The Journal of Systems and Software 82 (2009) 1642–1656

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Tactics based approach for integrating non-functional requirements in object-oriented analysis and design ☆

Tegegne Marew [a,*], Joon-Sang Lee [b], Doo-Hwan Bae [a]

[a] CS Division KAIST College of Information Science and Technology Daejon 305-701 Republic of Korea
[b] Information Technology Laboratory, LG Electronics Advanced Research Institute, Seoul 137-130, Korea

ARTICLE INFO

Article history:
Available online 28 March 2009

Keywords:
NFRs analysis and design
Software design
NFR framework
Aspect oriented software design

ABSTRACT

Non-Functional Requirements (NFRs) are rarely treated as "first-class" elements in software development as Functional Requirements (FRs) are. Often NFRs are stated informally and incorporated in the final software as an after-thought. We leverage existing research work for the treatment of NFRs to propose an approach that enables to systematically analyze and design NFRs in parallel with FRs. Our approach premises on the importance of focusing on tactics (the specific mechanisms used to fulfill NFRs) as opposed to focusing on NFRs themselves. The advantages of our approach include filling the gap between NFRs elicitation and NFRs implementation, systematically treating NFRs through grouping of tactics so that tactics in the same group can be addressed uniformly, remedying some shortcomings in existing work (by prioritizing NFRs and analyzing tradeoff among NFRs), and integration of FRs and NFRs by treating them as first-class entities.

1. Introduction

The analysis and design of Functional Requirements (FRs) has received much attention since the early days of software engineering. Not to mention various analysis/design methods, a number of approaches including waterfall (Ghezi et al., 1991), iterative (Spencer and Bittner, 2006), and Rational Unified Process (Jacobson et al., 1999) have been proposed for capturing those functional requirements. In addition, modeling languages such as Unified Modeling Language (UML) (Booch et al., 1999) make it easier, by providing a common vocabulary for every stakeholder, to capture, analyze, and document FRs (among other advantages). Therefore, software developers often dwell on eliciting, analyzing, designing, implementing, testing, and maintaining FRs when they are working on software projects.

On the other hand, Non-Functional Requirements (NFRs) such as performance, security, and usability often are incorporated, if they are ever included, into the final phase of software development as an after-thought. Still, complex and expensive software have failed miserably because of improper management of NFRs. However small, a number of approaches have been proposed for capturing and analyzing NFRs.

We believe there is room for improvement for handling NFRs. Of the shortcomings of the various approaches, the common one is the failure to fully incorporate NFRs into every phase of the software life cycle and addressing them systematically. Our belief is that there is enough research regarding systematic methods for NFRs elicitation and NFRs implementation. What we need is an approach that eases the burden in NFRs analysis and design. In addition, such approach must address all NFRs. Admittedly, some of the approaches actually help in the analysis and design of specific subset of NFRs. Unfortunately, it is difficult if not impossible to employ such approaches (as attested sometimes by the developers of those approaches themselves) for some very important NFRs. For instance, approaches that use AOSD (http://www.aosd.net/) hardly are appropriate for NFRs that affect some part or the whole of the software architecture. On the other hand, architecture-centric approaches rarely are suitable for NFRs that deal with single classes/components. Therefore, we need an approach that attempts to cover all the different types/kinds of NFRs.

On top of integrating NFRs into every phase of the software life cycle, any approach that deals with NFRs has to be concerned with tradeoff analysis among NFRs to handle competing requirements. Management of NFRs is complicated because the approaches used to realized some NFR may have a positive or negative impact on another NFR. Often tradeoff analysis involves the systematic treatment of interdependency among NFRs.

☆ This research was supported by the Ministry of Information and Communication (MIC), Korea, under the Information Technology Research Center (ITRC) support program supervised by the Institute of Information Technology Advancement (IITA) (IITA-2008-(C1090-0801-0032)).

* Corresponding author.
E-mail addresses: tegegnem@se.kaist.ac.kr, tegegnem@gmail.com (T. Marew), lovelee@lge.com (J.-S. Lee), bae@se.kaist.ac.kr (D.-H. Bae).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.03.032

One factor that can be used to facilitate the analysis of NFRs' interdependency is prioritization. Prioritization refers to the relative importance the user puts on the different NFRs. For instance, in an Automated Teller Machine (ATM) application users often put much more emphasis on confidentiality than usability. Such prioritization is useful for negotiating during conflicts. Existing approaches either do not address prioritization at all or do not use the prioritization result effectively. Therefore, we need an approach that elicits users' preference among NFRs and use such information extensively during conflict resolutions among NFRs.

The existing approaches readily recognize the effect of interdependencies among NFRs by providing a number of qualitative methods to calibrate these interdependencies. Such methods are can be made more effective by incorporating prioritization information (even though they identify critical NFRs (more discussion is given in Section 8)) and quantitatively capturing interdependencies so that more freedom is given to the developer in capturing range of interdependencies. We believe a better approach, qualitative as well as quantitative, can be developed to understand these interdependencies and used to resolve conflicts among them.

To sum up, our research objective is to develop an approach for systematic incorporation of NFRs into the analysis and design phases of the software life cycle (that readily concentrates only on FRs) while settling any conflict among NFRs through tradeoff analysis.

The rest of this paper is organized as follows. In Section 2, we briefly discuss the research works that lay the ground work for our research. The following section, Section 3, presents our tactics' types classification scheme that is used throughout this paper. Section 4 explains in detail our proposed approach for integrating NFRs in object-oriented analysis and design. Subsequently, in Section 5, we discuss our approach for managing interdependencies among NFRs and resolving any conflict that may result from such interdependencies. In Sections 8 and 6, we discuss related research works and present case study, respectively. Finally, Section 9 summarizes our research and briefly sketches the future direction.

2. Background

The concepts in this paper closely depend on ideas contained in two works which we discuss in this section. One is the NFR Framework (Mylopoulos et al., 1992; Chung and Nixon, 1995; Chung et al., 2000) that helps us capture the NFRs of a given system. The other is classpects (Rajan and Sulllivan, 2005) which tries to combine the concept of class with the concept of aspect.

The NFR Framework is a qualitative approach for eliciting and analyzing NFRs. It treats NFRs as softgoals that need to be satisficed instead of satisfied. A goal is *satisficeable* if an expert after refining the goal into the subgoals deem the subgoals satisfactorily achieve the goal (Mylopoulos et al., 1992; Chung and Nixon, 1995; Chung et al., 2000). Originally used in Artificial Intelligence by Nilsson (1971), a *satisficeable* goal refers to solvable problem as opposed to a *deniable* goal that refers to unsolvable problem.

SIG, Softgoal Interdependency Graph, plays a central role in viewing and analyzing NFRs. In this paper, we utilize the fact that the NFR Framework decomposes NFR along both the topic, what kind of NFR it is, and the parameter, the entity in the system the NFR is applied to. The AND/OR decomposition of NFRs suggested by the NFR Framework also is used to capture NFRs as classpects.

One of the important features of the NFR Framework is the management of the interdependencies between different NFRs. The dependencies can be one of make, hurt, break, help, and undetermined. The Framework also has guidelines on how to combine these different contributions.

The other research work we use in our work is the notion of classpects (Rajan and Sulllivan, 2005) for leveraging the advantages of both class and aspects. Classpects can behave like traditional classes to be used as a blueprint for objects in OOSD. Moreover, classpects can exhibit behavior of aspects by capturing cross-cutting concerns and by possessing the required syntax and semantics for advising and/or be advised by other classpects. We employ classpects because it significantly improves the compositionality of aspect modules, expanding the program design space from the two-layered model of AspectJ-like languages (AspectJ) to include hierarchical structures. By allowing the joinpoints of aspects to be named in the method-join point binds, aspects not only can advise base classes but other aspects. From the example in Fig. 1, we can see the joinpoint, *ExceptionHandler*, is named. Therefore, if another classpect wants to advice this joinpoint, there is no problem of identifying *ExceptionHandler*.

3. NFRs tactics types classification

Often, a research about NFRs focuses on either a single NFR (e.g. performance) or a subset of NFRs (for instance, ATAM (Clements et al., 2002) on NFRs that require architectural transformations and AOP on NFRs that introduce new cross-cutting functionality). Our belief is we can solve this problem by *focusing on the tactics that are used to realize an NFR instead of the NFR itself.* Tactics (borrowed from ADD (Bachmann et al., 2005)), also known as operationalization in NFR Framework (Mylopoulos et al., 1992) and architectural tactics in ATAM (Clements et al., 2002), can be defined as any approach (programmatic, architectural design, process, etc.) used by a software developer to fulfill an NFR.

Fig. 2 depicts the primary focuses of users and software developers with respect to NFRs. Users often deal with the vague notions of widely known qualities such as security, maintainability, etc. On the other hand, developers, regarding NFRs, spend significant portion of their time answering how to fulfill those NFRs and the relationship among them.

Our investigation of the different tactics used for various NFRs (over 100 for security, accuracy, performance, maintainability, etc.) has resulted in the following two smilingly obvious observations are

- The tactics require quite varied approaches to implement them. For instance, authentication (for security) may need new operations while cohesion improvement (for maintenance) requires redesign (architectural transformation).
- The tactics can be grouped into categories that facilitate uniform treatment of members of each category. For instance, both authentication (for security) and exception handling (for reliability) can be implemented by introducing additional operations on the classes/components that need security or reliability.

These observations led us to NFR tactics' grouping that can be shown in Fig. 3 and is discussed below.

Since our objective is to integrate NFR analysis and design into existing Object-Oriented Software Development (OOSD) phases, we started by categorizing tactics into Tactics that Affect Analysis

```
class Exception {
  pointcut exception():
    execution(* *(..))&& !within(ExceptionHandler);
  static after exception(): ExceptionHandler();
  public void ExcpetionHandler() {
     /* Handle the exception */
   }
}
```

Fig. 1. Example of classpect code.

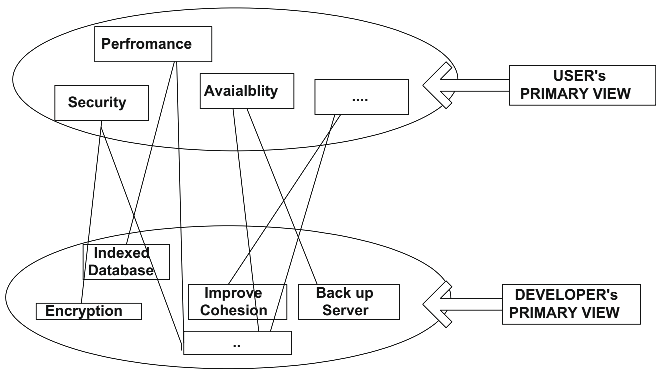


Fig. 2. The different primary focuses of users and developers.

