Using UML to Reflect Non-Functional Requirements

Luiz Marcio Cysneiros
Department of Computer Science
University of Toronto
cysneiro@cs.toronto.edu

Julio César Sampaio do Prado Leite
Departamento de Informática PUC- Rio
julio@inf.puc-rio.br

Abstract

The way requirements should drive the rest of the software development process has been a subject of many research projects in the past. Unfortunately, all of them focus primarily, when not exclusively, on the functional requirements regardless of the fact that non-functional requirements (NFR) are among the most expensive and difficult to deal with [6] [13] [4][10]. This work evolves out of a previous one [11] and aims at filling this gap, proposing a systematic approach to assure that conceptual models will reflect the NFRs elicited. We focus our attention on conceptual models expressed using the UML [25], and therefore, some heuristics are proposed to make UML suitable to handle NFRs.

1. Introduction

The market is increasing its demands for providing software that not only implements all the desired functionality but also some non-functional aspects as: cost, reliability, security, maintainability, portability, accuracy among others. These non-functional aspects must be treated as non-functional requirements (NFR) of the software. They, still, should be dealt with from the beginning of software development process [8] [9] [10], throughout the whole life cycle.

Not eliciting NFRs or dealing with then later in the process has led to a series of histories reporting failures in software development, including the deactivation of a system right after its deployment [4][18]. Studies point out these requirements as being among the most expensive and difficult to correct [6] [13] [10].

Only a few works propose some way of explicitly dealing with NFRs under the process-oriented approach.

From the perspective of the existing literature, most of the work that tackles the use of NFRs during the software development process is still partial [2][3][17]. Chung's NFR Framework [9] is among the most complete works on NFR and proposes a framework that uses non-functional requirements to drive design and to support architectural design.

However, there is still a gap regarding the integration of NFR into conceptual models.

Some of our previous works [10] [11] emphasize the premise presented by Chung [9] that the lack of integration of NFR to functional requirements, can result in projects that will take more time to be concluded, as well as bigger maintenance costs.

Our work aims at filling this gap, i.e., proposing a strategy that leads conceptual models to reflect the NFRs elicited. It deals with some of the dynamic aspects of the NFR that were not covered in previous works and enhances the integration process presented before [11].

We conducted three case studies to validate the proposed strategy. Two of these case studies were controlled experiments and they were based on the use of the specification for the implementation of the Light Control System for the University of Kaiserslautern [12]. The third case study could be classified as a real life case study and was conducted during the development of a software for a controlling a clinical analysis laboratory. Section 5 will present further details.

The results of the case studies suggest that the use of the strategy can lead to a more complete conceptual model as well as to a faster time-to-the market process, since errors due to not properly dealing with NFR, which are usually difficult and time consuming to correct, can be avoided.

Section 2 will present an overview of the entire strategy to deal with NFR, while Section 3 will detail the integration process and Section 4 will present the heuristics on how UML classes may handle NFR. Section 5 will present the results from three case studies used to validate the strategy and Section 6 will conclude and present some future works.

2. An Overview of the Strategy to Deal with NFR

We face the software development process as being composed of two *independent* evolutionary cycles that may be dealt with separately.

In our strategy the software development process is carried out through two independent cycles, one regarding the functional aspects of the system and the other the non-functional aspects of the software.

Since we face them as independent cycles we propose to establish convergence points between both cycles.

Through the use of these convergence points we can express in the functional view all the actions and data that will be necessary to satisfice the NFRs tackled in the non-functional view.

Figure 1 shows the idea of these two views as well as its convergence points.

The so-called functional view includes some of the artifacts commonly used in the software development process, while the non-functional view uses the NFR Framework [9] as its basic representation.

We will not focus on any particular approach to build the conceptual models of the functional view. We understand that our strategy may fit to any approach one decides to use.

Although our strategy deals with all the artifacts showed in Figure 1, this article will tackle only the use of UML class diagrams to handle NFR.

Regarding NFR we understand that, although our strategy may be used to almost any type of NFR, we understand that its results will be more effective when addressing NFRs that effectively demand any kind of action to be performed by the system, and therefore affects the software design.

NFRs such as “small learning curve” or “detailed documentation” are not likely to impact on software design and hence will possibly benefit from our strategy only in the elicitation part.

On the other hand, NFRs such as “Safety”, “Performance”, “Accuracy” and others, frequently demands the design to be carefully studied in order to satisfice these NFRs. Hence, these NFRs will be more likely the type of NFR that our strategy will be more helpful.

As portrayed in Figure 1, we propose to build both the functional and the non-functional views anchored in the Language Extended Lexicon (LEL) [19]. This will lead to a smooth integration between both views. Therefore, building the Lexicon will be the first thing to be carried out in this strategy.

The objective of the LEL is to register the vocabulary of a given UofD[1]. It is based upon the following simple idea: understand the problem’s language without worrying about deeply

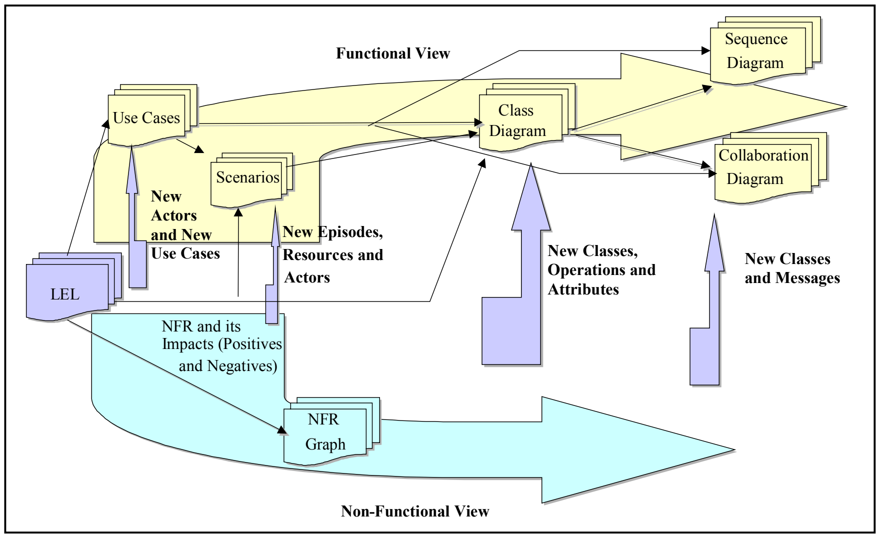

Functional View
Sequence Diagram
Use Cases
Class Diagram
Scenarios
Collaboration Diagram
New Actors and New Use Cases
New Episodes, Resources and Actors
New Classes, Operations and Attributes
New Classes and Messages
LEL
NFR and its Impacts (Positives and Negatives)
NFR Graph
Non-Functional View

Figure 1 – An Overview of the Strategy to Deal With NFR

[1] *“Universe of Discourse is the general context where the software should be developed and operated. The UofD includes all the sources of information and all known people related to the software. These people are also known as the actors in this UofD.”*

understanding the problem [19]. The main objective of the LEL is to register signs (words or phrases) peculiar to a specific field of application.

The LEL is based on a code system composed of symbols where each symbol is an entry expressed in terms of notions and behavioral responses. The notions must try to elicit the meaning of the symbol and its fundamental relations with other entries. The behavioral response must specify the connotation of the symbol in the UofD. Each symbol may also be represented by one or more aliases.

