (note: the total cost of ULMTS to the University shall not exceed $875,000).

reliability – The reliability requirement is defined as "the extent to which a program can be expected to perform its intended function with required precision" [MCC77, 134]. (see also the precision NFR) Reliability is one of the most important non-functional requirements: if a software system cannot perform its functionality without experiencing failure, the developers have failed to fulfill their responsibilities as enumerated in the SRS. A software system that is not reliable to an acceptable degree (as defined by the customer) is virtually worthless.

Pressman provides a measure of reliability as expressed in terms of mean-time-between-failure (MTBF):

MTBF = MTTF + MTTR

where the acronyms MTTF and MTTR stand for mean-time-to-failure and mean-time-to-repair, respectively [PRE01, 287].

Because the reliability requirement describes the lack of failure shown by a system that has already been deployed, it may be found in the Future category of the UML hierarchy, in the FutureOperation sub-category.
Example: The ULMTS software shall be expected to perform its intended function with a MTBF (mean time between failures) of thirty days.

repeatability – The predictability requirement describes the accuracy of a system while it is undergoing testing. If a test case is run and it yields a particular result, the same result should occur every time the same test is repeated. Repeatability is a good indicator of system correctness, but it is not foolproof; an incorrectly implemented specification may still yield the same result after a number of repeated tests. A related requirement is predictability (see the predictability NFR). If one test is repeated multiple times, the probability that it will yield the same result again means there is a high level of predictability.

Because this requirement concerns a software system's behavior during the testing phase, the predictability NFR may be found in the Testing category of the UML hierarchy.
Example: During the testing phase of the development of the ULMTS system, testers shall be able to repeat a particular test three times, and the test shall yield the same result after each test.

replaceability – The replaceability requirement describes the ease with which either a software component or an entire system may be replaced with another. Concerning the replaceability of individual components, the fact that the environment in which a software system resides is always changing is justification for replacing modules on an as-needed basis; this constant maintenance and evolving of a system is called reengineering (for a more thorough discussion of reengineering, see also the reengineering cost requirement). A system's modules may be more easily replaced if they have a high level of cohesion and a low level of coupling (see also the cohesiveness and coupling requirement).

The replaceability of an entire system describes the ease with which a system may be supplanted with another. This replacement is also related to retirement, in which useful parts (such as data) of a system that are no longer going to be used are salvaged so they may be added to the system that is replacing it (for a more thorough discussion of system retirement, see also the retirement cost requirement).

Because this requirement concerns both system components and entire systems that are already in use, the replaceability NFR may be found in two places in the UML hierarchy: first, in the Design category, and second, in the Future category, in the FutureMaintenance sub-category.

Example: The total cost (to the University) of replacing the ULMTS system shall not exceed $17,750 (note: the total cost of UMLTS shall not exceed $875,000).

replicability – The replicability requirement describes the ease a system or one of its components may be duplicated or copied. Generally, whole systems are not copied, but frequently their components are duplicated by the developers. This replication takes place when, in object-oriented languages such as C++ and Java, classes have been written with a high level of reuse in mind. For example, the same Employee class may be used both in a payroll program and in a sales program (for a more thorough discussion of component reuse, see also the reusability requirement). In this example, the replicability would simply be the duplication of the Employee class.

Because this requirement concerns the copying and reuse of a system's individual components, the replicability NFR may be found in the Design category of the UML hierarchy.
Example: Since the ULMTS system will be used only by the main campus library, it is not necessary that any components need to be duplicated or copied.

response time – The response time requirement describes the "time from the submission of a user request until the first response is produced" [SIL99, 33]. Response time is important in interactive systems where processes can produce some output fairly early, and continue computing new results while previous results are being output to the user [SIL99, 33]. A system's CPU(s) is largely responsible for the length of response time, but a system's output devices may also limit the speed of responses.

Because this requirement concerns the time performance of one or more hardware elements of a system, the response time NFR may be found in the Performance category of the UML hierarchy, in the HardwarePerformance sub-category, in the HardwareTimePerformance sub-subcategory.
Example: The time from the submission of a ULMTS user request until the first response is produced shall take no longer than three seconds.

responsiveness – The responsiveness requirement describes the degree to which users of a software system can "do what they want, when they want, in a way that is clear" [KIT95, 130]. Thus, this NFR concerns a system's functionality. If users have been trained how to use a given system and have learned most of its capabilities (see the trainability and learnability requirements), they should have a general idea of what can and cannot be accomplished using the system. However, no user can be expected to memorize every single function a system may perform; a degree of intuition is at work whenever a human user is working with software. The availability of help files may inform the user about unfamiliar system features, but sometimes a user will simply attempt to "do something" without taking time to consult documentation. It is in these cases that user actions reveal a system's responsiveness to high or low.

Because this requirement describes the ease with which a user may accomplish an intended task, the responsiveness NFR may be found in the EndUser category of the UML hierarchy.
Example: The time from the submission of a ULMTS user request until the first response is produced shall take no longer than three seconds.

retirement cost – As its name suggests, the retirement cost describes the monetary amount of retiring a software system. When the cost of maintaining and reengineering a system becomes too expensive and troublesome, it may be retired (see also the maintainability and reengineering cost requirements). However, retiring a system does not mean simply pulling the plug or deleting all its files. Certain data may be transferred to the newer system that is replacing the one being retired. For example, a large database of names and addresses that has been collected over a ten year span is far too valuable to throw away and start anew. In such a case, the cost of transferring the data to another system would be part of the cost of retiring the earlier system.