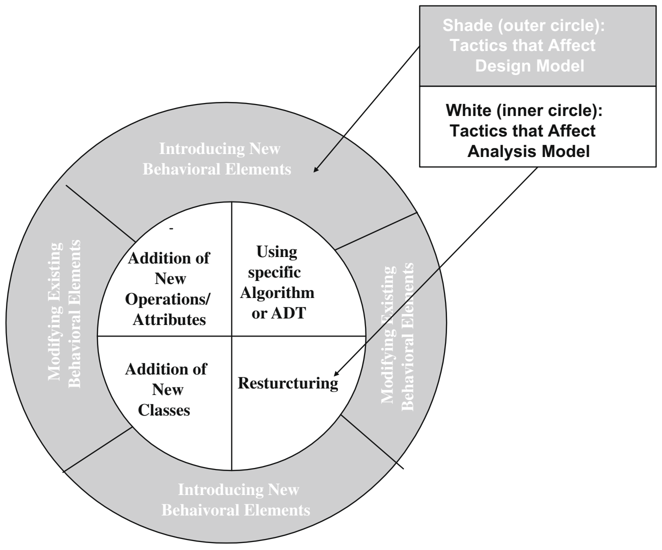

Shade (outer circle): Tactics that Affect Design Model
White (inner circle): Tactics that Affect Analysis Model
Introducing New Behavioral Elements
Modifying Existing Behavioral Elements
Addition of New Operations/ Attributes
Using specific Algorithm or ADT
Addition of New Classes
Resturcturing
Modifying Existing Behavioral Elements
Introducing New Behaivoral Elements

Fig. 3. Classification of fundamental tactics' types of NFRs.

Model (AT) and Tactics that Affect Design Model (DT). Such classification is the result of two important observations. First, some tactics need only the information available in the analysis model while some need more information that is added/refined in the design phase and is available in the design model. Second, augmenting the analysis model by including the effect of tactics in the category AT results in analysis model that is rich with information not only from FRs but NFRs and thus provides the addition/refinement that takes place during design phase with a more realistic model of the software the user required.

Among ATs, we have the tactics that are grouped under tactics' types for Addition of Operations/Attributes, Addition of New Classes, Restructuring, and Using Specific Algorithm or ADT. To see why this classification is needed, we have to remember that the classification is based on the difference in implementation techniques.

For the tactics that introduce new functionality we can use aspects or just add the operations in existing classes. A good example of such tactic is "log in" for authenticating users. We normally use aspects or just add a new operation. On the other hand, usability NFR often requires stand alone classes that are used during the initialization and the closing of the system. For such NFR, the tactic used introduces new classes for the system.

The tactics in Restructuring refers to those tactics that require a specific order of operations, deletions of operations, etc. For instance, the tactic "first-order" improves performance by requiring components to execute their operations in the order where the critical (time consuming) tasks are started the earliest. This tactic does not introduce its own operations rather requires rearrangement of existing operations of a component. On the other hand, the tactics that Using Specific Algorithm or ADT require a certain component or groups of components to be developed for implementing a specific algorithm or ADT. The performance and reliability communities have a variety of algorithms for improving those NFRs. These tactics require the developer to use such algorithms in the development of the classes.

Among the DT, we have tactics that Introducing New Behavioral Elements and Modifing Existing Behavioral Elements. This classification is based on the observation that some tactics extend behavior of the system to satisfy an NFR while the rest modify the behavior of the functional model of the system. These tactics often work hand in hand with the structural model in achieving their goals.

One important needed to be made about this classification is that it is basic/fundamental tactics' types classification. Therefore, a typical tactic can have aspects/behaviors that encompass one or more of that tactics' types described. As a result, we can have tactics that have characteristics of one or more of the groups in the tactics' types classification outlined above.

What we mean by fundamental is that we believe higher level (thus, more non-trivial tactics) can be decomposed into tactics that fall into the Classification. For instance, one tactic for improving maintainability is "anticipate changes". A closer look at this tactic reveals that, we have to decompose it further based on the context of a particular system we are going to develop. For instance, we may try to use Generic Data Types or Templates (from programming view) in our implementation of the system so that the system can accommodate further changes. This would fall into "Modifying Existing Design Elements" data structure. We may also want to include specific algorithms that may not be necessary right now but may be useful in the future (which means we have a tactic in the group) "specific algorithm or ADT". Furthermore, we may need to add additional features for the software that may not be useful right now but we expect that will be important as the software evolves. That can be grouped under "Add new behavioral elements". Therefore, a higher level tactic such as "anticipate changes" can be decomposed further to other tactics that fall into the Classification.

4. Tactic-based NFRs modeling

4.1. Overview

Our research can schematically be depicted as shown in Fig. 4. The diagram begins with the requirements elicitation and finishes with the implementation by going through the phases prescribed by an object-oriented approach. We augmented this process by introducing phases that are relevant for NFRs modeling in the analysis and design phases. As can be seen from Fig. 4, the left branch after the requirements classification consists of phases widely known for capturing FRs. In the right branch we have the NFRs modelling phases which are later combined with the FRs modelling phases.

We can have a bird's-eye view of the process by quickly going through the steps. Requirements Elicitation, Requirement Classification, and Use Case Modelling and Analysis are standard OOAD phases that result in the analysis model. It is here we start to introduce new phases for NFRs modeling, *Tactic Classification*. We first need to design the Softgoals Interdependencies Graph (SIG) based on the NFR Framework (Mylopoulos et al., 1992; Chung and Nixon, 1995; Chung et al., 2000) and classify the tactics that realize the NFRs according to the classification scheme in Section 3.

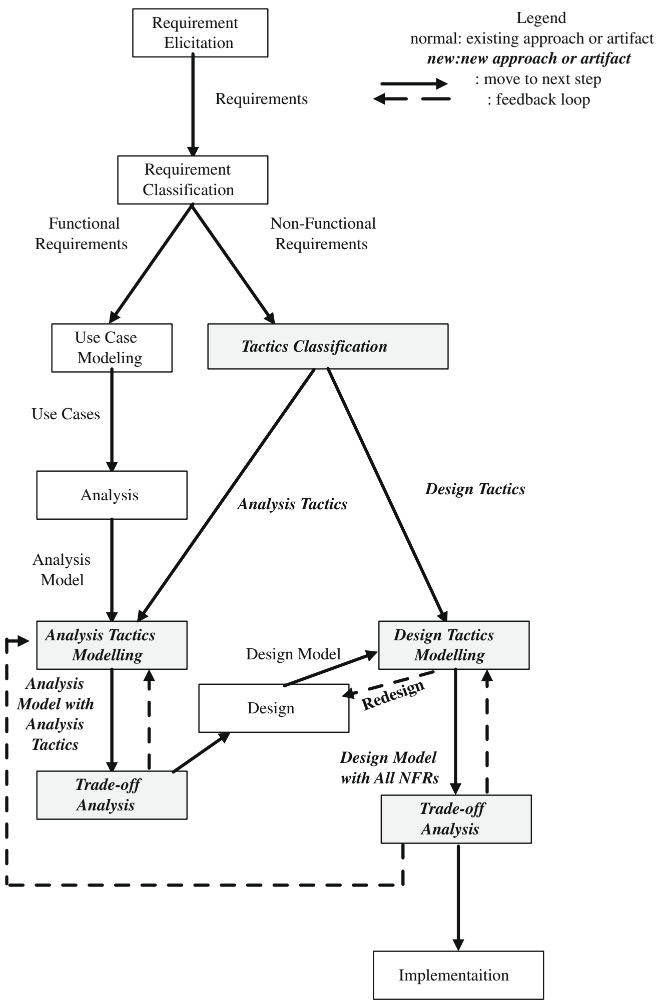


Fig. 4. Overall flow of tactic-based NFRs modeling.

Tactic Classification is followed by *Analysis Tactics Modeling*. This phase accepts the analysis model of the analysis phase and the analysis tactics of the tactics classification phase. In this phase, the analysis tactics are modeled using classpects (Rajan and Sulllivan, 2005), classes, new algorithms, etc after thorough understanding of the analysis model. After AT Modelling, we have *Tradeoff Analysis*. There are two tradeoff analysis phases in the diagram: the first after the analysis tactics modeling phase and the second after the design tactics modeling phase. During these phases, the relationship among NFRs is analyzed. Sometimes the tactic chosen to realize one NFR has adverse effect on another. The analysis of theses phases may force us to go back to other phases, Analysis Tactics Modelling or Design Tactics Modeling, and redo the modeling. This is followed by the normal OOAD phase of Design.

The Design phase is followed by *Design Tactics Modeling* which uses the design model with the design tactics of the tactics classification to model those NFRs. As an output of this phase, we have an integrated design model that satisfies both the NFRs and FRs. Finally, the Implementation phase realizes the system that incorporates both the FRs and NFRs of the user.

This overview presents how we approach NFR modelling in OOAD. It is important to note that the overview is a process of processes. Each process will be discussed in the coming sections where we lay out the roles and responsibilities of the developer and the activities he has to accomplish.

4.2. Analysis tactics modeling

4.2.1. Process for analysis tactics modeling

This is the first point of contact between FRs and NFRs. We have the analysis model of FRs and the analysis tactics of NFRs. All the different groups in this category (Addition of New Operations/ Attributes, Addition of New Classes, Restructuring, and Using of Specic Algorithm or ADT) employ one or more of the processes depicted in Fig. 5 and discussed below.

Step 1: Creating SIG

The first step, creating SIG, is already discussed elsewhere NFR Framework (Mylopoulos et al., 1992; Chung and Nixon, 1995; Chung et al., 2000).

Step 2: Synthesizing classpects from SIG

At this stage, we use rules that are designed to transform a part of the SIG to a set of classpects based on information in the SIG as well as the class diagram of the analysis model. In the beginning, the classpects are very much skeletal in that the actual classes they advise and the advises they give are not completely discovered. To fully specify the advises, we need to discover the classes that the classpects are supposed to advise. We do that at Step 3 of the approach. Once we discover such classes we need to fill out the actual advises the classpects give to those classes since the advises the classpects give so far is skeletal as explained below.

Using the information from the SIG, we will discuss the rules for synthesizing classpects from a SIG. We use the textual form of SIG as suggested by Chung et al. (2000) to discuss the rules but the corresponding graphs can be seen in Fig. 6. In the textual form of a SIG, X(Y) refers to a topic (which often is an NFR) is applied to a Y (a parameter).

To save space we discuss only one of the rules in detail, **Rule 1** (for a more detailed discussion, please refer to (Marew and Bae, 2006)).