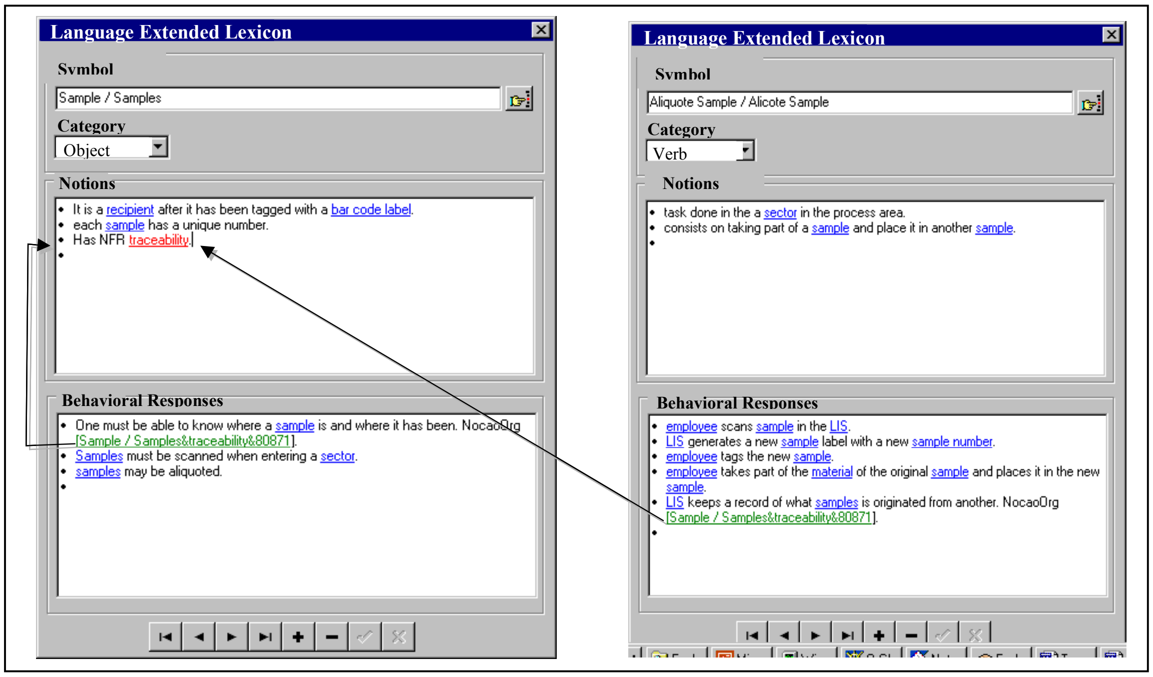
The construction of the LEL must be oriented by the minimum vocabulary and the circularity principles. The circularity principle prescribes the maximization of the usage of LEL symbols when describing LEL entries, while the minimal vocabulary principle prescribes the minimization of the usage of symbols exterior to the LEL when describing LEL entries. Because of the circularity principle, the LEL has a hypertext form. Figure 2 shows an example of two entries in the LEL.

Although the LEL can handle non-functional aspects of the domain, at least the very first version of the LEL is usually mainly composed of symbols related to functional requirements.

This is due, in the first place, to the very abstract nature of non-functional requirements. In addition, quality aspects, in spite of its importance, are usually hidden in everyone's mind.

However, it does not mean that the software engineer cannot register information about non-functional requirements. A well-defined set of symbols representing the vocabulary of the UofD is a key point to the strategy.

Building the non-functional view departs from the use of an existing LEL. In order to aid on this process we have extended the LEL to help NFR elicitation. The LEL is now structured to express that one or more NFRs are needed by a symbol. It is also structured to handle dependency links among one NFR and all the notions and behavioral responses that are necessary to satisfice this NFR. Figure 2 shows these new



Sample / Samples

Object

- It is a recipient after it has been tagged with a bar code label.
- each sample has a unique number.
- Has NFR traceability.

- One must be able to know where a sample is and where it has been. NocaoOrg [Sample / Samples&traceability&80871].
- Samples must be scanned when entering a sector.
- samples may be aliquoted.

Aliquote Sample / Alicote Sample

Verb

- task done in the a sector in the process area.
- consists on taking part of a sample and place it in another sample.

- employee scans sample in the LIS.
- LIS generates a new sample label with a new sample number.
- employee tags the new sample.
- employee takes part of the material of the original sample and places it in the new sample.
- LIS keeps a record of what samples is originated from another. NocaoOrg [Sample / Samples&traceability&80871].

Figure 2 – Example of LEL Entries and its Extension to Deal with NFR

It is important to make it clear that the LEL is not restricted to holding information related to functional requirements. Its idea is to register the entire vocabulary in the UofD and therefore it might also include the non-functional aspects of the domain.

features of the LEL.

We can see in Figure 2 that the symbol Sample belonging to a Clinical Analysis Laboratory information system (Case Study 3 detailed later in Section 5), has in its notions the reference to the need of the NFR *Traceability*.

It also shows that in order to satisfice this NFR we had to add one behavioral response to the symbol Sample itself (the one that states that one should be able to know where a sample is and where it has been). We also had to add another behavioral response to the symbol Aliquote Sample (the one that states that the system – LIS – should keep a record of what samples were originated from another sample).

We can also see beside these behavioral responses a pattern that was included to establish a dependence link between these behavioral responses and the notion that originated its need

This will help us to further represent the NFRs and its operationalizations. It also helps when for any reason we need to update the LEL.

For example, suppose that reviewing the LEL we decide now that Sample does not need any traceability any more. We should be able to easily see what notions and behavioral responses may be excluded from this symbol or from other symbols. The OORNF tool [21] does that automatically.

We have extended the OORNF tool to support these extensions. This tool was originally developed to support the requirements baseline proposed by Leite [20] with some NFR support.

The tool has a knowledge base on NFR that can and must be constantly updated. This knowledge base stores a variety of NFRs and some common way to decompose them. The tool brings along with the definition of these NFRs a set of possible conflicting NFR as well as a set of NFRs that may be positively or negatively impacted by this NFR.

The OORNF tool also provides support for entering the LEL, scenarios and CRC cards together with a support to create the scenarios from the LEL entries [15] and the CRC cards from the LEL entries and the elicited scenarios [22].

The first step on building the non-functional view will be to enhance the existing LEL with the NFRs that are desired by the stakeholders. To do that, we will run through all the LEL symbols using the knowledge base on NFR present in the OORNF tools to ask ourselves and the stakeholder (whenever possible) if any of the NFRs present in this knowledge base may be necessary to each of the LEL symbols. Each NFR found may be represented in the symbol as showed in Figure 2.

After representing the need for an NFR, the software engineer has to ask himself and the stakeholders what should be necessary to do to satisfice this NFR. Again, the knowledge base in the OORNF tool may be used. All the information that arises from this questioning may be represented in the LEL either in the same symbol or in another symbol if necessary.

Coming back to Figure 2, when we get to the symbol Sample (after examining all the previous symbol in the LEL), we asked the stakeholders and ourselves if any of the NFRs in the knowledge base would apply for samples. The answer was that traceability was essential once the laboratory could not afford to lose one sample since drawing a new one could be sometimes almost impossible or at least very painful.

We then represented this NFR in the notion of the symbol Sample as seen in Figure 2 and started to ask how could we guarantee this traceability to work.

One of the answers was that every time a sample is aliquoted (expression used in this domain that means to create an aliquote, or yet to draw from one recipient to another) this procedure has to be documented in the software so one can know which sample was originated from another sample. We represented this answer as an entry in the behavioral responses of the symbol Aliquote sample and then established a dependency link between this behavioral response and the NFR traceability stated in the notions of the symbol Sample.

Although the LEL can handle some representation of the NFR and its impacts, it is not the best tool to deal with them in a more complete way, so one can reason about their interdependencies and conduct the necessary tradeoffs that arises during this process.

