From Non-Functional Requirements to Design through Patterns

Daniel Gross and Eric Yu
Faculty of Information Studies, University of Toronto, Toronto, Ontario, Canada

Design patterns aid in documenting and communicating proven design solutions to recurring problems. They describe not only how to solve design problems, but why a solution is chosen over others and what tradeoffs are made. Non-functional requirements (NFRs) are pervasive in descriptions of design patterns. They play a crucial role in understanding the problem being addressed, the tradeoffs discussed, and the design solution proposed. However, since design patterns are mostly expressed as informal text, the structure of the design reasoning is not systematically organized. This paper proposes a systematic treatment of NFRs in descriptions of patterns and when applying patterns during design. The approach organizes, analyzes and refines non-functional requirements, and provides guidance and reasoning support when applying patterns during the design of a software system. Three design patterns taken from the literature are used to illustrate this approach.

1. Introduction

Requirements Engineering is now widely recognized as a crucial part of software engineering, and has established itself as a distinct research area. Equally important is how requirements drive the rest of software development. In particular, during the design phase, much of the quality aspects of a system are determined. Systems qualities are often expressed as non-functional requirements, also called quality attributes e.g. [1,2]. These are requirements such as reliability, usability, maintainability, cost, development time, and are crucial for system success. Yet they are difficult to deal with since they are hard to quantify, and often interact in competing, or synergistic ways. During design such quality requirements appear in design tradeoffs when designers need to decide upon particular structural or behavioral aspects of the system. A good designer is one who can do this well, and who has learned how to address a range of the quality requirements properly from experience.

The NFR framework [3,4,5] took a significant step in making the relationships between quality requirements and design decisions explicit. The framework uses non-functional requirements to drive design [6] to support architectural design [7,8] and to deal with change [9].

In the software design area, the concept of design patterns has been receiving considerable attention [10,11,12,13]. The basic idea is to offer a body of empirical design information [12] that has proven itself and that can be used during new design efforts. In order to aid in communicating design information, design patterns focus on descriptions that communicate the reasons for design decisions, not just the results. It includes descriptions of not only “what”, but also “why” [13]. Given the attractiveness and popularity of the patterns approach, a natural question for RE is: How can requirements guide a patterns-based approach to design?

This paper argues that a systematic approach to organizing, analyzing, and refining non-functional requirements can provide much support for the structuring, understanding, and applying of design patterns during design. NFRs that are explicitly represented in design patterns aid in better understanding the rationales of design, and make patterns more amenable to structuring and analysis. We use examples from the design pattern literature to illustrate how NFRs can be represented and used to guide the application of design patterns during the design process.

The next section introduces design patterns, and uses the well known “Observer” pattern [10] as a typical example from the literature to motivate our approach. Section 3 enumerates the objectives and the main features of our approach. Section 4 illustrates the approach using the Observer pattern. Section 5 uses two patterns from a more complex real-life example (the design of a distributed alarm system) to further demonstrate the feasibility of the approach. Section 6 discusses the approach. Section 7 points to related work. Finally, section 8 concludes and points to future work.

2. What are design patterns

Christopher Alexander, the renowned (building) architect widely acknowledged to be the originator of the pattern idea, explains that “each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution … as an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves” [14]. Jim Coplien, a noted researcher and practitioner in the software patterns community, explains that a pattern is a “piece of literature that describes a design problem and a general solution for the problem in a particular context”, and elaborates that "if we understand the forces in a pattern, then we understand the problem (because we understand the trade-offs) and the solution (because we know how it balances the forces), and we can intuit much of the rationale. For this reason, forces are the focus of a pattern" [12].

Patterns are primarily textual based descriptions. Graphical descriptions such as pictures, object diagrams and the like, often accompany the text, usually to represent the desired solution structures that are suggested by the pattern. Some approaches use informal means of representing solution structures [14] while others suggest more formal representations to facilitate analysis & tool support when applying pattern solutions during systems design [15,16,17].

Patterns are written using predefined templates or forms. Several pattern forms exist in the literature, differing by the kind of categories they emphasize. For example, there is the Alexandrian form [18], the GOF (Gang of Four) form [10], and the Coplien form [19]. Coplien and Schmidt [20] provide additional examples. All forms contain the basic categories: name, problem statement, context, description of forces, solution and related patterns. Sometimes pattern forms do not use all categories explicitly, but ask them to be discussed during the pattern presentation.

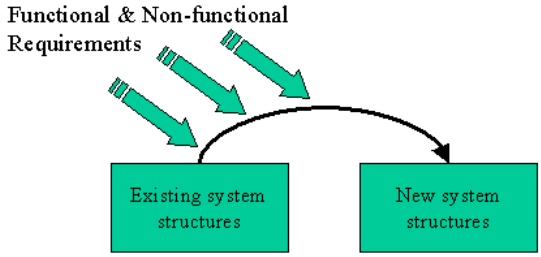

Fig. 1: Transforming solution structures when applying patterns during design

Applying a design pattern may be understood as transforming the system from one stage of development to the next (Figure 1). The system requirements together with the design structures established so far during development define the context and the set of forces that characterize the problem. The context is the “current design state” of the system under development. The solution section of the design pattern then describes what new structures to incorporate into the current

design state in order to resolve the forces in a favorable manner. After applying a design pattern, new system structures are established ("new design state"), which in turn become the context for a new design problem and forces to be resolved, and for which other design patterns may be provided.