If **A1(B) AND A2(B) SATISFICE A(B)**

We have a node A(B) refined along the topic into A1(B) and A2(B). Normally, A1, A2, and A are NFRs and B is a class in the system where A1 and A2 are decompositions/refinement of A. For instance A can be security while A1 and A2 could be confidentiality and integrity, respectively. Since A, A1 and A2 are NFRs, we model them as classpects. Therefore, we create three classpects: A1, A2 and A to encapsulate the advises where A advises B while A1 and A2 advise A. At first glance, it may seem A1 and A2 should advise B. However, that would lose the information that A is refined into

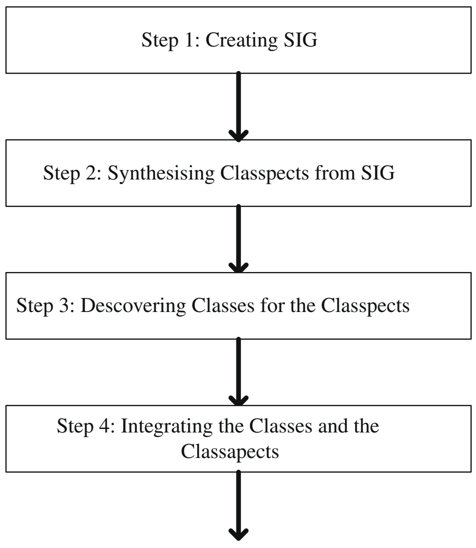

Step 1: Creating SIG
Step 2: Synthesising Classpects from SIG
Step 3: Descovering Classes for the Classpects
Step 4: Integrating the Classes and the Classpects

Fig. 5. Processes for analysis tactics modeling.

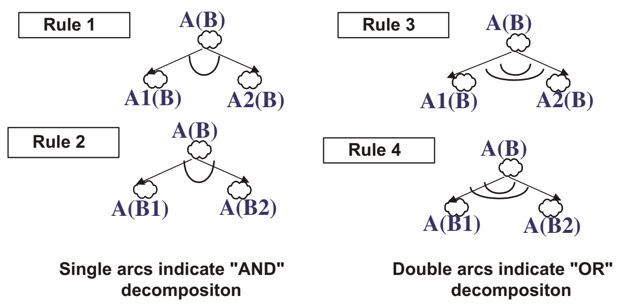


Fig. 6. The different cases of SIG for whom the rules are created.

A1 and A2. We can preserve the information if we ensure A1 and A2 advise A instead of B. We can advise A even if itself is advising B because the classpect has the dual properties of a class and an aspect. Similarly, if necessary, A1 and A2 themselves could be further advised by other classpects.

The code we generate by following such rules have the following characteristics:

- It is skeletal. The actual arguments of the methods and their bodies are not yet filled. We just insert text messages based on the tactics in the methods of the classpects. Actually how that is implemented, obviously, depends on the class that is advised.
- It preserves the hierarchical structure of the SIG. Because each rule respects the hierarchy, the resultant classpect structure mirrors the SIG. This is useful for maintainability since it facilitates traceability between the SIG and the classpects.

We can see the application of such rules using Fig. 7 borrowed from (Cysneiros and Leite, 2004). Here the white clouds represent softgoals and the shaded clouds represent operationalization/tactics. The decomposition of the root is via attributes. The skeletal code of the corresponding classpects generated using the rules discussed above can be seen in Fig. 7. In the classpect SafetyRoomMal, lines 2–4 describe the constructor which designates a Room class (here we are not sure if there is a room class in the functional requirement but we use it as the first approximation since Room is used in all softgoals in the SIG as a parameter) as one of its own attributes. This helps the classpects to access the attributes of Room if there is a need to do so. In lines 5–7, we have the method that advises the class Room and in lines 8–10 we have the crosscut specifications describing which methods and when (before, after, around) the advice occurs.

Even if we only show in Fig. 7, the classpects synthesized from the root and the node *safety [Room.Malfunction.OI]*, the same can be done for the other softgoal, *safety* [*Room.Malfunction.Motion Detector*]. The node "set room as occupied" will be recorded in the advice method and later (as shown in the "Discovering Classes for Classpects" step), the information in the corresponding functional model element (class) is used to convert "set room as occupied" into source code.

Step 3: Discovering classes for classpects

Once we synthesize the classpects for the NFRs identified in the SIG, we need to find which classes in the functional requirements the identified classpects apply to (advise). There are a number of approaches achieving this objective. These include.

- For both NFRs and FRs elicitation, use the same vocabulary. Used by Cysneiros and Leite (2004), this method ensures the parameters of the NFRs in the SIG are also used in the FRs elicitation. Unfortunately, this approach requires both FRs and NFRs analysts to precisely know the objects they have identified early on. The NFRs' analyst should understand what the FRs analyst meant to say by that term, syntactically and semantically, and see if that is exactly what he (the NFRs analyst) also wants to model.
- Start with use cases and consider the classes that collaborate to implement the use cases. Often a specific NFR can be seen as if it applies to a particular use case(s) since use cases are used for functional decomposition and the quality of each functionality is captured by a specific NFR. For instance, if we have a use case *withdrawal*, an FR, in an ATM system, we can easily see security, an NFR, is a required quality. Once we identify the use case, we can investigate the classes in the sequence diagram that implements the use case to see which class(es) the NFR should be applied to.
- Use the requirements document. Usually, a specific NFR is mentioned along side with the entities it applies to. If those entities are later decided to be realized as classes, then the classpect that realizes the NFR can be applied to those classes.

Step 4: Integrating the classes and the classpects

At the final step, we have to integrate the classpects we synthesized from the SIG in the step 2 and the classes we determined to be advised by these classpects in the Step 3. In the SIG, the leaves of the tree structure are operationalizations/tactics. These are captured in the body of the methods of the classpects as we saw Step 2. However, the information required to realize these methods is not completely available in the classpects. Such information (being functional requirement related) is available in the class diagrams.

Now using the information from the class diagram we fill out the skeletal code. When we fill out the code we have to realize one important difference between the information contained in the SIG and UML diagrams. In UML diagrams, we don't have imple-

```
public class SafetyRoomMal{
1   Room rm;
2   public SafetyRoomMal(Room rm){
3      this.rm = rm;
4   }
5   public void safteyfun(){
6     inform facility manager
7   }
8   before execution{(public * Room.*(*))
9   && returns (ret) && args():
10   call safetyFun();
}
```

(a)SafetyRoomMal Classpect

```
public class SafetyRoomMalOI{
  SafetyRoomMal srm;
  public SafetyRoomMalOI(SafetyRoomMal srm){
     this.srm = srm;
  }
  public void safteyfun (){
     all ceiling lights on
     and inform user
  }
  before execution(public *
           safetyRoomMal.*(*))
  && returns (ret) && args():
  call safetyFun();
}
```

(b) SafetyRoomMalOI Classpect

Fig. 7. An example for application of the rules.

mentation details. Yet, in the SIG, operationalization nodes capture implementation details. Since NFRs are quite general and their realization is quite independent of the particular application we are developing, it is possible to incorporate the information at the SIG (now in the classpects) to the functional design documents. If that is not desirable, the implementor can fill out the skeletal classpects himself/herself.

4.2.2. Application of analysis tactics modeling on the various analysis tactics

Tactics in the group "Addition of operations/attributes"

These tactics are very common among some NFRs such as in security that the IEEE ISO 9126 (ISO/IEC, 2001) groups as *functionality*. All the different steps of Section 4.2.1 are needed to model these tactics.

Tactics in the group "Addition of new classes" Such tactics could be handled just as if they were FRs. They need new classes to satisfy the corresponding NFRs. We create the class with the required behavior. Even if we use classpects for both as aspects and classes, there is an important difference between these kinds of classpects and the ones created by tactics in the previous section: these classpects do not advise any other classpect. Thus, they act as classes in traditional object-oriented analysis and design.

Tactics in the group "Restructuring"

These tactics require steps 1, 3 and 4 of Section 4.2.1 so that once the developer understands which class the tactic needs to be applied to, he remodels the class according to the tactic. Structural rearrangement of operations is often enough without adding additional code.

Tactics in the group "Using a specific algorithm or ADT"

Again we need steps 1, 3 and 4 of Section 4.2.1. The difference being now the class we discover in step 3 is remodelled according to a specific algorithm or use a new ADT. For instance, if we decide to use indexing for our database so that performance is improved, the class/es that deal with the database may need to employ a specific data structure, say binary tree, for facilitating such operations. In such event, the class/es need to include all the operations and attributes for managing binary trees.

Among the four groups of tactics we discussed, the first one is the most common whereas the others are not even addressed by other approaches because they are less common. For example, in most of AOP based approaches such as (Xu et al., 2006), the other three categories are hard or even impossible to implement.

4.3. Design tactics modeling

We use the artifacts of the design phase to model the design tactics. This phase produces a model that not only satisfies all FRs but all NFRs.

We handle the tactics in this category with what Bachmann et al. (2005) calls an intuitive approach. The tactics impose design changes on a large section of the system in heterogenous fashion. Therefore, the manner developers manage those tactics is very tactic specific. Still, they have the following general properties:

- They do not often affect the entire system; at least, there are subsystems or specific services where those NFRs are significantly important. For example, performance is more important for services that produce what the user wants. Another example is security which is only important in services that need to be confidential.
- Even in the part of the system affected by the tactic, only a small portion of that part is critical. Critical refers to the part of the system that either achieves most of the objectives of the tactic or the part that has a significant positive or negative impact on other NFRs. For instance, if we decide to improve reliability by introducing back-up system, often we need to introduce a lot of redundancies all over the system. However, the critical part would be the servers that handle the processing.
- Often we can negotiate without resorting tradeoff between two NFRs. For example, if we choose to use 128 bit encryption for improving security, this may have a strong penalty on performance. However, if we change it to 64 bit encryption, the penalty on performance might be acceptable.

Based on these properties, we follow the following steps to realize design tactics. (The approach is depicted in Fig. 8.) There are two paths: the first one (the left) is for the tactics that introduce new behavioral elements and the other is for tactics that modify existing behavioral elements. The steps are designed to address the following problems.

- For analysis tactics, the specific part of the system the tactic is applied to is easily available from the parameter part in the SIG. For instance, when a "encryption" is the operationalization/tactic of the node *encryption [Account*], we can deduce that encryption is applied to an Account parameter which most probably is modelled as a class. On the other hand, if we have "reduce coupling" as a tactic for *maintainability [system*], we have to find out which part/s of the system has high coupling and reduce its/theirs coupling.
- Since design tactics often have a wide reaching effect compared to analysis tactics, the introduction of a given tactic often has a serious positive/negative impact on other NFRs. To facilitate the analysis of such effect, we need to find out which part of the system that is affected by the tactic has an impact on other parts of the system.