To fully represent and reason with NFRs we propose to use the NFR Framework proposed by Chung [7] with some minor adaptations.

The NFR Framework [23][7][9] faces NFR as goals that might conflict among each other and must be represented as softgoals to be satisficed. Each softgoal will be decomposed into sub-goals represented by a graph structure inspired by the and/or trees used in problem solving.

This process continues until the requirements engineer considers the softgoal satisficed (operationalized) [9], so these satisficing goals can be faced as operationalizations of the NFR.

Another way of understanding operationalizations is that they are, in fact, functional requirements that have arisen from the need to satisfice NFR. This can explain why we frequently face doubts about if a requirement is functional or non-functional.

In the domain of the laboratory quoted before, we could face a requirement like the one "Samples should be traceable so one can know where this sample is ever".

Some may tend to think this is a functional requirement while, in fact, it is an operationalization of the Traceability NFR [see Chung 00 page 160] for a comprehensive list.

In fact, this will be showing how the functional requirement "The software must handle samples" would be constrained by the NFR Traceability.

In accordance with [9], for us an NFR has a *type*, which refers to a particular NFR as for example security or traceability. It also has a subject matter or *topic*, for example sample as showed in the above example. We would then represent it as Traceability[Sample].

The NFR framework was extended to represent the operationalizations in two different ways. We called them dynamic and static operationalizations.

Dynamic operationalizations are those that call for abstract concepts and usually call for some action to be carried out.

On the other hand, static operationalizations usually express the need for some data to be used in the design of the software to store information that is necessary for satisficing the NFR [11]. Figure 3 shows an example of an NFR graph where we can see these two types of operationalizations.

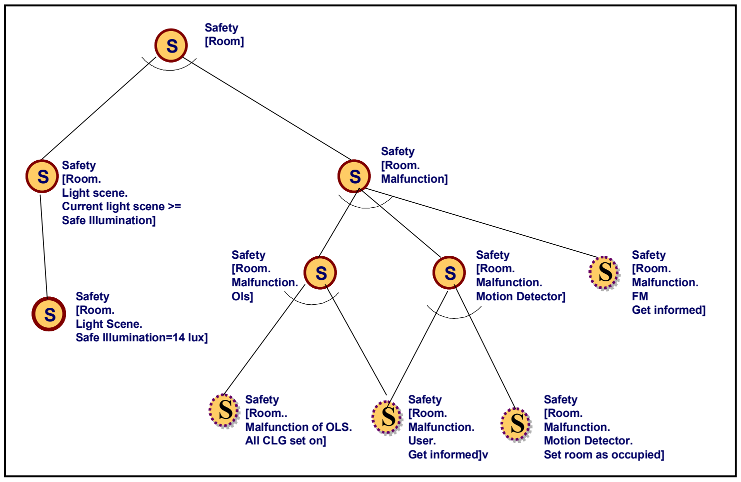


Figure 3 – An Example of an NFR Graph

On the top of the Figure 3 (extracted from the Case Study I – A Light Control System), we can see the node that represents the root of this graph represented as Safety [Room], meaning that room is a place that has to be safe regarding illumination aspects.

One of the operationalizations that represent part of this NFR satisficing can be seen on the left side of the figure represented by a bold circle denoting a static operationalization. Here, we can see the need of some information in the system that represents the minimum illumination in lux that can be used in a room.

Some dotted circles appear on the bottom of the figure representing dynamic operationalizations. One of them, Safety [Room.Malfunction.User get informed], represents that the user may be informed of any malfunction that occurs in the room. The letter S inside each node represents that this sub-goal is Satisficed. The letter P can also be used for those ones that are Partially satisfied or D for those ones that are Denied. Further details can be seen in [11].

It is important to stress that the identifier that appears close to the NFR on the root of the graph (NFR Topic) has necessarily to be a symbol of the LEL. In Figure 3 we see that the root node is represented by Safety [Room], so room has to be a symbol of the LEL.

If one cannot find the word or sentence intended to be used as a topic for an NFR either one symbol represented in the LEL has an alias not defined or the LEL is incomplete and should therefore be updated.

To build the NFR model we will search every entry of the LEL looking for notions that express the need for an NFR.

For each NFR found, we must create an NFR graph expressing all the operationalizations that are necessary to satisfice this NFR. At the end of this process we will therefore have a set of NFR graphs that will represent the non-functional aspects of the system.

Once we have this set of NFR graphs ready, we have to look for possible interdependencies, positive or negative, among NFRs. For example, an NFR pointing out that the software might need a high level of Security may have a negative impact (a negative interdependency) on another NFR like Usability, since when one enhances the security controls he/she frequently hurts usability aspects.

On the other hand, one might face an NFR for Performance that claims for the need of saving space used to store some information. To operationalize this NFR one option would be to use uncompressed format. This operationalization could eventually contribute positively (a positive interdependency) for another NFR that calls for a good response time when dealing with these information. Further details can be seen in [9].

It is important to point out that all the effort on NFR tradeoffs due to positive and negative interdependencies will take place in the non-functional view, i.e., using the NFR framework. What will be integrated into the functional view will be the result that one gets after all the necessary reasoning on NFR interdependencies and its consequences, i.e. the operationalizations.

We propose three heuristics to helps us on finding these interdependencies.

1- Compare all NFR graphs of the same type searching for possible interdependencies. For example, we may put all the NFR graphs that has the type Safety together to see if there is any interdependency among them
2- Compare all the graphs that are classified in the knowledge base [21] as possibly conflicting NFRs. For example compare graphs of Security with graphs of Usability. Frequently to add security to a system we have to sacrifice its usability.
3- Pair wise all the graphs that were not compared while applying the above heuristics.

Figure 4 shows an example of a negative interdependence found when we were applying the third heuristic during one of our case studies. This graph was extracted from a software for clinical analysis laboratory (Case Study III).

This figure shows the pair wising of a graph that deals with operational restrictions that applies for patient's report with another one that deals with reliability of tests. Pair wising these graphs we could see that the sub-goal that deals with the aspects of how to assure reliability when electronically signing patient's report has a negative influence on the operationalizations that states that in order to satisfice operational restrictions regarding time restrictions to print the patient's report, the system (LIS) should electronically sign all the patient's reports.

It is important to clarify that the sub-goals that appear in Figure 4 as partially satisficed (P) were considered satisficed (S) before we carried out the

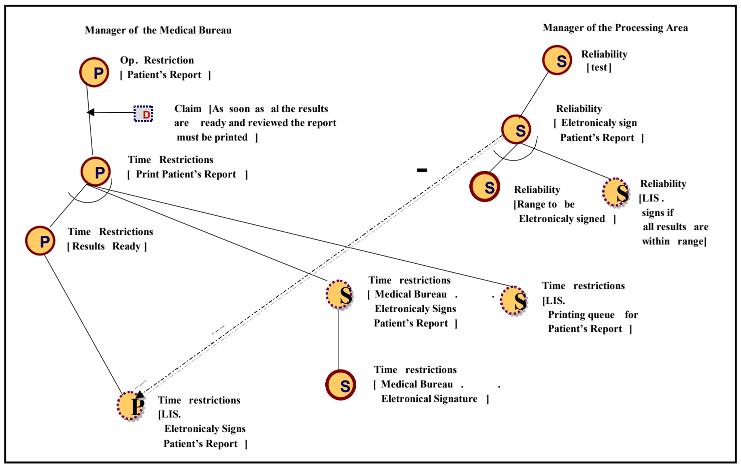

Manager of the Medical Bureau
Manager of the Processing Area
P
Op. Restriction [Patient's Report]
D
Claim [As soon as al the results are ready and reviewed the report must be printed]
P
Time Restrictions [Print Patient's Report]
P
Time Restrictions [Results Ready]
P
Time restrictions [LIS. Eletronicaly Signs Patient's Report]
S
Reliability [test]
S
Reliability [Eletronicaly sign Patient's Report]
-
S
Reliability [Range to be Eletronicaly signed]
S
Reliability [LIS . signs if all results are within range]
S
Time restrictions [Medical Bureau . Eletronicaly Signs Patient's Report]
S
Time restrictions [LIS. Printing queue for Patient's Report]
S
Time restrictions [Medical Bureau . Eletronical Signature]

Figure 4 – Identifying and Solving Interdependencies Among NFR graphs

pair comparison. These sub-goals were considered partially satisficed only after we negotiated with the stakeholder to compromise with an intermediary approach.