Because this requirement describes the costs associated with saving and transferring data from a system after it is no longer economical to use, the reengineering cost NFR may be found in the Cost category of the UML hierarchy, in the CostPostdeployment sub-category, in the Redo sub-subcategory.
Example: The total cost (to the University) of retiring the ULMTS system if (and only if) a replacement system is ready to be installed shall not exceed $2,000 (note: the total cost of UMLTS shall not exceed $875,000).

reusability – The reusability requirement may describe both an SRS and the components which comprise a software system.

An SRS is reusable "if and only if its sentences, paragraphs, and sections can be easily adopted or adapted for use in a subsequent SRS" [DAV93, 260]. Davis lists three techniques for making an SRS more reusable:

*Write the SRS using symbolic constants, then later systems with similar functionality can change the value in the symbolic constants.

*Use formal models.

*Create a library of *abstract requirements types*, generic requirements paragraphs that are instantiated by providing information about the customizable characteristics of a particular application [DAV93, 260].

The reusability requirement as it relates to systems describes "the extent to which a program (or parts of a program) can be reused in other applications – related to the packaging and scope of the functions that the program performs" [MCC77, 47]. If components are created with reusability in mind, developers may save money and time, not only in component creation time but also testing and debugging time. Because the impact of reuse on the software industry is enormous, a number of major companies and industry consortia have proposed standards for reusable component software, including OMG/CORBA, Microsoft COM, and Sun JavaBeans Components [PRE01, 312].

The reusability requirement can be found in two places in the hierarchy of NFRs: in the SRS category and the Design category.
Example: The components that make up the ULMTS (classes) shall be written in such a manner that they may be reused when similar functionality is required (see the code reuse section of the SRS, 2.4.1.1).

risk analysis cost – As its name suggests, the risk analysis cost requirement describes the monetary amount of performing risk analysis before a software system is built. A risk is a problem that might or might not arise during system development. Regardless of the type of problem, it is in the developers' and customers' best interest to "identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur" [PRE01, 291].

The cost of a project will almost certainly be affected by a number of potential risks. If customer requirements change, new prototypes may need to be built and new developers hired for the project. If development technologies change, a system's builders may need to buy or update current software tools and hardware configurations. If as many potential risks can be identified as possible and the likelihood of their occurrences ranked by probability, software engineers may significantly reduce development costs.

Because this requirement concerns risks (potential problems that may or may not arise) during the early stages of system development and the cost of these risks, the risk analysis cost NFR may be found in two places in the UML hierarchy: first, in the Design category; and second, in the Cost category, in the CostPredeployment sub-category.
Example: The total cost (to the University) of conducting risk analysis on the ULMTS system shall not exceed $13,000 (note: the total cost of UMLTS shall not exceed $875,000).

risk analysis time – As its name suggests, the risk analysis time requirement describes the time required to perform risk analysis before a software system is built (for definitions of risk and risk analysis, see the risk analysis cost requirement).

A good example of the ways time to conduct risk analysis can vary is to compare reactive and proactive risk strategies. Using a reactive risk strategy, members of a software development team do not allocate any time for predicting or assessing any potential risks. Members of a software development team that practice a proactive risk strategy allocate time before any technical work begins identify potential risks, assessing their probability and impact, and ranking their importance [PRE01, 149]. The first group has saved time in the short run by not conducting risk analysis, but when problems inevitably arise, it will be the second group who is more likely to deliver the final system on time since they had the foresight to spend time up front anticipating risk.

Because this requirement concerns risks (potential problems that may or may not arise) during the early stages of system development and the time required to identify and rank these risks, the risk analysis time NFR may be found in two places in the UML hierarchy: first, in the Design category; and second, in the Time category, in the TimePredeployment sub-category, in the Developer sub-subcategory.
Example: The length of time to conduct risk analysis of ULMTS shall not exceed 5 days (note: the total length of the development of ULMTS shall not exceed 240 days).

robustness – The robustness requirement describes the degree to which a program or system can recover gracefully whenever a failure occurs [PAG00, 224]. It also describes the time it takes the system to restart after experiencing system failure [KOT98, 118]. This requirement is closely related to the recovery and recoverability NFRs since it relates to the actions that must take place when a system fails. (see recovery and recoverability)

A counterpart to this requirement is fault-tolerance, which describes how resistant to fault a software system is. Fault is a general term meaning stoppage; it makes no distinction between errors and bugs in the software and malicious, security-related actions taken by hackers.

Because this requirement concerns the recovery of a system after a fault has taken place, the robustness NFR may be found in the SystemIntegrity category of the UML hierarchy, in the Error sub-category.
Example: The ULMTS will be robust to the degree that, upon encountering an incorrect step, process, data definition, or thrown exception, ULMTS will allow the user to revert to a previous state before failing gracefully.

safety – The safety requirement comprises the "assessment of potential hazards that may affect software negatively and cause an entire system to fail" [PRE01, 237]. Safety almost always refers to the protection of human life; in computer-based systems that, for example, control weapons guidance, radioactive materials handling, or space travel, safety should be one of the most thoroughly analyzed, planned, and tested non-functional requirements in the system. The safety requirement is closely related to reliability, but as Pressman writes, "it is important to understand the subtle differences between them. Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However, the occurrence of a failure does not necessarily result in a hazard or mishap. Software safety examines the ways in which failures result in conditions that can lead to a mishap. That is, failures are not considered in a vacuum, but are evaluated in the context of an entire computer-based system. [PRE01, 272]

Although all safety issues should be thoroughly examined and discussed in the design phase of any project, they manifest themselves after a system has been deployed and is in operation; as such, the safety NFR may be found in the Future category of the UML hierarchy, in the FutureOperation sub-category.
Example: No more than three (3) library patrons per day shall be maimed, critically injured, or killed when using ULMTS to search for library materials.

scalability – The scalability requirement refers to the ease with which a system may be made smaller or larger, although most of the time, *increasing* the system's size is the concern, not reducing it.