For each tactic

- *Step 1:* Based on the SIG and the design model, find the part of the system that is affected by this tactic. We can either use NFR-specific theory to predict the effect of the tactic and thus find the parts of the system affected by the tactic or use an expert's opinion to find out which part of the system is affected by the tactic. For instance, if it is an "authentication" tactic, the part of the system affected is the "log-in" sequence diagram. This is determined by mainly the expert's opinion or by applying an NFR-specific theory. To see an example of using an NFR-specific theory let us find out what part of the system is affected when a sensor is modified using abstraction (borrowed from

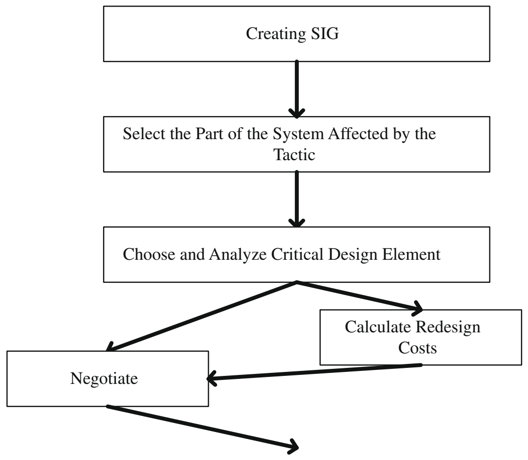

Creating SIG
Select the Part of the System Affected by the Tactic
Choose and Analyze Critical Design Element
Calculate Redesign Costs
Negotiate

Fig. 8. Process for design tactics modeling.

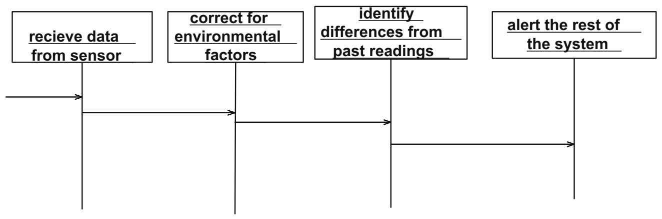

recieve data from sensor
correct for environmental factors
identify differences from past readings
alert the rest of the system

Fig. 9. Finding out the part of the system affected by the tactic.

(Bachmann et al., 2005)). The different parts of the system needed for this discussion are depicted in Fig. 9. The NFR we are trying to improve is maintainability (reduce the amount of time that takes to make further modification on the system) and the tactic is "abstraction". Here the problem is the "receive data from sensor" subsystem will be affected when the sensor is modified and that modification will ripple to the other parts of the system. By applying Impact Analysis Theory (Bachmann et al., 2004) (an example of NFR-specific theory), we find the part of the system that are affected when "abstraction" is applied (Fig. 9).

- *Step 2:* Choose a critical design element affected by the tactic. This may be a portion of the system whose modification according to the tactic brings a significant improvement on the NFR or have a positive/negative impacts on other NFRs. For instance, for a cryptography tactic the critical design element is the database. Note that if this tactic has an impact on more than one NFR, there may be more than one critical element for this tactic. To identify the critical design element, one can
  - Use an NFR-specific theory to assess the effect of applying the tactic,
  - Choose the element that can be considered to be heavily affected by the tactic or
  - Apply another NFR-specific theory (for another NFR) to find a design element whose modification by this tactic affects the other NFR.

In the previous sensor example, "receive data from sensor" subsystem is critical because it has the highest cost of modification (calculated according to Impact Analysis Theory) as depicted in Fig. 10.

- *Step 3.* Calculate redesign costs: For any tactic, there is at least a cost of introducing that tactic to the system. This might be adding new classes, modifying an existing class according to a new algorithm, etc. However, for those tactics that are grouped under "Modify existing elements", there is the danger of the ripple effect of such modification. For instance, in the on-going example of "receive data from sensor", the cost of introducing the tactic costs 1 day and adjusting the rest of the system costs an additional 0.8*1 + 0.8*0.2*1 + 0.8.0.2.0.2*1 or 1 day using Impact Analysis Theory in (Bachmann et al., 2004). Thus, if we want to employ this tactic, we should see if such redesign cost is affordable.
- *Step 4.* Negotiate: Try to negotiate the tactic and the NFRs on which the tactic has a negative impact. For instance, instead of using three levelled back-up system (to improve reliability as expense of slow performance), we may settle for two levels. If this fails use Q-SIG (Section 5), to decide if the tactic is good from the entire system's point of view.

5. Tradeoff analysis

After Analysis Tactics Modeling and Design Tactics Modeling, often we have to analyze the tradeoff of the different tactics. This may lead us to go back and remodel the NFRs. To facilitate the tradeoff analysis we need to prioritize the NFRs according to the user's view of the system. Coupled with prioritization, we use Q-SIG (Quantified SIG) to arbiter between different competing requirements.

5.1. Prioritization

Most of the existing research regarding NFRs rarely consider the relative importance users put on the various NFRs. Even if there are various methods to rank entities of concern based on multiple criteria, we choose to use Analytical Hierarchical Process (AHP) (Saaty and Vargas, 1991; http://mat.gsia.cmu.edu/mstc/multiple/node4.html) for prioritizing NFRs. AHP is simple (shallow learning curve), can be automated, highly mature (have been used for around 30 years), uses quantitative measure and has clear-cut steps (Mead). The NFRs are often similar across projects eliminating the need to substantially modify the table or analysis done in earlier projects. Besides, a number of researches in software engineering have used it in a variety of contexts including prioritizing NFRs (Karlson and Ryan, 1997; Zhu et al., 2005). Unfortunately, those researches that used AHP for prioritizing NFRs have not gone far enough to suggest a methodology/approach for analyzing and designing NFRs.

We use a more simplified version of AHP where we only use AHP to rank NFRs among themselves (called "criteria" in AHP terminology). If we use the full AHP, we need "alternatives" that would fulfill the various "criteria" to various degrees. In case of NFRs, those alternatives could be different system designs that satisfy the NFRs. That would require us to build different system designs (use cases, class diagrams, sequence diagrams, etc.) for each alternative design which is very impractical.

5.2. Q-SIG

In this section, we present our enhanced version of SIG, Q-SIG. Q-SIG is a quantified version of SIG. Instead of assigning qualitative descriptions like "denied", "satisficed", and "break", we quantify those subjective measurements. This enables us to denote continuous shades of satisfaction and to further algebraically analyze such descriptions to see the big picture. A typical Q-SIG is shown in Fig. 11. The enhancements on a normal SIG are:

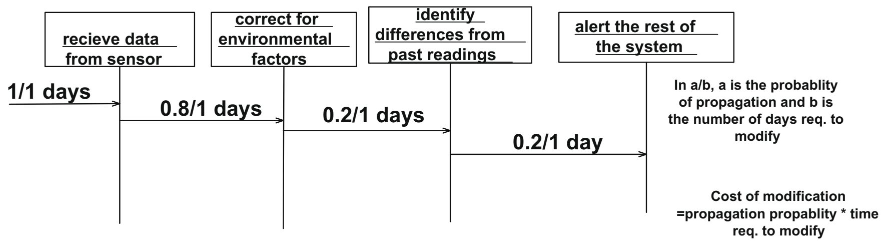
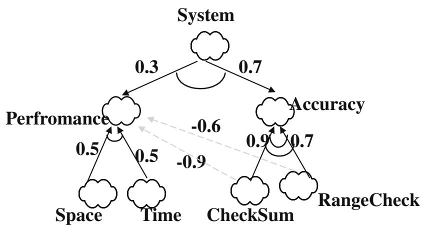

recieve data from sensor
correct for environmental factors
identify differences from past readings
alert the rest of the system
1/1 days
0.8/1 days
0.2/1 days
0.2/1 day
In a/b, a is the probablity of propagation and b is the number of days req. to modify
Cost of modification =propagation propablity * time req. to modify

Fig. 10. Finding the critical part with respect to the tactic.



Single arcs denote "AND" decompostion
Double arcs denote "OR" decomposition
Dashed arrows indicate "negative" contribution

Fig. 11. Example of Q-SIG.

- Neither the nodes or the links are labelled qualitatively.
- Downward arrows are for decomposition and upward arrows are for signifying contributions of a child for a parent.
- "AND" children decomposition must add up to 1. For example, in Fig. 11 "Time" and "Space"'s decomposition of "Performance" add up to 1.
- The value of a node is the normalized sum of the contribution from each child. By normalized sum we mean, the sum of each child's contribution divided by the number of "AND" groups, i.e. the group of children combined by "AND" as well as the number of softgoals that are not originally refined from this goal but has a positive/negative contribution to this node.
- All leaf nodes have the value 1.
- The contribution of a child to its parent is the value of the child multiplied by the link value that connects them (we have to start from the leaves and propagate upward). For example, in Fig. 11, CheckSum's contribution to Accuracy is (1)*(0.9) which is 0.9.
- All link nodes have values in the range [−1,1]. A positive value indicates the child node helps the parent node while a negative value indicates the chid node hurts the parent node.

The objective is to get the maximum value for the root node. In the example, we can see CheckSum (a method of validation by comparing the sum of each bit with an already known value) or RangeCheck (an other method of validation by investigating if the value is within the allowed range) can be used to realize Accuracy. To decide which one to choose we can compare the value of System if we choose CheckSum or RangeCheck. The resulting graphs and the calculations can be seen in Fig. 12. We can see CheckSum should be chosen since System has a higher value (0.66) than when RangeCheck is chosen (0.61). Since we have many alternatives, we want to decrease the amount of calculation. One way to do that is by not considering those parts of the graph whose values do not change among alternatives.

As we have seen, the values in the graph come from different sources. Firstly, the decomposition numbers are the results of our prioritization (Section 5.1). Performance seems less important (0.3) compared to Accuracy (0.7). This information cannot be represented by existing SIG. AHP can be used to figure out the child's contribution to the parent too. In our example, Time and Space or CheckSum and RangeCheck's value can be decided by following AHP. Time and Space are equally important for the satisfaction of Performance, thus both have 0.5 contributions; while CheckSum gives more confidence about accuracy than RangeCheck because RangeCheck cannot be sure if the value is exactly what the user wants except that it is within the allowed range where as Checksum through adding each bit guarantees higher degree of accuracy and thus CheckSum is assigned 0.9 compared with RangeCheck's 0.7.