This is exactly what is represented in Figure 4, i.e., the patient's reports will be electronically signed by the system *only* when all the results are within a predefined range considered safe to this task.

Although being important, heuristic three may become inappropriate to be used when the number of NFR graphs is very large. We do not have a rule-of-thumb to this number but we have used it in a case study where we dealt with more then 70 NFR graphs and it was still worth to use. During this case study the total overhead of using the proposal was 7% [12] and therefore we think it was quite worth to use it.

To integrate the functional and non-functional views we propose a process to drive the operationalizations that are represented in the set of NFR graphs to the functional view.

The integration process can take place either in the early phases, integrating the NFRs into the use case or scenario models, or later, integrating

NFRs into class, sequence and collaboration diagrams.

Notice that we do not propose to deal with NFR elicitation in any of these models. NFR's satisficing usually requires a very detailed reasoning to reach its operationalizations. Also, one NFR frequently presents many interdependencies with other NFRs. Dealing with these particular aspects of NFRs can be quite difficult to be accomplished directly in use cases, scenarios or even class diagrams.

Thus, we propose to deal with NFR in the NFR view using the approach we previously described to build the non-functional view, and then to integrate the operationalizations found into the models used in the functional view.

We also propose a traceability mechanism to make it easier to navigate between models.

3. The Integration Process

Since functional and non-functional views are cycles that are independent from each other, we need to establish convergence points where the integration of both views can be done, i.e., a way to represent in the functional view the operationalizations found in the non-functional views.

The integration method is based on the use of the LEL as an anchor to build both the NFR graph [7] and the class diagram. It means that every root of each NFR graph has to refer to a symbol of the LEL and every class of the class diagram has to be named using a symbol of the LEL.

The use of the LEL as an anchor to construct both views will facilitate their integration. It can also be used to validate both models since, if for some reason one cannot find a symbol of the LEL for naming a class, it means that either the symbol is missing in the LEL definition, and therefore should be added to it, or any symbol of the LEL has an alias that was not yet considered.

Using this anchor, the integration process is centered on the search of a symbol that appears in both models and later in evaluating the impacts of adding the NFR operationalizations to the class diagram. Figure 5 depicts the integration method for the class diagram.

We start the process by picking out a class from the class diagram. We will then search all the NFR graphs from the non-functional view looking for any occurrence of this symbol.

For each graph in which we find the name of the class we are searching now, we have to identify the dynamic and static operationalizations in this graph.

We have to check if the operations, in the case of dynamic operationalizations, and attributes, in the case of static operationalizations, that belong to the class already fulfill the needs expressed in the graph's operationalizations. If they do not, we have to add operations and attributes so that this class ends up fulfilling these needs.

Notice that addition of new operations may sometimes call for the inclusion of new attributes in order to implement the desired operation.

The same applies for new attributes that are added to classes to satisfy static operationalizations. We may often have to add new operations to handle these new attributes.

Let us take for example a class named *Room* from a light control system (extracted from Case Study I detailed in Section 5). We had to search all the NFR graphs in the non-functional view looking for the symbol Room. We could then find the NFR graph showed in the Figure 3 where we can see five operationalizations, four dynamic and one static.

The four dynamic operationalizations state

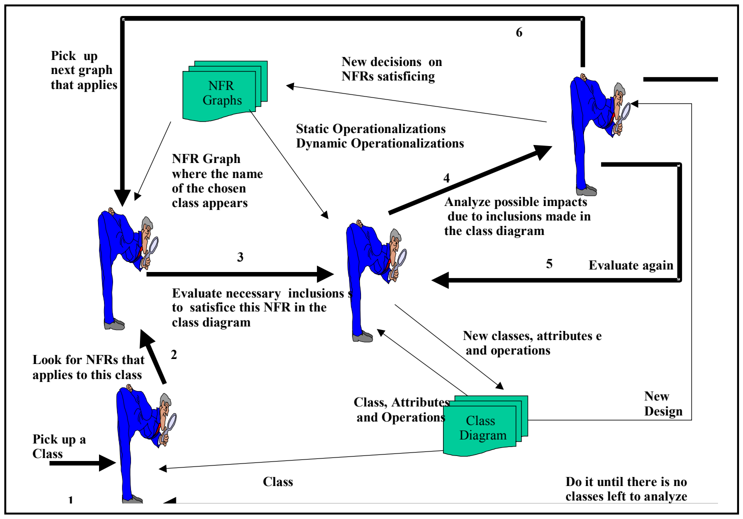

6
Pick up next graph that applies
NFR Graphs
New decisions on NFRs satisficing
Static Operationalizations
Dynamic Operationalizations
NFR Graph where the name of the chosen class appears
4
Analyze possible impacts due to inclusions made in the class diagram
3
5
Evaluate again
Evaluate necessary inclusions s to satisfice this NFR in the class diagram
New classes, attributes e and operations
Look for NFRs that applies to this class
2
Class, Attributes and Operations
Class Diagram
New Design
Pick up a Class
Class
Do it until there is no classes left to analyze
1

Figure 5 – The Class Diagram Integration Process

that the software must:

1) Turn all the lights on when a malfunction of the outdoor light sensor occurs;
2) Advise the user of a malfunction of both outdoor light sensor or motion sensor;
3) Set the room as occupied in the case of a motion sensor malfunction and
4) Advise the facility manager (FM) of any malfunction.

Now we have to check if any operation in the class *Room* already performs these actions. If they do not, we should add operations to handle these actions.

Notice that if the class that we are analyzing is part of a **generalization** we have to check if any superclass or subclass of this class does not have operations that satisfy the needed operationalizations.

On the other hand, the static operationalization states that there should be an attribute fixing the minimum amount of light in a room as 14 lux. In this case, we have to check if there is an attribute in the class *Room* to store this information. Notice that what was said before regarding this class being part of generalizations also applies here.

It is important to state here that, although we strongly recommend the use of the LEL, this process can be carried out without using the LEL as an anchor. To do that, for every class in the class diagram one should see if there is any graph in the non-functional view semantically related to this class. Of course, if the integration process is carried out this way, we will not have a process as smooth and precise as when we use the LEL.

Since the requirements engineer has to be aware of the vocabulary used in the UofD to elicit requirements, we think that the use of the LEL has little impact on the overall overhead.

4. Heuristics on how to use UML artifacts to handle NFR

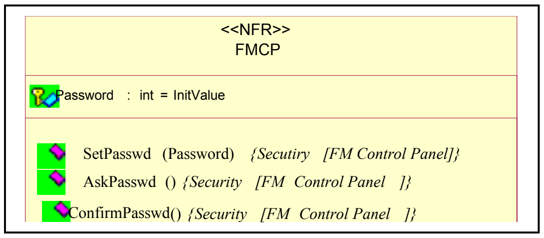
We propose four heuristics for introducing NFR into UML diagrams.