Both hardware and software systems may be scaled. A hardware system is said to be scalable if more components, such as CPUs, can be added to it, yielding more computing power. A software system is scaled "upward" (increases in size) when it achieves more functionality through the addition of more components, when it interacts with larger and larger databases, and when additional interconnections with information systems are implemented. Pressman offers a caveat: "As size and complexity grows, small changes can have far-reaching and unintended effects that can be problematic. Therefore, the rigor of configuration control mechanisms should be directly proportional to application scale" [PRE01, 266]. (see also the configurability requirement)

Because the scalability requirement implies alterations made to the size of the software system after it has been deployed, it may be found in the Future category of the UML hierarchy, FutureChange sub-category.
Example: Changes in user needs shall allow the functionality of the ULMTS to be increased for a period of seven years after its installation.

secondary-storage performance– As its name suggests, this requirement describes the performance of a system's secondary-storage, also called secondary memory (for a definition of memory, see the main memory performance requirement).

Secondary memories are used because often the main memory is too small. Some examples of secondary memory are magnetic disks, magnetic tape, and optical disks. There are two main differences between primary and secondary memories. First, the access time of primary memories is much faster than that of secondary memories, typically an order of magnitude faster. Another key difference is that of size: secondary memories are usually much bigger than primary memories. Performance in both types of memory is affected by the speed of the spinning disks and the rotational latency, "the delay until the desired sector rotated under the read/write heads" [TAN99, 130].

Because this requirement concerns the performance of secondary memory, the secondary-storage performance NFR can be found in the Performance category of the UML hierarchy, in the HardwarePerformance sub-category, in the HardwareSpacePerformance sub-subcategory.

Example: No lookups of data that reside on secondary storage (the primary hard disk) shall take longer than 50 milliseconds.

security – The security requirement describes the availability of mechanisms that control or protect programs and data [MCC77, 265]. It may also refer to the "probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled" [PRE01, 312].

As computers and software systems have taken on a much larger role in the storage and transacting of personal, financial, military and confidential data, security has become one of these system's most crucial attributes. In fact, security requirements comprise one of the three primary requirements in Chung, et. al.'s *Non-Functional Requirements in Software Engineering* [CHU99, 108] (along with accuracy and performance requirements). A software system that is not secure is virtually worthless, since unauthorized intrusion may allow data corruption, financial and informational theft, and even loss of human life.

Security is such an overarching topic in software engineering, it has been made a sub-category of one of the eleven top-level categories in the UML hierarchy, SystemIntegrity. For the sake of consistency, the security requirement can be found in the Security sub-category of SystemIntegrity.

Example: The sensitive user information (passwords, date of birth, addresses) stored in ULMTS shall be available only to the authorized system administrator(s).

sensitivity – The sensitivity requirement concerns input data during the testing phase of a software system. Pressman writes, "in some situations, (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing" [PRE01, 255].

Thus, a test that has a higher degree of sensitivity makes it more difficult to isolate the precise combinations of valid inputs that cause invalid results.

Because this requirement describes an activity performed when developers test a software system, the sensitivity NFR can be found in the Testing category of the UML hierarchy.
Example: During system testing, the ULMTS developers shall attempt to uncover data combinations within valid input classes that may cause instability or improper processing.

similarity – The similarity requirement describes how closely the components of a software system resemble one another. It can be useful for components, such as a system's classes to have a uniform structure. Some examples are using a naming convention for the names of all the classes, using only one type of notation for all of the variables (such as Hungarian notation), and listing all a class's public variables separately from the private ones. The similarity of components provides a uniform appearance that has several benefits. Developers may avoid naming conflicts if one naming convention is used. Organizing a class's variables and methods reduces the time looking for a member of a class in code. And by using a standard form of documentation on all projects, the developers may more easily grasp a large project than if documentation was different for every system built.

Because this requirement concerns the resemblance of a system's components, the similarity NFR may be found in the Design category of the UML hierarchy.
Example: To promote similarity between the components that comprise the ULMTS system, the developers shall use a uniform naming convention for the names of all the classes, use only one type of notation for all of the variables (Hungarian notation), and list every class's public variables separately from the private variables.

simplicity – The simplicity requirement describes "the degree to which a program can be understood without difficulty" [MCC77, 133]. This is closely related to the understandability requirement (see also). Pressman subdivides simplicity into three types: functional simplicity, structural simplicity, and code simplicity [PRE01, 122]. Functional simplicity requires the feature set (i.e., the sum of a system's functions) be as small as possible yet still meet all of its requirements. Structural simplicity requires a system's architecture be modularized so the propagation of faults may be limited (see also the modularity NFR). Code simplicity dictates a standard by which the code is written so it can be easily inspected and maintained (see also the conciseness NFR).

The simplicity requirement may also relate to the user interface, particularly as it contains the minimum number of components while still allowing the end user to perform all necessary tasks specified in the SRS. In the UML hierarchy, the simplicity requirement can be found in two places: the UserInterface category and the Testing category.
Example: To promote system simplicity, the ULMTS shall contain no extraneous functionality not required by the end users.

software cost – As its name suggest, the software cost requirement describes the monetary amount of software, a "sequence of instructions suitable for processing by a computer; processing may include the use of an assembler, a compiler, an interpreter, or a translator to prepare the program for execution, as well as to execute it" [IEE83, 272]. Since software is an intangible product (contrast this with the hardware cost requirement), its cost is comprised of all the efforts of the developers who built it, including requirements elicitation, design, prototyping, coding, testing, and debugging.