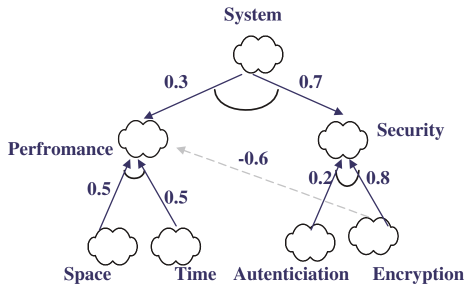
Secondly, when it comes to decide contributions of a tactic on an NFR as a side effect (positive/negative), the developer needs to use his knowledge about the tactic and the NFR (i.e. the decision is entirely subjective). In our example, CheckSum is decided to have a very strong negative impact on Performance (−0.9) compared RangeCheck's negative impact (−0.6) because CheckSum requires adding every bit of the data, thus computationally expensive, while RanageCheck requires only to see if the value is within the allowed range.

5.3. Sensitivity analysis

One weakness for trusting the Q-SIG is the subjective numerical values given by the expert. To alleviate such critical concerns, AHP and other multi-criteria decision approaches are augmented with robust and sensitivity analysis.

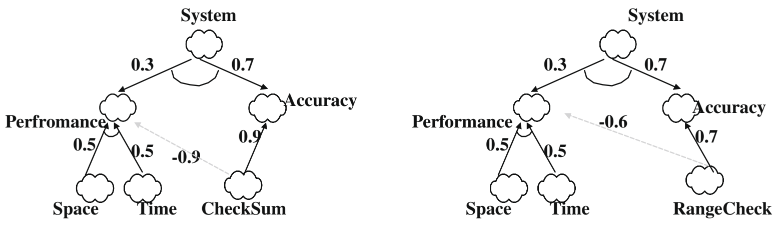
Sensitivity analysis (Zhu et al., 2005; Dobrica and Niemela, 2002; Triantaphyllou and Mann, 1994) is introduced to increase the confidence in our decision by looking at how much change in the initial weights can cause a change in our final decision.

In Fig. 13, encryption has a positive impact on security but a negative one on performance that can be analyzed by using Q-SIG. We choose to keep encryption even if it has a negative impact on performance since with encryption the overall value for the system is 0.82 which is clearly greater than the value for the system



With Encryption: 1*0.7+(1-0.6)*0.3=0.82
Without Encryption: 0.2*0.7+1*0.3=0.44

Fig. 13. Choosing between CheckSum and RangeCheck for realizing accuracy.



0.3*(1-0.9)+0.9*0.7=0.66   0.3*(1-0.6)+0.7*0.7=0.61

Fig. 12. Choosing between CheckSum and RangeCheck for realizing accuracy.

without encryption (0.44). To answer how confident we are about this decision, we first observe that it is the weight of security (0.7), that contributed the most for our decision to keep encryption. Next, we ask how much the weight of security (percentage-wise) can change for our decision to keep encryption change?

Let the w_p and w_q be the weights of performance and security respectively. In addition, let α and β be the positive and negative contributions of encryption to security and performance, respectively. Therefore, the equations for keeping and not keeping encryption become

$$1 * w_p + (1-\beta) * w_q \tag{1}$$

$$(1-\alpha) * w_p + 1 * w_q \tag{2}$$

Here, we have Eq. (1) > Eq. (2) when the numerical values are plugged in. We are looking for a new weight of security, w_r, that would bring Eq. (1) to be equal to Eq. (2). That means,

$$1 * w_r + (1-\beta) * w_q = (1-\alpha) * w_r + 1 * w_q \tag{3}$$

or

$$w_r = \frac{\beta}{\alpha} * w_q \tag{4}$$

Since the percentage-change required for w_p to be w_r is

$$\frac{w_p - w_r}{w_p} \tag{5}$$

putting Eq. (4) in Eq. (5) gives

$$1 - \frac{\beta * w_q}{\alpha * w_p} \tag{6}$$

Therefore if w_p changes by Eq. (6) (percentage-wise), the decision to keep encryption no longer holds. Numerically Eq. (6) evaluates to $1 - \frac{0.6*0.3}{0.8*0.7}$ which is approximately 68%. Since 68% is a huge change, we feel confident in that, unless the user has made a significant error in judgment, our decision to keep encryption is in line with his/her needs. Similar analysis can be carried out about the other subjective numerical values to increase confidence in our decision.

5.4. Ripple effect as a factor in applying a tactic

During our discussion on Design Tactics, one of the steps we identified to take for those design tactics that modify an existing design element is to "calculate redesign costs". Such step is necessary because applying a tactic sometimes requires us to change/modify the existing functional model and sometimes such modification might end up being too costly. Therefore, before actually deciding to accept a particular tactic, we should see how it affects the existing system.

In this section, we suggest a method for using information of a tactic's effect on the existing system as one factor whether to determine to accept to apply that tactic or look for another one. We base our approach on the work of Yau and Collofello's ripple effect algorithm (Yau et al., 1978; Yau and Collofello, 1985). Later the same algorithm was reformulated by HSue (2001) with matrix notation. In our work, we don't use the algorithm to find how a modification of a module is felt throughout the system, Rather, we assume that has already been done and use the result of such application of the algorithm to calculate the total effort required to apply the tactic. In (Yau et al., 1978), Yau and Collofello have given the formula for calculating the effort to applying a particular modification as,

$$U_a + M_a + \sum_{i=1}^{n}(U_{b_i} + M_{b_i}) \tag{7}$$

where U and M stand for understanding and modification, a is the module to be modified and the b_i's are all the modules that are affected by the modification of module a.

We present FR-QSIG (Fig. 14) as a graphical representation of a tactic's effect on the system. It is very closely related to Q-SIG and it has three basic layers. The first layer consists of a node representing the whole system. The second layer consists of nodes representing modules/classes that are potentially modified either directly or indirectly by all the tactics that are applied to the system. Finally at the third layer, we have nodes representing tactics. The edges between the tactics (third layer) and the modules/classes (second layer) can be either direct (represented by solid arrow from the tactic to the module it modifies) or indirect (represented by a broken arrow from the tactic to the module it affects because of the ripple effect of the modification the tactic started). We do not show the relationship between the modules because that would make the graph unreadable and such information does not add anything we need for our discussion.

Here, we can easily utilize Eq. (8) to calculate the modification effort required to apply a given tactic. The numbers beside each arrow (full or broken) indicate the modification effort estimated the tactic would require for a specific class. Therefore, with FR-QSIG, we can easily find by summing over each tactic, how much effort the tactic requires to be applied. Moreover, by summing over each class, we can know how much modification is going to be undertaken on each class.

As Q-SIG, FR-QSIG can be used to decide between tactics that might be used to realize identical NFR. Just comparing the amount of modification required to realize the tacite may not be the best way to compare tactics. The reason is because some tactics even if they need a high cost of application, they may be important in satisfying the software's NFRs. Therefore, we have to consider not only the modification required to realize a given tactic, but also the contribution of that tactic to software's NFRs. The contribution of a tactic to the system's NFR can be found from the QSIG of the system.

For example, in Fig. 12, CheckSum's contribution is (0.9*0.7) + (−0.9) (0.3) which is 0.36. This means, among the systems NFR, 36% is satisfied by realizing CheckSum. We will call such value, the "NFR Contribution Factor (NCF)". We can use this NCF along side the modification effort required to implement a given tactic to decide whether that tactic is worth realizing. To do that we also need to know how much of all the modification effort to implement all the tactics is going to be spent on this specific tactic. Here what we need is just the percentage of the total effort of modification that is needed to implement this particular tactic. We call this piece of information "Modification Effort Proportion (MEP)". Therefore, for tactic a_l, its MEP is given by

$$\frac{U_{a_l} + M_{a_l} + \sum_{i=1}^{n}(U_{b_i} + M_{b_i})}{\sum_{j=1}^{m}(U_{a_j} + M_{a_j} + \sum_{i=1}^{n}(U_{b_i} + M_{b_i}))} \tag{8}$$

where the denominator is the same as before (i.e. Eq. (8)), the denominator is just the total modification effort for realizing all tac-

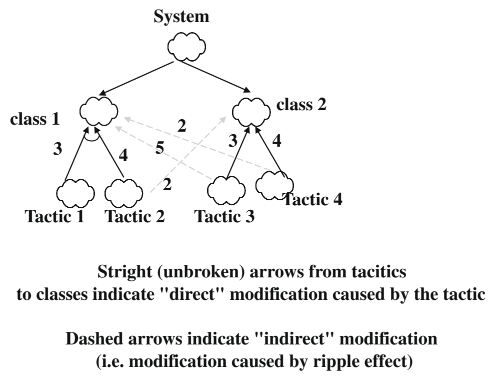

System
class 1
class 2
3
4
2
5
3
4
2
Tactic 1
Tactic 2
Tactic 3
Tactic 4
Stright (unbroken) arrows from tacitics
to classes indicate "direct" modification caused by the tactic
Dashed arrows indicate "indirect" modification
(i.e. modification caused by ripple effect)

Fig. 14. FR-QSIG, graphical representation of tactics' modification impact on the system.

tics. For example, in Fig. 12, tactic 4 has an MEP of $\frac{4+2}{4+2+3+5+2+4+3}$ which is 0.26 or 26%.

These two values, NCF and MEP, can be used to decide among competing tactics as follows:

- In software where modification effort is either negligible or not as important as satisfying NFRs, we can just use NCF to decide which tactic to apply. Software where modification effort is negligible is often software that is being built and therefore we have not yet written a lot of lines of code. Therefore, modification involves just redrawing of model documents which often requires negligible effort. In the other extreme, we have software where NFRs play a significant role (e.g. safety critical systems, embedded software, etc.) and therefore it does not matter how much we need to modify the system to satisfy a given NFR. In both instances, we may not care about modification and as a result only the tactic's NCF will decide whether that tactic is worth realizing.
- In software where NFRs do not play a significant role but modification is costly, we may decide to apply a given tactic based on its MEP. A good example of such software is an existing library system. Here almost every NFR is not very important (except for user interface). In such cases, deciding between tactics should not be based on which tactic is important for the software's fulfillment of NFR (i.e. high MCF) rather whether such tactic requires low modification (i.e. low MEP).
- In all other cases, we need to consider both NCF and MEP. We can find the ratio, which we call "N–M ratio (NMR)", between NCF and MEP to see if the tactic requires more modification than its contribution to the software NFR. For instance, if a tactic has an NCF value of 0.25 and MEP value of 0.2, it means the tactic can satisfy 25% of the user's needs but only consumes 20% of all the modifications needed to realize all the tactics proposed. Therefore, such tactic will have an NMR of $\frac{0.25}{0.20}$ or 1.25. Such value combines information from QSIG and FR-QSIG. Therefore, NMR is a more realistic measure of how useful the tactic is for both the user (because it captures how much the tactic is useful for the software NFR) and the developer (because it captures how easy/difficult is to implement the tactic). We can compare NMR values of competing tactics to decide which one to implement.