1 - Classes created to satisfice an NFR may be stereotyped as <<NFR>>. Since NFRs are often more difficult to be on designers' mind than functional requirements, stereotyping these classes avoid classes to be withdrawn from the class diagram during a reviewing process, because one could not find any reason why this class must exist.

Knowing which NFR was responsible for the inclusion of this class, one has only to examine its operations and/or attributes to find the link to the non-functional view.

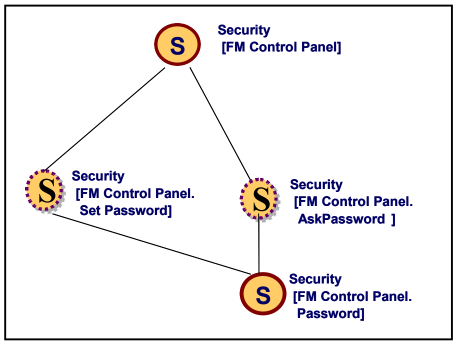
Figure 6 shows an example of such a class. This class was created when we were analyzing the class *Control Panel* in the class diagram for a light control system (Case Study II described in Section 5).

It is important to make it clear that the creation of a new class to satisfice an NFR will always be a design decision. The software engineer could had chosen, in this case, to add the same attributes and operations present in the class showed in Figure 6 to another already existing class such as the *Control Panel* class.



<<NFR>>
FMCP
Password : int = InitValue
SetPasswd (Password) {Secutiry [FM Control Panel]}
AskPasswd () {Security [FM Control Panel]}
ConfirmPasswd() {Security [FM Control Panel]}

Figure 6– Example of a Class Created to Satisfice an NFR



Security [FM Control Panel]
Security [FM Control Panel. Set Password]
Security [FM Control Panel. AskPassword]
Security [FM Control Panel. Password]

Figure 7 – NFR Graph for a Light Control System

As we were analyzing what possible NFRs could affect this class, we searched the set of NFR graphs in the non-functional view looking for the symbol Control Panel.

One of the graphs we found is showed in Figure 7. We can see in this figure that this graph calls for two dynamic operationalizations: 1) Set Password and 2) Ask Password. These two operationalizations were necessary because only the facility manager can change some parameters of the system for security reasons.

Therefore, there has to be not only a control panel for the facility manager, but also, there has to be a process to check the password in this control panel to avoid non-authorized people to use it.

As we did not find in the class *Control Panel* anything that would satisfice this NFR, we decided to create a new class (*FMCP* – Facility Manager Control Panel), as a subclass of the class Control Panel.

2 - Beside each operation that has been included to satisfice an NFR we may represent a link to the non-functional view. As in heuristic one, this is to enforce traceability between models, so the designer can easily check non-functional aspects whenever he changes anything in this class. To represent the NFR that has originated one operation we use the following syntax: ***{NFR [LEL symbol] }.***

Let us take for example the class *Room* mentioned before. Suppose we create an operation named AdviseUserofMalfuntion in order to perform one of the operationalizations, we should then represent it as follows: AdviseUserofMalfunction() {Safety [Room]}.

We present here a different syntax from the one presented in [11] for representing NFR. We have changed it so one can trace exactly which NFR has originated the need for a specific operation. This traceability was not possible in the former work when there were more then one NFR per class. This also holds for heuristics 3 and 4.

3 – We may represent beside an operation any possible pre or post condition that applies to this operation. We will represent these conditions between branches.

This heuristic is used to deal with operational restrictions that some NFRs impose. These operational restrictions should be inputted as pre or post conditions to an operation and whenever possible should be stated using OCL [24]. These pre and post conditions can also be stated in a note linked to the class.

4 - Beside each attribute that has been added to satisfice an NFR we may use the same expression we use in the operations to establish a link to the non-functional view.

Figure 9 shows an example with the results of applying the heuristics two to four. This figure shows the class *LightGroup* after the integration process was carried out during the Case Study II, detailed in Section 5. Figure 8 shows the class *LightGroup* before we carried out the integration process.

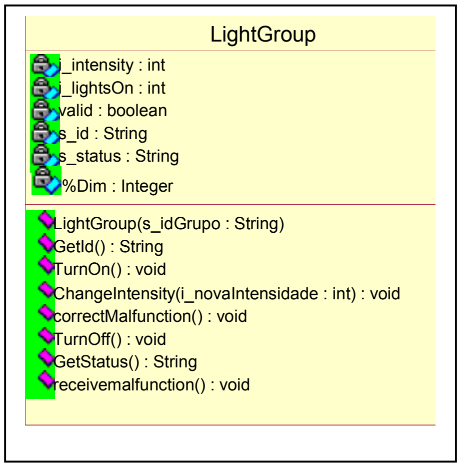


Figure 8 – Class LightGroup Before NFR Integration

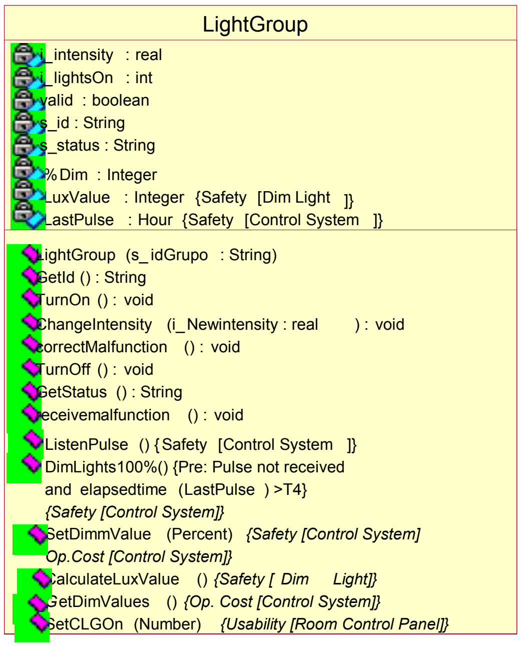

LightGroup
i_intensity : real
i_lightsOn : int
valid : boolean
s_id : String
s_status : String
%Dim : Integer
LuxValue : Integer {Safety [Dim Light]}
LastPulse : Hour {Safety [Control System]}
LightGroup (s_idGrupo : String)
GetId () : String
TurnOn () : void
ChangeIntensity (i_Newintensity : real) : void
correctMalfunction () : void
TurnOff () : void
GetStatus () : String
receivemalfunction () : void
ListenPulse () {Safety [Control System]}
DimLights100%() {Pre: Pulse not received
and elapsedtime (LastPulse) >T4}
{Safety [Control System]}
SetDimmValue (Percent) {Safety [Control System]
Op.Cost [Control System]}
CalculateLuxValue () {Safety [Dim Light]}
GetDimValues () {Op. Cost [Control System]}
SetCLGOn (Number) {Usability [Room Control Panel]}

Figure 9 – Example of Some features added to the UML notation

During the integration process, we analyzed each class we had in the class diagram. When we picked out the class *LightGroup* (showed in Figure 8 above), we searched the non-functional view looking for any occurrences of this symbol. Figure 10 illustrates one of the graphs found.

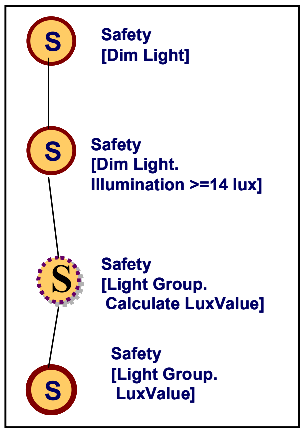


Figure 10 – NFR Graph for Safety applied to Dim Lights

This graph relates to the NFR Safety needed when lights are dimmed. Notice that one of the nodes (sub-goals) that decomposes this graph, shows that in order to be able to criticize if the illumination is greater than 14 lux, the light group has to be able to calculate the equivalent in lux of the percentage set to dim the lights (originally considered). Since there were no attributes or operations in the class doing that task, we added the attribute LuxValue and the operation CalculateLuxValue().