Since this requirement describes the monetary amount of a system to be built then delivered to the customer, the software cost NFR can be found in the Cost category of the UML hierarchy, in the CostPredeployment sub-category.
Example: The total cost of the software deliverables of ULMTS to the University shall not exceed $275,000 (note: the total cost of UMLTS shall not exceed $875,000).

software production time – This very broad requirement describes the interval of time between the moment the first phase of software development begins and the delivery of the final software system to the customer (for a definition of software, see the software cost requirement). *Production* is an overarching term; it may entail eliciting requirements from the customer, analyzing and clarifying specifications, design, constructing prototypes, eliciting customer feedback, coding, testing, debugging, and performing customer acceptance testing.

Because this requirement concerns the sum time of all development activities, it could have been placed in the Design and Testing categories of the UML hierarchy, but for simplicity's sake, it can be found in the Time category, under the TimePredeployment sub-category, in the Developer sub-subcategory.
Example: The length of time to produce the ULMTS software shall not exceed 7 days (note: the total length of the development of ULMTS shall not exceed 240 days).

space boundedness – The space boundedness requirement is a synonym for the total size of available memory. In the early days of computing, memory was expensive, so a system's space boundedness was an important factor to take into account when designing software. As time passed, memory became cheaper, lessening the importance of this requirement. Today, most customers' memory needs are easily met. It is only in areas of high memory use (such as streaming video and audio) where a system's space boundedness would need to be considered during the design phase.

Because this requirement concerns the size of a software system's available memory, the space boundedness NFR may be found in the Design category of the UML hierarchy.
Example: The ULMTS's total size of available memory shall be 1024K RAM and 80 gigabytes of secondary storage.

space performance – As its name suggests, the space performance requirement describes the performance of a software system as it relates to space. Space is a synonym for the amount of memory available to a computer or an entire software system.

There are at least two notions of space as they relate to memory. First, the size of each addressable location (sometimes referred to as "cells") in memory may be of varied size; this size may affect a system's space performance. For example, a system that uses sixteen bit memory addresses may store data items that are twice as large as one that uses eight-bit cells. Secondly, the total size of primary memory may affect space performance.
If a system has a small memory, performance may suffer if attempts to store data that exceeds the amount of available memory fails.

Because this requirement concerns the measurement of a system's performance as it relates to space, the space performance NFR can be found in two places in the UML hierarchy: in the Performance category, both in the HardwareSpacePerformance and SoftwareSpacePerformance sub-categories.

Example: The ULMTS system shall use sixteen bit memory addresses to facilitate 10,000 word accesses per second.

specificity – The specificity requirement refers to the application of program components to particular functionality. This requirement may be more clearly understood by looking at its counterpart NFR: generality, which describes the degree to which components (such as structs and classes) may be reused in other systems.

Just the opposite of generality, a component that is specific has little capacity for reuse and application in software systems other than the one it was designed and implemented for. A requirement with a high degree of specificity is not necessarily undesirable; it is just that there is no time saved because the component is not reused in other systems.

Because this NFR refers to the degree of potential application of the components that comprise a given software system, the specificity requirement can be found in the Design category of the UML hierarchy.

Example: To promote component generality, no specificity in the structure of the ULMTS's modules (classes) is necessary.

stability – Stability describes the degree to which requirements will change as the software system is built during the development phase. If it is possible to rank and order the SRS's requirements based on their likelihood of changing or not changing, it may be helpful for developers to annotate the SRS by relative stability so that they may be able to "determine where to build in flexibility" in the system [DAV93, 104]. A requirement may be considered stable if "the needs it addresses will not change in the expected life of the software [IEE93, 33].

Pressman lists four features of stability as they affect system testing:

*"Changes to the software are infrequent.

* Changes to the software are controlled.

* Changes to the software do not invalidate existing tests.

* The software recovers well from failures." [PRE01, 217]

The stability requirement, since it describes the degree of change or volatility of the requirements themselves during the development phase, can be found in the SRS category of the UML hierarchy. As it relates to testing, the stability requirement also falls into the Testing category.

Example: To promote stability during the development of the ULMTS,
changes to the system software shall be infrequent and controlled.

standardizability – The standardizability requirement describes both the process and the product of developers' work. The process standard "defines the procedures or operations used in making or achieving a product;" the product standard "defines what constitutes completeness and acceptability of items that are produced as a result of a process" [THA97, 186]. Concerning the development of a particular system, standardizability is of interest to the customer only in that if the developers' process has been standardized, there are a set of descriptions of what constitutes quality, and the probability of unforeseen difficulties being encountered is minimized since it is implied that the developers have used the process before with success.

Since this requirement deals with the process and product of the development phase of the system, the standardizability NFR may be found in the Design category of the UML hierarchy.
Example: Since the customer of the ULMTS system (university officials) has no interest in whether the developers' process has been standardized, standardizability will not be required.

subjectivity – The subjectivity requirement relates to an end user's individual perception of a software system. The perception one holds may be affected by many other attributes of a system: accuracy, compatibility, dependability, efficiency, intuitiveness, learnability, performance, reliability, safety, security, user-friendliness, and versatility. It is difficult to accurately quantify the perceptions of an organization, since one system may be used by many (hundreds or thousands) of users. Subjectivity is also difficult to measure because the users are human, and their perceptions are likely to change from day to day. Subjectivity may be measured by personal interviews, which is time-consuming, or questionnaires, which may not accurately quantify the user's emotions if all system users do not complete and return them.

Because this requirement concerns the personal perceptions that users have toward a software system, the subjectivity NFR may be found in the EndUser category of the UML hierarchy.
Example: Since it is difficult to accurately quantify the dozens of ULMTS users' perceptions, no requirement will be made on the system's subjectivity.

supportability – The supportability requirement "combines the ability to extend the program (extensibility), adaptability, and serviceability – these three attributes represent a more common term, maintainability – in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration), the ease with which a system can be installed, and the ease with which problems can be localized" [MCC77, 141].