6. Case study

In this section we illustrate the different issues raised by our research by applying our approach in the construction of the NFRs for a typical Automatic Teller Machine (ATM) application. We assume the functional model of the application is already developed whose class diagram is depicted in Fig. 15 from (http://www.math-cs.gordon.edu/local/courses/cs320/ATM_Example/indexOld.html; http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/). Table 1 lists the various NFRs of the ATM and the specific tactics used to satisfy them gathered at the end of NFRs elicitation. The last column requires a special attention as it displays the specific group the tactic is classified. The group dictates what approach needs to be used to fulfill the tactic in the analysis and design phases.

Among those tactics that are classified in the group of "additional operations/attributes", we will discuss *Exception for checking cash availability* in detail. We use the steps discussed in Section 4.2 as follows:

- *Step 1: Creating SIG*: Fig. 16 shows the SIG of fault tolerance that is decomposed along fault tolerance for the ATM machine and fault tolerance for the account. In the case of the machine, we must make sure we have enough cash before the machine disburses the cash. On the other hand, for the account, we need to make sure the amount requested by the user is within both the balance of his/her account and the allowed one-time withdrawal limit (set by the bank).
- *Step 2: Synthesizing Classpects from SIG*: We first synthesize the classpect code from the SIG alone for the FaultTolerance (ATM) as follows.

```
ATM atm;
public FTATM(ATM atm){
this.atm = atm;
}
public void FaultTol(){
check cashfund
}
pointcut
call FaultTol();}
```

- *Step 3: Discovering classes for the classpects*: We try to discover the class that this classpect (FTATM) tries to advise. Since there is an ATM class in the class diagram, we may be tempted to think that ATM in the SIG and ATM in the class diagram represent the same entities. However, a closer inspection reveals that the ATM the SIG refers to is actually the CashDispenser in the class diagram since it is the class that deals with disbursing the cash. This is a typical example the difference in the NFRs and FRs analysts understanding of the system. Often such differences can be solved easily. For instance, discovering CashDispenser as the entity referenced as ATM in the SIG is not very difficult since it is associated with our first guess ATM. This particular scenario demonstrates that syntactic matching from the SIG to class diagrams may fail requiring a more careful approach.
- *Step 4: Integrating the classes and the classpects*: Finally, we integrate the information from the class CashDispenser with the advise in the classpect FTATM as shown in Fig. 17. Note that the name is also changed to FTCashDispenser. The informal "check cashfund" advise in the body of FaultTol is now changed to an if statement that uses the information about *amount* from the class CashDispenser. In the future, if any modification is made to the SIG or the class diagram we can easily trace from one diagram to the other using this integration.

The SIG displayed in Fig. 18 requires to fulfill security through the confidentiality of the userinterface and database. Authentication for userinterface and database was achieved through adding operations as in the previous cases. In case of the database, in addition to authentication, we want the database to be encrypted using AES algorithm (Advanced Encryption Standard, 2001) which is a tactic grouped under "new algorithm/ADT" (Table 1). For tactics in this group, we need to identify the class that the algorithm is applied to and then apply the algorithm (since these steps are identical as the steps for the previous example, we do not need to discuss them in detail). After identifying that database in the SIG is actually Bank in the class diagram (since bank is the class that manages the database), we needed to introduce the necessary operations, attributes, and logic to fulfill the AES algorithm. This required a complete rework of the existing class Bank.

For the case of usability, the existing class diagram already contains classes like display and operation panal which serve as part of userinterface. Since the tactic *GUI* is classified under "New classes", we created GUI versions of existing userinterface classes.

In case of maintainability, we are required to reduce coupling which is a tactic grouped in "modify behavioral elements" since reducing coupling requires to analyze inter-class behavior and find classes to modify. Consequently, existing behavioral elements

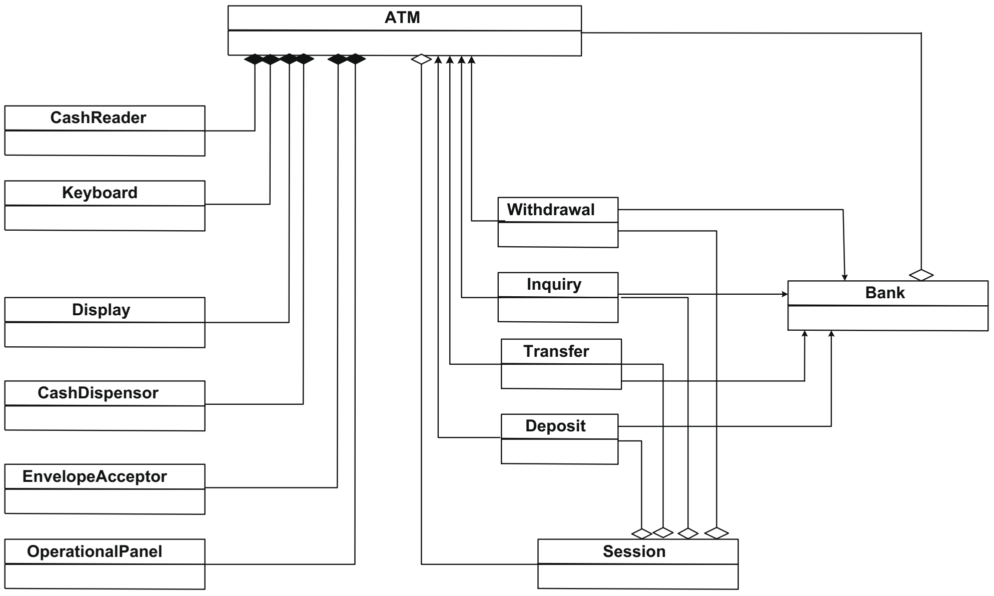


Fig. 15. Class diagram of the ATM application.

Table 1
The various NFRs and their tactics for this particular ATM.

NFR	Tactics for fulfilling the NFR	Classification
	Exception for checking cash availability	Addition of operations/attributes
Fault tolerance	Exception for checking cash limit	Addition of operations/attributes
	Exception for checking account balance	Addition of operations/attributes
Security/Confidentiality	Encryption using AES	Using new algorithm/ADT
	Authentication	Addition of operations/attributes
Usability	GUI	Addition of new classes
Maintainability	Reducing coupling	Modifying existing behavioral elements
Performance	Fast path	Introduction of new behavioral elements

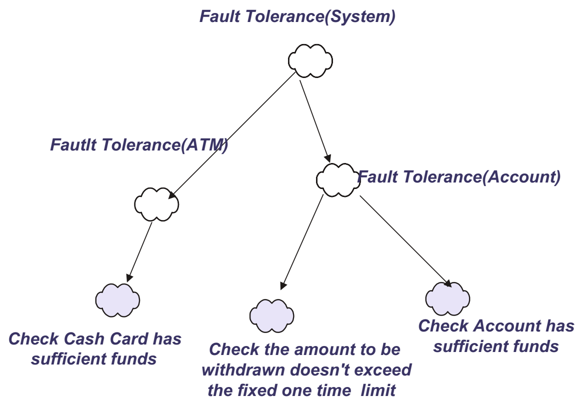
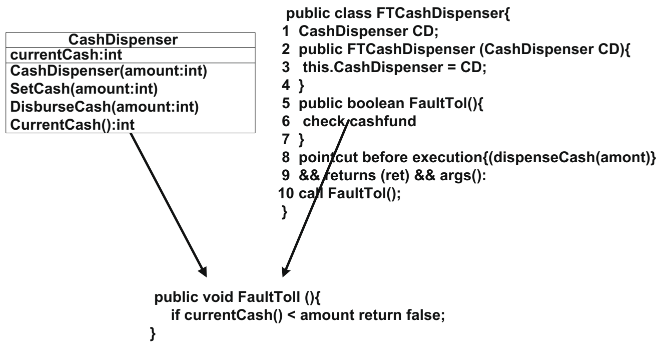

Fault Tolerance(System)
Fautlt Tolerance(ATM)
Fault Tolerance(Account)
Check Cash Card has sufficient funds
Check the amount to be withdrawn doesn't exceed the fixed one time limit
Check Account has sufficient funds

Fig. 16. SIG for fault tolerance.

those classes are involved in highly likely will be modified. The steps involved in fulfilling this tactic include

- *Step 1: Select the part of the system that is mainly affected by the tactic*: Applying the metrics "Coupling Between Objects (CBO)" metrics (Chidamber and Kemerer, 1994), we find that the classes ATM, Session, and Bank seems to have very high coupling. Since all of them participate in the use cases withdrawal, inquiry, balance, and deposit, these behavioral elements are affected.



```
public class FTCashDispenser{
 CashDispenser CD;
 public FTCashDispenser (CashDispenser CD){
  this.CashDispenser = CD;
 }
 public boolean FaultTol(){
  check cashfund
 }
 pointcut before execution{(dispenseCash(amont)}
 && returns (ret) && args():
 call FaultTol();
}
```

```
public void FaultToll (){
   if currentCash() < amount return false;
}
```

Fig. 17. Integration of CashDispenser and FTCashDispenser.

- *Step 2: Choose and analyze critical design element*: The three classes identified to have high CBO – ATM, Session, and Bank. The solution to reduce these classes' high coupling rests on the observation that the four classes withdrawal, inquiry, balance, and deposit have a similar behavior towards these three classes. Based on the above observation, we created a new class "Transaction" as a superclass of the four classes reducing the coupling of the system.
- *Step 3: Calculate redesign costs*. The introduction of "Transaction" does not require any modification on the rest of the system. The inheritance relationship between the four classes and Transaction allows us to maintain the existing behavioral elements (sequence diagrams).

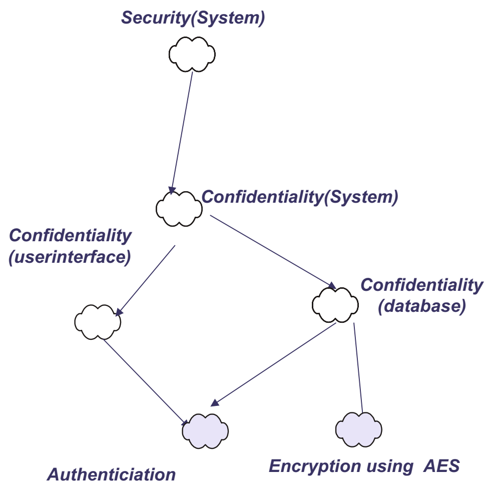


Fig. 18. SIG for security.

- *Step 4: Negotiate*. Even if the introduction of a new class would add a little overhead in performance, such increment is negligible.