Another graph found was the one regarding the NFR Safety applied to the Control System that can be seen in Figure 11.

We can see in this figure that to satisfice this NFR there has to be an operation to dim the lights to 100%, to be executed if the light group does not receive a signal from the control system after T4 sec.

Again, there were no operations in the LightGroup class to perform this task, thus we added the operations ListenPulse() and DimLights100%() together with the attribute LastPulse necessary to implement the ListenPulse() operation.

Notice that beside the operation DimLights100%(), we see a pre condition that states that this operation will be executed only if this class does not receive the signal within T4 sec.

Another example can be seen in Figures 12 through 16. This example was drawn from the third case study

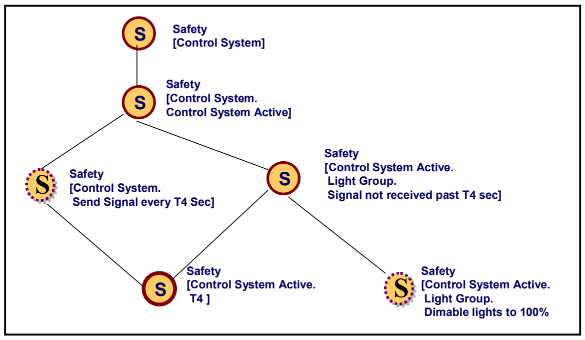


Figure 11– NFR Graph for Safety Applied to the Control System

regarding a software for a clinical analysis laboratory.

Figure 12 shows the class *Result to Inspect* before the NFR integration. This class represents all the results from tests that come from one or more analyzers (special equipments that perform several tests automatically) and are waiting to be inspected by specialized employees that will analyze if they are reliable or not.

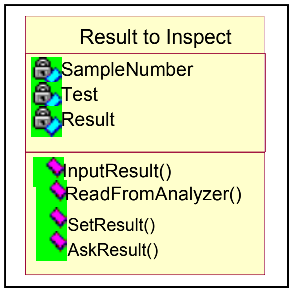
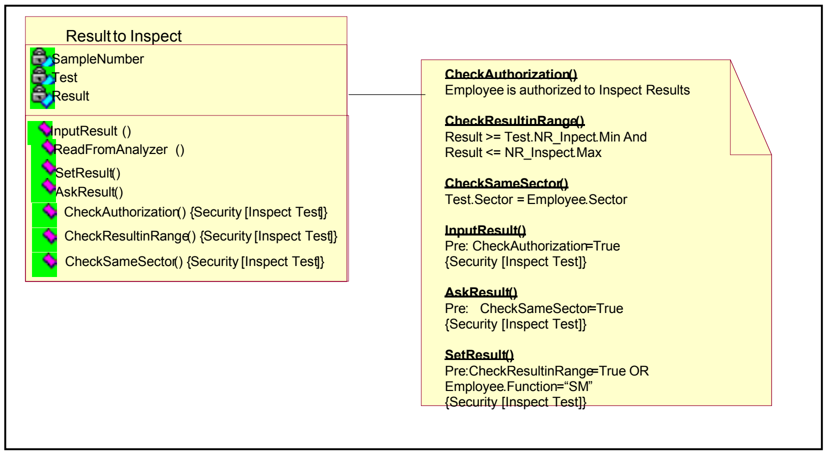


Figure 12 – Class Result to Inspect Before NFR Integration

Figure 13 – A First NFR Graph Found to be Integrated


Result to Inspect

SampleNumber
Test
Result

InputResult ()
ReadFromAnalyzer ()
SetResult()
AskResult()
CheckAuthorization() {Security [Inspect Test]}
CheckResultinRange() {Security [Inspect Test]}
CheckSameSector() {Security [Inspect Test]}

CheckAuthorization()
Employee is authorized to Inspect Results

CheckResultinRange()
Result >= Test.NR_Inpect.Min And
Result <= NR_Inspect.Max

CheckSameSector()
Test.Sector = Employee.Sector

InputResult()
Pre: CheckAuthorization=True
{Security [Inspect Test]}

AskResult()
Pre: CheckSameSector=True
{Security [Inspect Test]}

SetResult()
Pre:CheckResultinRange=True OR
Employee.Function="SM"
{Security [Inspect Test]}

Figure 14 – Class Result to Inspect after integrating the First NFR Graph

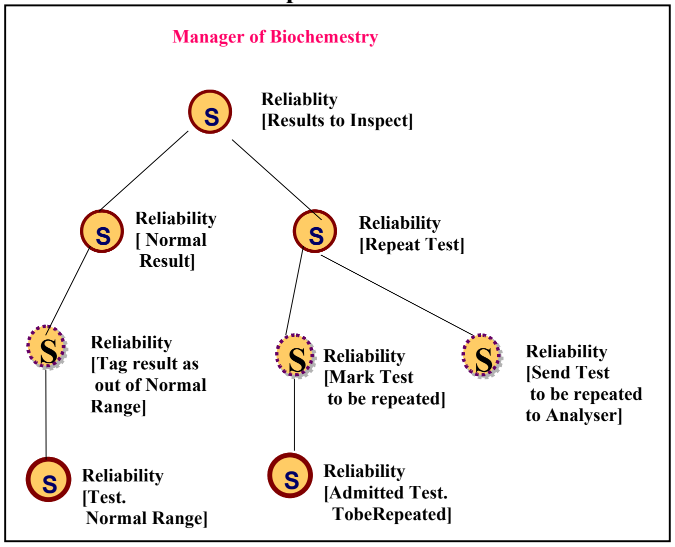

Manager of Biochemestry
S
Reliablity [Results to Inspect]
Reliability [Normal Result]
Reliability [Repeat Test]
Reliability [Tag result as out of Normal Range]
Reliability [Mark Test to be repeated]
Reliability [Send Test to be repeated to Analyser]
Reliability [Test. Normal Range]
Reliability [Admitted Test. TobeRepeated]

Figure 15 – Another NFR Graph

Using the integration process, we searched the set of NFR graphs looking for any occurrences of this symbol of the LEL. Figure 13 shows one of the NFR graph found.

We can see in Figure 13 that there are some specific concerns about security regarding the results to inspect.

In order to satisfice this NFR it is necessary, besides a regular check to see if the employee is authorized to access the software module that implement the result to inspect, a more detailed control.

We can see that the software may check if the employee trying to input or change results belongs to the same sector where the test is processed.

We can also see that the software must check if the inputted value is within a range considered safe to be inputted by any regular employee. Values out of this range can only be inputted by the sector manager.

Figure 14 shows the class after the integration process. We can see in this figure that three new operations were added to satisfice the required operationalizations found in the NFR graph showed in Figure 12.

We can also see a note with many pre and post conditions that may prevail to many of the operations of the class. Notice that here three existing classes were affected by pre and post condition. In this case, we use the same pattern for traceability to establish that these conditions came from NFR. We do not need to do that for the operations that were driven by NFR, like CheckresultinRange, since the operation already carries the traceability link.

Lets take for example the operation SetResult(). This operation is responsible for associating a result to a test in a definitive way. We can see in Figure 14 that this operation can only be executed if the result was previously checked to be in range or if the employee inputting the result is a sector manager.