Although the supportability plays a part in the testing phase and interoperation with other systems, the majority of the work done to support a system is done after it has been deployed; for this reason, the supportability requirement can be found in the Future category of the UML hierarchy, in the FutureMaintenance sub-category.
Example: The ongoing operation of the ULMTS shall be supported by the University library system administrator at a time cost of no more than two hours per work week.

surety – The surety requirement describes the confidence a customer has in a software system. Surety may be increased if the customer plays a large role in system development, giving the developers feedback as the project progresses, test-driving the system prototypes, and performing acceptance testing. Confidence in a system may be affected by many other attributes: accuracy, compatibility, dependability, efficiency, intuitiveness, learnability, performance, reliability, safety, security, user-friendliness, and versatility. Confidence is difficult to measure because users are human, and their perceptions are subjective and are likely to change from day to day (see also the subjectivity NFR). Surety may be measured by personal interviews, which is time-consuming, or questionnaires, which may not accurately quantify the organization's emotions if all system users do not complete and return them.

Because this requirement describes the confidence that users have toward a software system, the surety NFR may be found in the EndUser category of the UML hierarchy.

Example: Since the customer of the ULMTS (university officials) will play a large role in system development (by giving developers feedback as the project progresses, test-driving the system prototypes, and performing acceptance testing), end-user surety will be achieved with no further effort.

survivability – The survivability requirement describes "how well the software will perform and support critical functions without failures, within a specified time period, when a portion of the system is inoperable" [IEE83, 66]. This definition implies that the system is modular; when one module is not functioning correctly, other modules are (see also the modularity requirement).

There are two strategies for giving a software system a high degree of survivability. First, the more independent a system's components are, the less their correct operation depends on other components. This component independence leads to a higher degree of survivability of the system as a whole. Second, by using data redundancy, a system is more survivable since if critical data is lost in one area, there is an identical set of the data in another, non-disabled area of the system.

Because this requirement concerns a system's ability to withstand failure after it has been deployed, the survivability NFR may be found in the SystemIntegrity category in the UML hierarchy, in the Error sub-category.
Example: The ULMTS shall be survivable to the degree that, upon encountering an incorrect step, process, data definition, or thrown exception, ULMTS will allow a user to revert to a previous state before failing gracefully.

susceptibility – The susceptibility requirement describes how vulnerable a system is to unauthorized access by malicious users (hackers). Susceptibility is indirectly proportional to the security requirement (see security) in that a highly secure system will have a low level of susceptibility. As computers and software systems have taken on a much larger role in the storage and transacting of personal, financial, military and confidential data, security has become a crucial attribute. A software system that has a high degree of susceptibility is virtually worthless, since unauthorized intrusion may allow data corruption, financial and informational theft, and even loss of human life.

Because this requirement concerns a system's vulnerability to malicious use, the susceptibility NFR may be found in the SystemIntegrity category of the UML hierarchy, in the Security sub-category.
Example: Access to all ULMTS software and data by unauthorized persons can be controlled by using testing metrics written by testers intentionally trying to compromise the system as described in the testing section of the SRS (5.1.1.4).

sustainability – The sustainability requirement describes the effort to keep a system functioning after it has been delivered to the customer and deployed. The sustainability NFR is closely related to maintainability, which describes "the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements" [PRE01, 417] (see also the maintainability requirement). Software needs to be sustained because the environment in which a system exists is constantly changing, which necessitates alterations be made to that system to insure it remains accurate and dependable (see also the accuracy and dependability requirements).

Because this requirement describes the effort expended to keep a software system functioning after it has been deployed, the sustainability NFR can be found in the Future category of the UML hierarchy, in the FutureMaintenance sub-category.
Example: The ULMTS shall have the capacity to have any encountered errors corrected, be adapted if its environment changes, or enhanced if the customer desires a change in requirements for a period of two years after installation.

testability – Testability describes the effort required to test a program to ensure that it performs its intended function [MCC77, 100]. This is one of the more important non-functional requirements, since testing a system is the essence of proving it meets its specification and requirements. Testability "occurs as a result of good design. Data design, architecture, interfaces, and component-level detail can either facilitate testing or make it difficult" [PRE01, 233]. Once the testing phase of a software project begins, developers design test cases that will demonstrate adequate coverage, which is to say, a sufficient number of execution paths through the software. Two common test case approaches are white-box testing and black-box testing. In white-box testing, the internals of the system's components are exposed to the testers; for example, all of a class's functions' implementations are visible. In black-box testing, the design details of the system components are invisible to testers; only the inputs to and outputs from the system components are of concern to the testers [PRE01, 233].

The testability requirement can be found in the Testing category of the UML hierarchy.
Example: ULMTS shall be testable during the testing phase of system development by performing the suite of test cases as provided by the independent testing group and as specified in the testing section of the SRS, section 5.1.2.1.

testing time – As its name suggests, this requirement describes the length of time it takes the developers to test a given software system before releasing it to the customer. The testing time is directly related to the testability NFR (see testability); the more testable a system is, the less time it will take to test. The testing time requirement is also directly related to the size of a software system. Obviously, the smaller a system is and the fewer the modules that comprise it, the easier it may be tested.

Another factor that affects the time required to test a system is the independence of its modules. If a system's components are not dependent upon one another, have a low degree of coupling (see the coupling requirement), and have a high degree of cohesion (see the cohesion requirement), they may be tested independently, decreasing the time to test the whole system in the process. (see testability)

Because this requirement concerns both testing and the length of time to complete one phase of the software development process (the testing phase), the testing time NFR can be found in two places in the UML hierarchy: first, in the Time category, in the TimePredeployment sub-category, in the Developer sub-subcategory; and second, in the Testing category.
Example: The testing phase of ULMTS shall not exceed 85 days (note: the total length of the development of ULMTS shall not exceed 240 days).

throughput – The throughput requirement describes the "number of processes the CPU (central processing unit) completes per time unit" [SIL99, 317]. A related requirement is response time (see the response time NFR) which describes the time from the submission of a user request until the first response is produced. if the CPU throughput is maximized, a system's response time is minimized, as well as turnaround time (the interval from the time of submission of a process to the time of completion) and waiting time (the sum of the periods that processes spend waiting in a CPU's ready queue) [SIL99, 154].