The *Fast Path* tactic (Smith and Williams, 2002) for realizing performance is classified in "Introducing new behavioral elements" category. This tactic suggests to improve performance by introducing a new use case/subsystem for the most frequently used scenarios. In our ATM example, fast path is going to be applied in the withdrawal scenario (such decision was made during the development of the SIG for performance). What we are required to do is to introduce a new button in the display of the ATM interface which will allow the user to request a predefined (based on bank's policy, user's behavior, etc.) cash without actually entering a new value. The steps in fulfilling such a tactic grouped in "Introducing new behavioral elements" are

- *Step 1: Select the part of the system that is mainly affected by the tactic*: As decided by the SIG, the withdrawal sequence diagram is the part of the system affected by this tactic.
- *Step 2: Choose and analyze critical design element*: The entire sequence diagram of withdrawal is critical since the whole part is responsible for the fulfillment of the tactic. A new conditional path is added to the existing withdrawal sequence diagram as depicted in Fig. 19.
- *Step 3: Negotiate*. No other NFR is negatively affected by this tactic.

During the analysis of the effect of a tactic on another NFR, we found out that encryption has a negative impact on performance. This prompted us to carry out tradeoff analysis for which prioritization of the various NFRs is the first step. Applying AHP, we have the various weights of the NFRs as shown in Table 2.

Next, we analyzed the Q-SIG of Fig. 20. The Q-SIG only shows the part of the system we are interested in (security and performance). The 0.42 and 0.19 weights for security and performance are the result of applying AHP and the other values are assigned by expert like the negative impact of encryption on performance in the scale of 0 to 1 is assigned 0.3. Since the NFRs value of the system with encryption (its NCF), (0.55), is greater than the NFRs value of the system without encryption (0.47), we decided to keep the encryption. To increase our confidence in our decision, we carried out a sensitivity analysis using Eq. (6) of Section 5.3 and found out it to be around 81%.

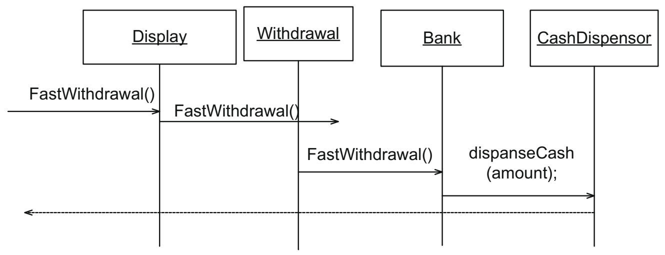


Fig. 19. Newly added sequence diagram for fast path.

Table 2
The relative weights of the various NFRs in the ATM application.

FT	Security	Usability	Maintainability	Performance
0.27	0.42	0.04	0.08	0.19

7. Discussion

Our case study has highlighted the strengths as well as the areas for further improvement of our proposed approach. Our ability to treat each tactic using a process that is suitable for specific groups but not all tactics can be seen as a good point of our research. Unlike other research, we have used ideas from the aspect oriented community to implement some tactics but we also haven't insisted to use aspects to implement all tactics. In the implementing of tactics through classpects, identification of classes that are advised by classpects is still not fully resolved. Our approach uses steps that are as intuitive as processes in other approaches and thus is not more difficult to implement or less useful. However, there are still difficulties in implementing tactics in groups like "Modifying Existing Behavioral Elements" which is common among all existing approaches. We believe, a further refinement of the tactic classification will help us improve the effectiveness of our approach.

Our case study also has demonstrated how useful our tradeoff analysis is in choosing between design alternatives. Our usage of both the expert opinion and the user's preference has increased our confidence in our decision. Still, even though we have managed to control the human error that can occur during prioritization by using AHP and sensitivity analysis, the values experts assign as a tactic's contribution to an NFR are still not subject to scrutiny. Fortunately, the effect of such values is diminished because of the whole tradeoff analysis that takes into account more than the expert opinion to make the final decision.

8. Related work

In this section, we compare various research works that focus on whole or part of the issues our work touches on.

8.1. NFR-themed approaches

Here, we discuss six research works that deal significantly with NFRs. Briefly these works are

- *Non-Functional Requirement (NFR) framework* (Mylopoulos et al., 1992; Chung and Nixon, 1995; Chung et al., 2000): Even though very useful in capturing NFRs at early requirement gathering phase, since the NFR-Framework is qualitative, it is not easy to describe the various shades of "satisfaction" except the few that are prescribed in the Framework.

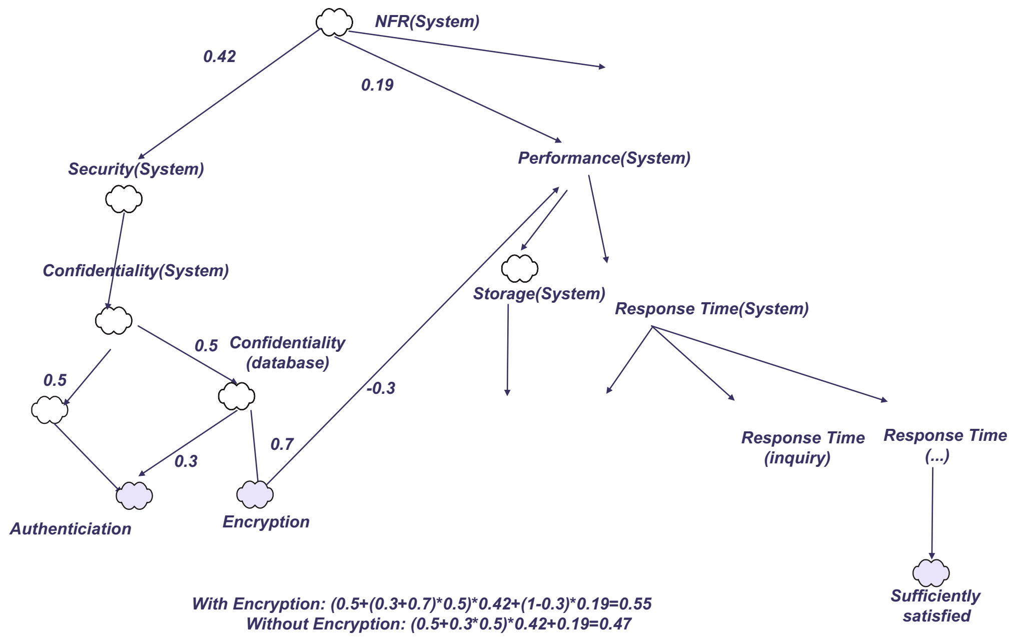


Fig. 20. Tradeoff analysis between security (encryption) and performance.

- *Arch. Pattern for Dependability Systems*: Xu et al. Xu et al. (2006) address the research question of transforming dependability requirements into corresponding software architecture constructs, by proposing three types of components. Among the three architectural components, the checkable components verifies whether a NFR is fulfilled instead of actually realizing the NFR. This group includes the majority of NFRs and thus it is difficult to use this work to actually implement those kinds of NFRs. Moreover, the work does not treat conflict resolution among NFRs.
- *Architecture Tradeoff Analysis Method (ATAM)*: ATAM (Clements et al., 2002) is a structured technique for understanding the tradeoffs inherent in the architectures of software intensive systems. The ATAM is good for high-level evaluation of an architecture but looses its usefulness when developers want a more fine-grained analysis especially when trying to integrate with an approach for functional modelling.
- *Non-functional Requirements: From Elicitation to Conceptual Models*: Cysneiros and Leite (2004) present a process to elicit NFRs, analyze their interdependencies, and trace them to functional conceptual models. They focus their attention on conceptual models expressed using Unified Modeling Language (UML). They show how to integrate NFRs into the Class, Sequence, and Collaboration Diagrams. Since it is based on the NFR-Framework, it suffers from some of the shortcomings from the Framework, e.g. lack of quantitativeness.

Table 3
Comparison table of different NFRs realization approaches.

Criteria		NFR Framework	Arch. pattern for dependability systems	ADD	NFR from elicitation to impl.	AOSD	ATAM	Our Approach
NFRs Management	Decomposition	*n*-Layer	*n*-Layer	None	*n*-Layer	None	2-Layer	*n*-Layer
	Classification	None	2-category	None	None	None	None	currently six
Targeted NFRs		All	Some NFRs (that introduce new functions)	Some NFRs (that require arch. transformation)	Most NFRs (that lead to new functions or constraints)	Some NFRs (that could be modeled as "cross-cutting")	Some NFRs (that require organizational change)	All
Development phase		Early analysis	Early analysis	Arch. design	Elicitation, analysis, and design	Impl.	Arch. design	Analysis and design
	FRs artifacts as input	Not explicitly	Class diagrams	Only high level arch.	Class, sequence, and collaboration diagrams	Class diagrams	High level arch.	Class, sequence, and deployment
Ease of integration with FRs models	Same language?	No, because no explicit FRs representation	Yes, UML	No, because no FRs representation	Yes, UML	No, because no NFRs representation	No, because no FRs representation	Yes, UML
	Traceability	Not easy (just intuitively)	Not easy (no explicit approach)	Difficult because the FRs are not considered	Easy because they use the same vocabulary	Difficult because no NFRs representation	Difficult because no FRs representation	Easy, because there is an approach for traceability
Prioritization		Not considered	Not considered	Not considered	Not considered	Not considered	Considered, local	Considered, global
Relationship between NFRs		Qualitative	Not considered	Only through their effect on arch.	Qualitative	Not considered	Quantitative, local	Quantitative and qualitative

- *Aspect Oriented Software Development (AOSD)*: AOSD (http://www.aosd.net/) is an approach to software development that addresses limitations inherent in other approaches, including object-oriented programming. AOSD is often considered as a good candidate in realizing NFRs because NFRs are often crosscutting concerns which are the focus of AOSD. Still, there are a number of NFRs that are not easily handled by aspects and thus AOSD fails to capture such NFRs.
- Designing software architectures to achieve quality attribute requirements (ADD): In Bachmann et al. (2005) the authors describe a structure called a "reasoning framework" as a modularization of quality attribute knowledge for automatically transforming an arch. spec by while meeting a certain quality requirement. The major shortcomings of ADD are the unnecessary strict restriction during specifying an NFR which greatly limits the application of ADD and the unavailability of the inner workings of the tool that ADD uses for demonstration purposes.