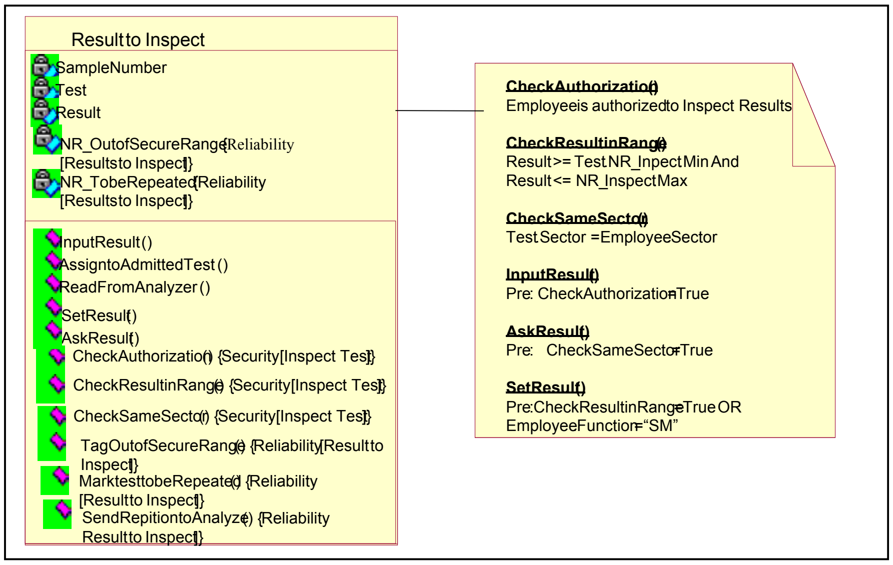
Continuing the process, we found another NFR graph with the symbol Result to Inspect. This graph is portrayed in Figure 16.

In this graph we can see that in order to satisfice the NFR Reliability applied to Result to Inspect, the software must tag results that are out of a range considered to be normal, i.e., results that are out of the range usually experienced by the average population..

The software must also allow the employee to tag some tests to be repeated, as well as to provide a way to send all these tests to the analyzer.

Examining the class *Result to inspect*, we could see that none of these conditions was satisficed yet, and therefore we added three more operations and two attributes. Figure 15 show the final design of the class *Result to Inspect*.

Once all the classes have been used to search the graphs from the non-functional view, the class diagram will then reflect all the attributes and operations that are necessary to implement the needs arisen from NFR satisficing.



Resultto Inspect
SampleNumber
Test
Result
NR_OutofSecureRange{Reliability [Resultsto Inspect]}
NR_TobeRepeated{Reliability [Resultsto Inspect]}
InputResult()
AssigntoAdmittedTest ()
ReadFromAnalyzer ()
SetResult()
AskResult()
CheckAuthorization() {Security[Inspect Test]}
CheckResultinRange() {Security[Inspect Test]}
CheckSameSector() {Security[Inspect Test]}
TagOutofSecureRange() {Reliability[Resultto Inspect]}
MarktesttobeRepeated() {Reliability [Resultto Inspect]}
SendRepitiontoAnalyze() {Reliability Resultto Inspect]}
CheckAuthorization()
Employeeis authorizedto Inspect Results
CheckResultinRange()
Result>= Test.NR_InpectMin And Result<= NR_InspectMax
CheckSameSector()
Test.Sector =EmployeeSector
InputResult()
Pre: CheckAuthorization=True
AskResult()
Pre: CheckSameSector=True
SetResult()
Pre:CheckResultinRange=True OR EmployeeFunction=“SM”

Figure 16 – Final Design for the Class Result to Inspect

5. Strategy Validation

To validate our strategy we performed three case studies. All the case studies were inspired in the use of the project replication proposed by Basili [1].

Since our strategy can be seen as an addition to any software development process, we used three different projects, each one using its own team for developing a conceptual model for the software.

These conceptual models aimed to represent the clients' requirements under the point of view of each one of these three teams.

On the other hand, another team represented by one of the authors of this paper, was in charge of evaluating how correct these conceptual models were regarding NFR.

First building the non-functional view and then integrating it with the conceptual models did this evaluation.

Thus, we found several new classes, operations and attributes that should had been specified in the conceptual model and were not. These classes, operations and attributes were faced as errors in the conceptual models, since, if the software driven from these conceptual models had been delivered to the client without these new classes, operations and attributes, the lack of them would had been pointed out by the client as errors in the software that would had to be fixed.

As a qualitative evaluation on the changes performed in the software, due to NFR satisficing, would be highly subjective, we decided to perform a quantitative analysis regarding how many new classes, operations and attributes were added to the conceptual models to satisfice NFRs.

It can be stated that including new classes, operations and attributes may lead to new errors, as well as to a not so high quality model regarding aspects as coupling and cohesion.

Although it is true, we think that, first, not all of the changes may introduce new errors.

Second, coupling and cohesion can be easily modified after you introduce the new classes, operations and attributes.

Third, not having these new classes, operations and attributes would be for sure errors in the software.

Finally, if all the changes had been considered since the first stages of conceptual models design, the new errors that could be introduced adding these classes, operations and attributes due to the use of the strategy, could have happened anyway.

The Case Study I was conducted using the conceptual models created by Breitman [5] as part of her Ph.D. thesis. She carried out a case study using the implementation of the Light Control System for the University of Kaiserslautern. This system was first proposed in a workshop on Requirements Engineering at schloss Daghstuhl in 1998 [12]. We used the system specification distributed during the Daghstuhl Seminar [12] to build the non-functional view, including the LEL definition enclosed in the specification.

Once we finished it, we integrated the NFRs found in this view with the conceptual models that were built by Breitman. The new classes, operations and attributes that have arisen from this integration were counted as errors on the original conceptual model.

The Case Study II was conducted using the conceptual models created by a group of undergraduate students of PUC-Rio's computer science course to be used as an evaluation for the discipline of Software Project. They also used the specification distributed during the Daghstuhl Seminar 12 to build their conceptual models.

As the source of requirements for both Case Study I and II was the same, we in the Case Study II the same set of NFR graphs from the non-functional perspective of the Case Study I, and integrated them to the conceptual models that were built by the students. Again, new classes, operations and attributes that have arisen from this integration were counted as errors on the original conceptual model.

The Case Study III was conducted together with a software house specialized in building software for clinical analysis laboratories. They were responsible for developing a new information system to a laboratory. They built a conceptual model that expressed the software requirements in their point of view.

One of the authors of this paper acted as the second team in this case study and built the non-functional view by making use of structured interviews with some of the stakeholders and by using some available documentation such as ISO 9000 quality manuals.

Once the non-functional view was ready, we used the integration process showed in Section 3 and, consistently with the other case studies, all the new classes, operations and attributes that have arisen from this integration were counted as errors on the original conceptual model.

All the three case studies presented a considerable number of changes in the analyzed conceptual models as can be seen in Table 1.

Compiling the numbers, we can see that 46% of the existing classes were somehow changed to satisfice NFRs. We can also see that we found a number of new operations that represented 45% of the existing ones.

Similar number could be found for attributes. We found a number of new attributes that corresponded to 35% of the existing attributes. These numbers clearly suggest that if the strategy had been used during the development of the evaluated software, we could have got a more complete software and probably fewer demands for changes after deployment. These numbers are consistent with the number we got in previous case studies that have used a former version of the strategy [11].

In order to measure the overhead of using the proposed strategy, we first measured the time spent by software house team from the Case Study III from the initial phases of the software development until they finished the conceptual model.

They consumed a total of 1728 hrs/man of work. On the other hand, we measured the time we took to build the non-functional view and integrate it to the conceptual models. We consumed 121 hrs/man to do that. Therefore, we state that the estimated overhead is 7%. This number is also coherent with the overhead found (10%) in the case studies of [11]. The difference can be result from either minor mistakes in the measurement or to improvements introduced by the new strategy.