Because this requirement concerns the time performance of one or more hardware elements of a system, the throughput NFR may be found in the Performance category of the UML hierarchy, in the HardwarePerformance sub-category, in the HardwareTimePerformance sub-subcategory.
Example: The CPU of the ULMTS system shall have the capability to complete 10,000 processes per second.

time performance – As its name suggests, the time performance requirement describes the performance of a software system as it relates to time. This may include the time required for database lookups, reads and writes to secondary storage, throughput, processing, connecting to a network, or transmitting across a network.

Several factors may affect the time performance of a system. A system that uses a CPU (or multiple CPUs) that does not support multitasking will be slower than one that does. Many users simultaneously making data requests or transactions may slow down the total speed of the system. A smaller bandwidth translates into slower network connections than a system using a wider bandwidth.

Because this requirement concerns the measurement of a system's performance as it relates to time, the time performance NFR can be found in two places in the UML hierarchy: in the Performance category, both in the HardwareTimePerformance and SoftwareTimePerformance sub-categories.
Example: The ULMTS system shall exhibit the process speed, response time, throughput, and temporal efficiency as described in section 3.1.7.3 of the SRS.

timeliness – The timeliness requirement describes the degree of appropriateness of the software to its intended environment when it is delivered to the customer. Timeliness may be an important concern if the system's environment frequently changes. It may also be important to receive the software in a timely manner if it is to be used for a particular point in time, such as the much-anticipated Y2K transition. A related requirement to requirement is promptness; if it is necessary for a system to be delivered in a timely manner, the developers should not exceed the time budgeted for system development and miss any deadlines (see also the development time requirement).

Because this requirement concerns the point in time a system is transferred from its developers to its customers, the timeliness NFR may be found in the Time category of the UML hierarchy, in the TimePostDeployment sub-category.
Example: The interval of time between the moment the first phase of software development begins and the delivery of the final software system to the customer shall not exceed eight months despite any unforeseen changes in the users' environment.

tolerance – The tolerance requirement describes the damage that occurs when the program encounters an error [MCC77, 245]. Tolerance is a more generalized version of the fault-tolerance requirement (see also).

A system's tolerance is closely related to its survivability. Whereas tolerance describes the amount of damage a system can handle, survivability describes how well the software will perform after experiencing this damage.

A system may be designed to have a higher degree of tolerance. As with the survivability requirement, the more independent a software system's components are, the less their correct operation depends on other components. This component independence leads to a more tolerant system as a whole, since damage to one component is not likely to affect others.

Because this requirement concerns the damage that may affect a software system's ability to correctly perform its functionality, the tolerance NFR may be found in the SystemIntegrity category of the UML hierarchy, in the Error sub-category.
Example: The ULMTS will be tolerant to the degree that, upon encountering an incorrect step, process, data definition, or thrown exception, ULMTS will allow the user to revert to a previous state before failing gracefully.

traceability – The traceability requirement describes "the ability to trace a design representation or actual program component back to requirements" [MCC77, 208]. As such, traceability represents a path from the completed system backwards to the specifications, particularly the SRS.

Davis lists four techniques for achieving traceability:

*"Number every paragraph hierarchically. You can later refer to any requirement by a paragraph and sentence number, e.g., requirement 2.3.2.4s3 refers to the requirement in sentence 3 of paragraph 2.3.2.4.

*Number every paragraph hierarchically and include only one requirement in any paragraph. You can refer to any requirement by a paragraph number.

*Number every requirement with a unique number in parentheses immediately after the requirement.

*Use a convention for indicating a requirement, e.g., always use the word *shall* in a sentence containing a requirement; then use a simple *shall*-extraction tool to extract the number of all sentences with *shall*" [DAV93, 69].

Since the traceability NFR connects the finished system with the customer's original specifications, this requirement may be found in the SRS category of the UML hierarchy.
Example: Every feature of the ULMTS system shall be easily traceable back to a specific requirement in the Software Requirements Specification (SRS).

trainability – The trainability requirement describes the degree to which the software assists in enabling new users to apply the system [MCC77, 140]. Trainability is the counterpart to the learnability requirement (see also learnability). Whereas a system is learned, a user of that system is trained. Trainability is accomplished more efficiency based on two other NFRs: intuitiveness, learnability, operability, simplicity, understandability, and user-friendliness.

Because the trainability requirement refers to the system as it is learned by the end user, it can be found in the EndUser category of the UML hierarchy.
Example: Users new to the ULMTS system shall be able to effectively operate the system after a training period of two weeks, at three hours a day.

transferability – The transferability requirement describes the ease with which a software system may be moved from one hardware configuration to another. Transferability may be a desirable attribute if it is known that there will be a change in hardware at some point after the software has been deployed. Some factors that may affect a software system's transferability are memory requirements (code-space performance and data-space performance) and network connection requirements.

Because this requirement describes the effort expended to move a software system to a different hardware configuration after it has been deployed, the transferability NFR can be found in the Future category of the UML hierarchy, in the FutureMaintenance sub-category.
Example: Since the ULMTS software will run on the library's existing hardware and not be ported to other platforms, it is not necessary that the software be independent of any environmental constraints.

transparency – The term "transparency" in software engineering means hiding technical details from a user. In a network environment, it may mean hiding where in the network a particular file is located [SIL99, 209]. In a client/server configuration, transparency may mean designing the client interface so the presence of the server is not known or visible. The purpose of designing a software system so that it is transparent is generally to avoid confusing or overwhelming a user of the system. The counterpart requirement to transparency is visibility (see also the visibility NFR).