8.1.1. Criteria of comparison

So far we have discussed six research works that are concerned with various aspects of NFRs. In this section we provide six criteria for comparing these works with ours. Table 3 provides the analysis of the approaches based on these criteria. The criteria are:

- NFRs Management: The technique the approaches use to manage NFRs: decomposition (for the NFRs themselves) and classification (for the tactics used to fulfill those NFRs).
- Development Phase: The phase at which the approaches can be applied.
- Targeted NFRs: The class of NFRs for whom the approach is highly suitable for.
- Ease of integration with FRs models: The disparity between the FR and NFR model when the approach is used. We have three sub criteria for investigating how easy is to integrate the NFRs approaches with FRs ones (assumed to be objected-oriented): FRs artifacts as inputs (does the approach use some of the artifacts from FRs approaches as an input), the same language(does both approaches use the same representation?) and traceability (how easy is to trace from the NFRs approach to an FRs approach).
- Consideration of prioritization: The strategy used by the approaches to address prioritization among NFRs. Most approaches based on the NFR-Framework consider "criticality" which is different from prioritization. In our opinion, a "critical" requirement implies a system must realize that requirement no matter what where as a prioritization is used to give the developer the guideline on how much important for the user each requirement is.
- Consideration of relationship among NFRs: How do the various approaches depict and analyze NFR interdependencies.

8.1.2. Result of comparison

The following table summarizes the analysis of the NFR-themed approaches according to the comparison criteria.

8.2. Research Works that focus on prioritization or tradeoff analysis of requirements including NFRs

Tradeoff analysis in general or prioritization and interdependency analysis of requirements in particular have been the focus of different researchers. In this section, we discuss those that are closely related to our approach. These are:

- *AGORA Attributed Goal-Oriented Requirements Analysis Method*: Kaiya et al. (2002), presented a graph based, quantitative approach for augmenting the goal-oriented requirement analysis (GORA) methods such as I* (http://www.cs.toronto.edu/km/istar/). The work helps developers assigns numerical values for the nodes to capture positive/negative contribution among each other as well as the preference for each goal. The primary difference between their approach and ours is that in AGORA, preference refers to the degree one stakeholder thinks another stakeholder prefers a particular node which is different from prioritization in our work that tries to quantify how much a given stakeholder prefers a node compared to other nodes.
- *Process-Oriented Metrics for Software Architecture Evolvability*: Subramanian and Chung (2003) added a quantitative dimension to the SIG of the NFR-Framework. Even though such addition is a welcome step in the right direction, it falls short of what is achieved through our work. First, the quantification in (Subramanian and Chung, 2003) is assigning numerical values for the various discreet values softgoals and links can have in SIG. Moreover, "criticality" not "prioritization" is addressed and thus developers cannot compare softgoals and show their preference of each NFR relative to the others.
- *Quantitative Model for the Evaluation of Software Architectures*: Zayaraz and Thambidurai (2007) presented an architecture evaluation model that focuses on the prioritization among NFRs. Their model is detailed in its consideration of the subgroups of NFRs and normalization of prioritization. Still, their approach only concentrates on the entire architecture instead of specific design alternatives. Moreover, they don't consider interdependencies among NFRs.
- *Analysis of Conflicts among Non-Functional Requirements Using Integrated Analysis of Functional and Non-Functional Requirements*: Sadana and Liu (2007) present an approach not only considers conflicts among NFRs as well as NFRs on FRs. That is an important contribution as we have pointed out in 5.4 the impact of NFRs on the FRs should be considered for the full evaluation of NFRs. However, V. Sadana et al.'s work targets very high level requirements, is highly qualitative, and assumes NFRs have conflict with each other instead of their tactics having conflict with NFRs.

9. Conclusion and future work

We have leveraged existing work to better tackle the problem of incorporating NFRs in object-oriented analysis and design (OOAD). We believe our work has three major contributions. First, it suggests a tactics types classification framework for management of NFRs. We developed the framework so that tactics that are realized at the same type in OOAD are grouped under the same category. We argued why it is better to focus on the tactics of NFRs than the NFRs themselves for analyzing and designing NFRs. Second, it proposes an approach we can use for incorporating NFRs into software analysis and design. The approach provides various processes suitable for realizing tactics in each tactic group. We Third, it suggests a better approach to manage tradeoff among competing NFRs by considering prioritization and the positive/negative impact of tactics on NFRs. To address the relationship between NFRs qualitatively, quantitatively, and globally, we enhanced SIG by including quantitative information. We also suggested FR-QSIG to address the effect of tactics that require the system to be modified. We showed how such analysis can help us to decide between tactics that realize the same NFR.

In future, we are planning on conducting industry scale case studies. We hope such case studies will provide us with information on how practitioners use our approach and a better validation on the various quantitative measurement we proposed. Besides, we expect to get feedback on how to improve the approach. Another future work we are considering is to incorporate our approach into existing object-oriented development tools so as to ease the burden on the developer when he/she is trying to capture and implement NFRs.

References

Advanced Encryption Standard, Federal Information Processing Standards Publication, 197, 2001.

AspectJ: <http://eclipse.org/aspectj>.

Bachmann, F., Bass, L., Klein, M., Shelton, C., 2004. Experience in using an expert system in designing for modifiability. In: Proceedings of the WICSA, IEEE Press.

Bachmann, F., Bass, L., Klein, M., Shelton, C., 2005. Designing software architectures to achieve quality attribute requirements. IEE Proceedings – Software 152 (4), 153–165.

Booch, G., Rumbaugh, J., Jacobson, I., 1999. The Unified Modeling Language User Guide. Addison-Wesley.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object-oriented design. IEEE Transactions on Software Engineering 20 (6), 476–493.

Chung, L., Nixon, B., 1995. Dealing with Nonfunctional Requirements: Three Experimental Studies of a Process-Oriented Approach. In: Proceedings of the 17th International Conference on Software Engineering, pp. 24–28.

Chung, L., Nixon, B., Yu, E., Mylopoulos, J., 2000. Non-Functional Requirements in Software Engineering. Kluwer Academic.

Clements, P., Kazman, R., Klein, M., 2002. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley.

Cysneiros, L., Leite, J., 2004. Nonfunctional Requirements: From Elicitation to Conceptual Models. IEEE Transactions on Software Engineering 30 (5), 328–350.

Dobrica, L., Niemela, E., 2002. A survey on software architecture analysis methods. IEEE Transactions on Software Engineering 28 (7).

Ghezi, C., Jazayeri, M., Mandrioli, D., 1991. Fundamentals of Software Engineering. Prentice-Hall International Edition.

HSue Black, 2001. Computing ripple effect for software maintenance. Journal of Software Maintenance and Evolution, Research and Practice.

ISO/IEC 9126-1 (2001). Software engineering. Product quality. Part 1: Quality model. International Organization for Standardization.

Jacobson, I., Booch, G., Rumbaugh, J., 1999. The Unified Software Development Process. Addison-Wesley.

Kaiya, H., Horai, H., Saeki, M., 2002. AGORA: attributed goal-oriented requirements analysis method. In: IEEE Joint International Conference on Requirement Engineering, RE02.

Karlson, J., Ryan, K., 1997. A cost-value approach for prioritizing requirements. IEEE Software 14 (15), 67–74.

Marew, T., Bae, D., 2006. Using classpects for integrating non-functional and functional requirements. In: ASTED Conference on Software Engineering, pp. 141–147.

Mead, N. <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements/545.html>.

Mylopoulos, J., Chung, L., Nixon, B., 1992. Representing and Using NonFunctional Requirements: A Process-Oriented Approach. IEEE Transactions on Software Engineering 14 (6).

Nilsson, N., 1971. Problem Solving Methods in Artificial Intelligence. McGraw-Hill, New York.

Rajan, H., Sulllivan, K., 2005. Classpects: unifying aspect- and object-oriented language design. In: Proceedings of the 27th International Conference Software Engineering, pp. 59–68.

Saaty, T., Vargas, L., 1991. Prediction, Projection, and Forecasting. Kluwer Academic Publisher.

Sadana, V., Liu, X., 2007. Analysis of conflicts among non-functional requirements using integrated analysis of functional and non-functional requirements. Computer Software and Applications Conference, COMPSAC.

Smith, C.U., Williams, L.G., 2002. Performance solutions: a practical guide to creating responsive, scalable software. Addison-Wesley.

Spencer, I., Bittner, K., 2006. Managing Iterative Software Development Projects. Addison-Wesley Object Technology.

Subramanian, N., Chung, L., 2003. Process-oriented metrics for software architecture evoluability, IWPSE.

Triantaphyllou, E., Mann, S.H., 1994. Some critical issues in making decisions with pair-wise comparisons. In: Proceedings of the Third International Symposium on the AHP, pp. 225–236.

Xu, L., Ziv, H., Alspaugh, T., Richardson, D., 2006. An architectural pattern for non-functional dependability requirements. Journal of Systems and Software, 1370–1378.

Yau, S., Collofello, J., 1985. Design stability measures for software maintenance. IEEE Transactions on Software Engineering 11 (9).

Yau, S., Collofello, J., McGregor, T., 1978. Ripple effect analysis of software maintenance. In: Proceedings Computer Software and Applications Conference (COMPSAC).

Zayaraz, G., Thambidurai, P., 2007. Quantitative model for the evaluation of software architectures, Software Quality Professional.

Zhu, L., Aurum, A., Gortorn, I., Jeffery, R., 2005. Tradeoff and sensitivity analysis in software architecture evaluation using analytic hierarchy process. Software Quality Journal 13, 357–375.

Tegegne Marew received his B.Sc. in Computer Science from Addis Ababa University, Ethiopia in 1999, and his M.S. in Computer Science from Korea Advanced Institute of Science and Technology in 2003. He is currently pursuing his Ph.D. in Computer Science with a software engineering focus in KAIST. His research interests include software engineering, software requirement engineering, software architecture design and analysis, object-oriented analysis and design, aspect-oriented programming, and software metrics.

Joon-Sang Lee received a B.S. in Computer Engineering from Dongguk University, Seoul in 1997, his M.S. in Computer Science from Korea Advanced Institute of Science and Technology in 1999 and his Ph.D. in Computer Science from Korea Advanced Institute of Science and Technology in 2003. He has served 4 years in LG Electronics as a software architect, 1 year in Korea University as a research professor, and has been leading the quality engineering part of Information Technology Laboratory in LG Electronics Advanced Research Institute since 2008. His research interests include software engineering, software architecture design and analysis, object-oriented programming, aspect-oriented programming, software quality, and software verification.

Doo-Hwan Bae graduated from the Seoul National University with a B.S. and a M.S. degrees in engineering in 1980 and 1982, respectively. He received a M.S. degree in Computer Science from the University of Wisconsin-Milwaukee in 1987, and a Ph.D. degree in Computer and Information Sciences from the University of Florida in 1992. Since then, he joined the faculty of Computer and Information Sciences department at the University of Florida as a visiting professor. In 1995, he moved to KAIST and currently is an associate professor at the Computer Science division in the department of Electrical Engineering and Computer Science. Since 2002, he is directing the ITRC Software Process Improvement Center, sponsored by Korean Ministry of Information and Communication. His research interests are in Software Engineering, especially in the areas of Object-Oriented Technology, Component-Based Software Engineering, and Software Development Process and its Improvement.