During this case study a third team was participating without interacting with the author and thus, paying few attention to NFR. This team was responsible for developing the software for the administrative/financial area of the laboratory.

We measured the total effort spent from the requirements elicitation to the moment prior to software deployment for both teams. Here, we considered the author as part of one of the teams. Therefore, all the time spent to integrate the NFRs was counted as time spent by the team in charge of developing the software for the processing area.

The team in charge of the software for the processing area spent 6912 hours/men from

	Existent Class	New Classes	%	Classes Affected	%	Existent Operations	Operations Found	%	Existent Attributes	Attributes Found	%
Case Study I	8	2	25	6	75	105	50	48	22	10	45
Case Study II	15	3	20	4	27	115	45	39	32	8	25
Case Study III	39	9	23	19	48	213	98	46	105	41	39

Table 1 – Results from the Case Studies

requirements elicitation to software deployment, in opposition to 8017 hours/men spent by the other team.

The first team was responsible for 52% of all the coding (measured in numbers of lines inside the programs) while the second team was responsible for 48% of all coding. We believe that consistently with the numbers presented in [11] the reason for the difference of time spent by each team was due to the effort spent on fixing problems, during the test and acceptance phases, due to NFRs not considered before.

Unfortunately, we do not have the time that teams from Case Study I and II took to build the conceptual models. We do have the time we spent to build the non-functional view and integrate it to the conceptual models.

We have consumed 45 hrs/man for the Case Study I and 21 hrs/man for the Case Study II. The difference between them lies on the use during the Case Study II of the same models from the non-functional view that we used in Case Study I. We did that because both case studies have used the same specification distributed during the Daghstuhl Seminar [12], so, the domain was literally the same and, therefore, the NFRs should not change from one case study to another.

6. Conclusion

Dealing only with functional requirements is no longer enough.

Costumers are demanding quality software, and that can only be achieved if we do consider non-functional requirements as early as possible.

Since errors due to NFR are the most expensive and difficult to correct [6] [13], not dealing or improperly dealing with NFR can lead to more expensive software and a longer time-to-the market.

This work tried to fill this gap in software development, i.e., how to achieve conceptual models that satisfice NFR.

We presented a strategy to integrate the NFR found in what we call non-functional view into the conceptual models.

This strategy is based on the use of the LEL as an anchor to build both functional and non-functional views.

Using this anchor, we showed some heuristics on how to use some of the UML artifacts to handle NFR and presented a systematic way to integrate NFR into Class Diagrams. This integration can be used in the early stages of software development, with ongoing projects or even to enhance legacy systems with NFR.

We validated our strategy performing three case studies. The results found in these case studies together with previous results [11] suggest that the use of this strategy can lead to a final conceptual model with better quality as well as to a more productive software development process.

Future work may include the extension of this strategy to other UML artifacts and to perform new case studies without any of the authors to verify how easily other developers can apply this strategy. . It is our goal to increase automation of the strategy. We plan to do research on techniques that would help reducing the clerical tasks of comparison and generation of NFR graphs.

7. Bibliography

[1] Basili, V.R. Selby, R.W., Hutchens, D.H. "*Experimentation in Software Engineering"* IEEE transactions on Software Engineering Vol. SE-12 No. 7 July 1986 pp733-742.

[2] Boehm, B. "*Characteristics of Software Quality"* North Holland Press, 1978.

[3] Boehm, Barry e In, Hoh. "*Identifying Quality-Requirement Conflicts"*. IEEE Software, March 1996, pp. 25-35

[4] Breitman,Karin Koogan, Leite J.C.S.P. e Finkelstein Anthony. *The World's Stage: A Survey on Requirements Engineering Using a Real-Life Case Study*. Journal of the Brazilian Computer Society No 1 Vol. 6 Jul. 1999 pp:13:37.

[5] Breitman, K.K. *"Evolução de Cenários"* Ph.D. Theses at PUC-Rio May 2000.

[6] Brooks Jr.,F.P.*"No Silver Bullet: Essences and Accidents of Software Engineering"* IEEE Computer Apr 1987, No 4 pp:10-19, 1987.

[7] Chung L., "*Representing and Using Non-Functional Requirements: A Process Oriented Approach*" Ph.D. Thesis, Dept. of Comp.. Science. University of Toronto, June 1993. Also tech. Rep. DKBS-TR-91-1.

[8] Chung, L., Nixon, B. *"Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approa*ch*"* Proc. 17th Int. Con. on Software Eng. Seatle, Washington, April pp: 24-28, 1995.

[9] Chung, L., Nixon, B., Yu, E. and Mylopoulos,J. "*Non-Functional Requirements in Software Engineering"* Kluwer Academic Publishers 2000.

[10] Cysneiros, L.M. and Leite, J.C.S.P. *"Integrating Non-Functional Requirements into data model"* 4th International Symposium on Requirements Engineering – Ireland June 1999.

[11] Cysneiros,L.M., Leite, J.C.S.P. and Neto, J.S.M. *"A Framework for Integrating Non-Functional Requirements into Conceptual Models*" Requirements Engineering Journal – Vol 6 , Issue 2 Apr. 2001, pp:97-115.

[12] Cysneiros, L.M. "Requisitos Não Funcionais: Da elicitação ao Modelo Conceitual" Ph.D. Thesis, Dept. de Informática PUC-Rio, feb 2001.

[13] Davis, A. *"Software Requirements: Objects Functions and States"* Prentice Hall, 1993.

[14] Fenton, N.E. and Pfleeger, S.L. *"Software Metrics: A Rigorous and Practical Approach"* 2nd ed., International Thomson Computer Press, 1997.

[15] Hadad, Graciela et. al. "*Construcción de Escenarios a partir del Léxico Extendido del Lenguage"* JAIIO'97, Buenos Aires, 1997, pp. 65-77.

[16] Keller, S.E. et al *"Specifying Software Quality Requirements with Metrics"* in Tutorial System and Software Requirements Engineering IEEE Computer Society Press 1990 pp:145-163

[17] Kirner T.G. , Davis A .M. , *"Nonfunctional Requirements of Real-Time Systems"*, Advances in Computers, Vol 42 pp 1-37 1996.

[18] Lindstrom, D.R. "*Five Ways to Destroy a Development Project"* IEEE Software, September 1993, pp. 55-58.

[19] Leite J.C.S.P. and Franco, A.P.M. *"A Strategy for Conceptual Model Acquisition "* in Proceedings of the First IEEE International Symposium on Requirements Engineering, SanDiego, Ca, IEEE Computer Society Press, pp 243-246 1993.

[20] Leite J.C.S.P., Oliveira, A.P.A., *"A Client Oriented Requirements Baseline",* Proc of the 2nd IEEE International Conference on Requirements Engineering, 1995.

[21] Neto, J.S.M. "I*ntegrando Requisitos Não Funcionais ao Modelo de Objetos"* M.Sc. Dissertation at PUC-Rio, Mar/2000.

[22] Leonardi, Carmen. et. al. "*Una Estrategia de Análisis Orientada a Objetos basada en Escenarios"* Actas II Jornadas de Ingeniaria de Software JIS97, Donstia, San Sebastian, España, Set. 1997.

[23] Mylopoulos,J. Chung, L., Yu, E. and Nixon, B*., "Representing and Using Non-functional Requirements: A Process-Oriented Approach",* IEEE Trans. on Software Eng, 18(6), pp:483-497, June 1992.

[24] Rational et al, *"Object Constraint Language Specification"* 1997. Http://www.rational.com.

[25] Rumbaugh, J., Jacobson, I. and Booch,G. *"The Unified Modeling Language Reference Manual"* , Addison-Wesley, 1999.