As this requirement relates to designing a system so that some of the details of the software are hidden, the transparency NFR may be found in the Design category of the UML hierarchy.
Example: To avoid confusing or overwhelming users of the ULMTS system, all technical and implementational details will be hidden from users.

understandability – Understandability describes "the ease with which a program component can be understood without reference to other information or other modules" [PRE01, 169]. A requirement is understandable if its readers, which include "customers, users, project managers, software developers, and testers" can easily comprehend the requirement's meaning with a minimum of explanation [DAV93, 208]. If users and customers do not fully understand a requirement in the SRS, they may erroneously approve it, leaving the developer to introduce a flaw into the system. If developers and testers do not fully understand a requirement in the SRS, they will be unable to build or test the system. "The burden of creating an understandable SRS falls on the shoulders of the writers; it is not the readers' responsibility to learn everything writers know in order to digest the SRS" [DAV93, 208].

Understandability is a major requirement affecting the testing phase. Testing will likely be more effective if:

*"The design is well understood.
*Dependencies between internal, external, and shared components are well understood.
*Changes to the design are well communicated.
*Technical documentation is instantly accessible.
*Technical documentation is well organized.
*Technical documentation is specific and detailed.
*Technical documentation is accurate." [PRE01, 338]

Because the understandability requirement describes the clarity of the specification to both the developers and the customers, the requirement can be found in the SRS category of the UML hierarchy. A system's understandability is also affected by the conciseness NFR (see also conciseness).
Example: Every ULMTS program component shall be able to be reasonably understood by developers without reference to other information or other modules.

uniform performance – The uniform performance requirement describes the degree of uniformity of a system's performance between different performance factors. For example, a system has a high level of uniform performance if its peak-period performance level is equivalent to its off-peak-period performance (see also the peak-period and off-peak-period performance requirements). Likewise, a system has a low level of uniform performance if its main-memory performance far outweighs its secondary storage performance (see also the main-memory and secondary storage performance requirements). The uniform performance requirement is similar to the
mean performance requirement (see also), except that instead of comprising the average of different performance factors, it describes their similarity.

Because this requirement concerns the performance of both time and space, hardware and software, the uniform performance NFR may be found in all four sub-categories that are permutations thereof (HardwareTimePerformance, HardwareSpace-Performance, SoftwareTimePerformance, and SoftwareSpacePerformance).
Example: The ULMTS's peak-period performance level shall not vary by more than twenty percent of its off-peak-period performance.

uniformity – The uniformity requirement describes two aspects of a system's design and development. Uniformity may refer to "the use of uniform design and documentation techniques throughout the software development project" [MCC77, 43] and it may also describe the system's specifications as listed in the SRS. As it relates to the latter definition, a specification is uniform "to the extent that its provisions do not conflict with each other, other specifications, or objectives [DAV93, 144].

The uniformity requirement, as it describes specifications, may be subdivided into internal uniformity, which states that no subset of individual requirements within an SRS conflict with each other, and external uniformity, which states that no requirements contained in an SRS conflict with any already baselined project documentation [DAV93, 144].

Because this requirement concerns both a system's design and its specifications, the uniformity requirement may be found in two places in the UML hierarchy: in the Design category and in the SRS category.
Example: The individual modules (classes) of ULMTS shall follow the standard variable, function, and documentation format as described in the ISO-9000 C++ standard, section 1.31.19.

usability – The usability requirement describes "the effort required to learn, operate, prepare input, and interpret output of a program" [MCC77, 222]. "Usability" is almost synonymous with "user-friendliness" (see user-friendliness). Pressman distinguishes between the two by defining usability as an attempt to quantify user-friendliness, and lists four characteristics that can be used to measure usability:

*"The physical and/or intellectual skill required to learn the system
* The time required to become moderately efficient in the use of the system
* The net increase in productivity (over the approach that the system replaces)
* A subjective assessment (sometimes obtained through a questionnaire) of users' attitudes toward the system" [PRE01, 306]

Since the usability requirement concerns the end-user interacting with the user interface in the contexts of training *and* operation, this requirement can be found in two places in the UML hierarchy; first, in the UserInterface category, Learnability sub-category, and second, in the UserInterface category, ContinuedUse sub-category.

Usability is a more general variation of the closely-related learnability requirement (see also learnability).
Example: The end users who are subjected to the ULMTS system's operational features throughout its lifetime shall experience an acceptable level of ease of use when interacting with the system.

user-friendliness – The user-friendliness requirement describes a system's user interface. User-friendliness is nearly synonymous with the usability requirement (see also usability). A user-friendly system should hide the details of the underlying software from those users who do not wish to see it; this may be accomplished through the use of "a consistent user interface, default parameter settings, the use of icons and windows, self-explanatory commands, and online help messages" [THA97, 192]. A developer wishing to construct a user-friendly system should keep the end-user in mind through the development phase, constantly asking him or herself, "how can I make this system more human-friendly and less technically-oriented?"

Since the user-friendliness requirement concerns the end-user interacting with the user interface in the contexts of training *and* operation, this requirement can be found in two places in the UML hierarchy; first, in the UserInterface category, Learnability sub-category, and second, in the UserInterface category, ContinuedUse sub-category. The user-friendliness requirement is closely related to the operability requirement (see operability).
Example: To promote user-friendliness, the ULMTS system shall employ a consistent user interface, default parameter settings, the use of icons and windows, self-explanatory commands, and online help messages.

validity – The validity requirement describes the action of validation, "the set of activities that ensure that the software that has been built is traceable to customer requirements" [PRE01, 283]. Validation is almost always mentioned with verification (see also the verification requirement). Together, "verification and validation" (V&V) are two of the main tasks in system testing and software quality assurance (SQA).

Validity is nearly synonymous with the correctness requirement (see also correctness). Since the validity NFR concerns determination of the correctness of the final system compared to the customer's needs and requirements, it may be found in the SRS category of the UML hierarchy. Since validity is one half of the verification and validation process, which is concerned with the system testing development phase, it may also be found in the Testing category of the hierarchy.
Example: The ULMTS deliverables shall be valid in that they do not vary from the system described in the SRS; developers will be held liable for any variance in said deliverables.

variability – The variability requirement describes how easily a software system may be changed once it has been deployed. Making modifications to a system is related to the degrees of cohesion exhibited by its components and the degree of coupling existent between components [YOU79, 118] (see also the cohesiveness and coupling requirements).
The modifiability requirement differs from adaptability in that adjustments to the system are made after the system is put into place, whereas with the adaptability requirement, modifications are made to the system before it is implemented so that it satisfies the constraints of all elements already in place.

As this requirement relates to the changeability of a system that has already been delivered to the customer and put into use, the variability NFR can be found in the Future category, in the FutureChange sub-category of the UML hierarchy.
Example: Changes in user needs shall allow changes to the functionality of the ULMTS to be possible.

verifiability – The verifiability requirement describes the action of verification, "the set of activities that ensure that software correctly implements a specific function" [PRE01, 268]. Verification is almost always mentioned with validation (see also validation). Together, "verification and validation" (V&V) are two of the main tasks in system testing and software quality assurance (SQA).

Verifiability may also refer not only to system testing, but also to the system's SRS. In this capacity, an SRS is verifiable if "there exist finite, cost effective techniques that can be used to verify that every requirement stated therein is satisfied by the system as built" [DAV93, 133]. There are several reasons why a requirement may be difficult to verify:

*Ambiguousness: if multiple interpretations exist for a requirement, there is no way to verify it.

*Undecidable: A requirement may be undecidable if it calls for a condition equivalent to the halting problem, for example, "The system shall never halt."

*Not worth cost: If a requirement calls for a minimum percentage of human deaths in case of failure, it is not worth the cost to test [DAV93, 211].

The verifiability may be found in two places in the UML hierarchy: in the Testing category and the SRS category.
Example: The correctness of the final ULMTS deliverable shall be verified with a suite of acceptance tests described in the SRS, section 5.2.1.3.

versatility – The versatility requirement describes the number of different functions one component or system can do. Which is to say, the more functionality a system has, the higher its level of versatility. Individual components are said to be versatile if they can be reused (a related requirement to the versatility NFR is the reusable NFR). For example, if developers have created a Person class for a payroll system and are able to reuse the same Person class without having to modify it for use in another development project, that class may be said to be versatile. Component versatility is hard to implement since future customer needs may be hard to predict. System versatility may be achieved simply by increasing the software's level of functionality.

Because this requirement describes the reusability of components or the total functionality of a software system, the versatility NFR may be found in the Design category of the UML hierarchy.
Example: The ULMTS will be versatile in that it will allow authorized users to search, add, remove, and change all materials in the main database.

visibility – This requirement describes the ease with which a system's functionality may be observed, particularly as it relates to its testability. Building visibility into a software system is important, because potential errors can be detected earlier by the developers if they manifest themselves more obviously as the system is tested. A high level of visibility in a system may be achieved in a number of ways:

*"Distinct output is generated for each input.
*System states and variables are visible or queriable during execution.
*Past system states and variables are visible or queriable (e.g., transaction logs).
*All factors affecting the output are visible.
*Incorrect output is easily identified.
*Internal errors are automatically detected through self-testing mechanisms.
*Internal errors are automatically reported.
*Source code is accessible." [PRE01, 237]

Because of its close relationship with a system's testability, the visibility requirement can be found in the Testing category of the UML hierarchy.
Example: Visibility in the ULMTS during the testing phase shall be achieved in three ways:
a) Distinct output shall be generated for each input.
b) System states and variables shall be visible and queriable during execution.
c) Past system states and variables shall be visible and queriable by the use of transaction logs.

wrappability – In software engineering, "wrapping" means placing some functional unit (such as a segment of code) into a logically cohesive unit, such as a struct or a class. In a procedural language, a function might be a free-standing entity. In converting these procedural-language functions into an object-oriented paradigm, a designer may take that and other functions, and "wrap" them together, possibly along with member data (variables). The wrappability requirement is closely related to the concepts of encapsulation and data-hiding, in that by wrapping functions into "wrapper classes", one may hide how the functions accomplish their work, leaving only the functions' signatures in interfaces to allow others to use these wrapper classes knowing only what they do, not how they do it.

Because this requirement describes the design and placement of units of code into higher-level containers, the wrappability NFR may be found in the Design category of the UML hierarchy.
Example: The implementation of the ULMTS system shall achieve wrappability with the use of an object-oriented language (C++); all member data and member functions will thus be wrappable within objects instantiated from a library of classes.

VITA

JAMES D. MOODY, II

Personal Data:	Date of Birth: April 22, 1968
	Place of Birth: Knoxville, Tennessee
	Marital Status: Happily Married
Education:	Public Schools, Johnson City, Tennessee
	University of Tennessee, Knoxville, Tennessee; English, B.A., 1993
	East Tennessee State University, Johnson City, Tennessee; English, M.A., 1998
	East Tennessee State University, Johnson City, Tennessee; Computer Science, M.S., 2003
Professional Experience:	Instructor, East Tennessee State University, College of Applied Sciences, 2002 - 2003
	Graduate Assistant, East Tennessee State University, College of Applied Sciences, 2003 - 2003