Consider the "Observer" pattern as a typical example from the pattern literature [10] The Observer pattern is described using the GOF form, which includes, among others, an *Intent* section, a *Motivation* section, a *Structure* section, and a *Related Pattern* section. The *Intent* of the pattern is to "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically". The *Motivation* section explains that "a common side-effect of partitioning a system into collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability". The same section provides an example to illustrate the patterns intent: "Many graphical user interface toolkits separate the presentational aspect of the user interface from the underlying application data. Classes defining application data and presentation can be reused independently… Both a spreadsheet object and a bar char object can depict information in the same application data object using different presentations". Finally, the section explains that the "key objects in this pattern are **subject** and **observer** … All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state. This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications … Any number of observers can subscribe to receive notifications" (bold face in the original). The *Structure* section provides an object diagram representing the pattern solution that deals with the requirements of consistency, loose coupling, reusability and extensibility. Finally, the *Related Pattern* section suggests that the Mediator and the Singleton pattern may be useful when applying the Observer pattern.

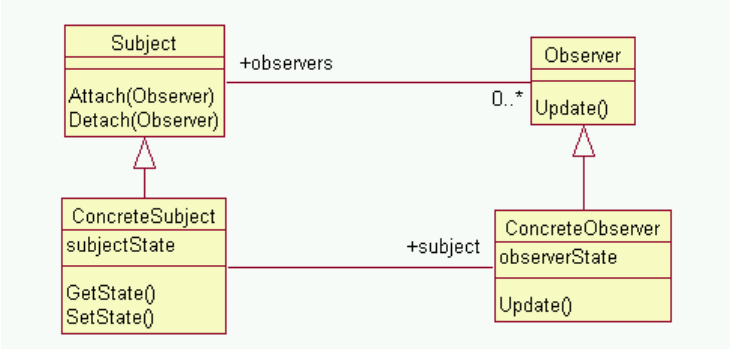

Fig. 2: Observer pattern solution structure in UML[1] (adopted from [10])

The object diagram in figure 2 (from [10]) shows the solution structures proposed by the pattern. From this figure a designer attempting to learn or use the pattern cannot tell why these structures are being advocated. The intentions and rationales are implicit. The problems and forces that these objects and structures aim to address and resolve are presented discursively in the text. Such textual descriptions are, however, not easily amenable to analysis and tool support.

3. A Requirement driven approach to design patterns

We propose an approach that makes the reasoning structure behind a design pattern explicit, and amenable to systematic analysis. It focuses on the non-functional requirements that the pattern

[1] UML – Unified Modeling Language [21]

addresses, the tradeoffs that are involved, and how the requirements are met when the pattern is applied. The approach aims to:

- **Clarify the role of NFRs in design patterns.** NFRs discussed in design patterns are explicitly represented. This makes them "first class citizens" which enables referring and analyzing the role they play in the argumentation structure of the design patterns.

- **Provide structure for characterizing each pattern**. NFRs relate to each other and to the alternative solutions discussed in patterns. NFRs are the criteria for evaluating why to accept or reject a solution. Making such relationships explicit allows characterizing pattern solutions in terms of their NFR related properties.

- **Systematically support for the application of patterns during design**. NFRs need to be systematically addressed during design. Characterizing patterns with respect to NFRs allows for better addressing system wide NFRs when retrieved, selected, and then applied patterns to the current design situation.

- **Support forward engineering and traceability.** NFRs become focal points to be addressed by patterns. Designers are guided while addressing NFRs, when searching, retrieving and selecting patterns. Keeping track how NFRs drive the applying of patterns during design, allows to retain how the system structures evolved and thus facilitates NFR related tracability.

To achieve the above objectives, the approach adopts the following as its main features:

- **Represent requirements as design goals:** Functional and non-functional requirements are represented as goals to be achieved during design. Non-functional requirements are denoted by *NFR softgoals*. NFR softgoals are used to express the forces in design patterns, and the quality requirements to be achieved by the intended software system. We call them *softgoals* because they typically do not have clear-cut criteria of achievement. A qualitative style of reasoning is used during analysis. Functional goals represent the functions that the system is to achieve. They serve as focal points for analyzing alternative system functions and structures during the design process.

- **Show relationships among the goals in a graph:** NFR softgoals discussed in a pattern are arranged as a goal graph (NFR goal graph). NFR goals are connected to reflect their respective contribution to each other. The contributions are typed according to whether they are positive or negative and whether "partial" or "sufficient".

- **Identify implicit or intermediate goals:** While connecting goals through contribution links additional goals may be identified. They provide further clarification on what the NFRs mean and how they relate among each other in the pattern.

- **Show how known solutions achieve goals:** Solutions in a design pattern describe known design techniques that can be adopted during system design. Solutions are represented as *operationalizing softgoals*. They "operationalize" because they turn goals into solutions; they are still treated as goals because there can still be different way for achieving them. For each potential solution, the goal graph shows how, and how well, it achieves the stated NFR softgoals.

- **Identify unintended correlation effects among goals and solutions:** Correlation links describe non intended "side effects" solution structure have on various NFR softgoals. Through the use of contribution and correlation links synergistic and/or competing NFR softgoals may be specified.

- **Show how alternative solution structure contribute differently to goals:** Alternative solutions in a pattern (represented as operationalizing softgoals) contribute differently to the NFR softgoals in the goal graph. Each proposed solution can be analyzed in terms of its NFR softgoal achievement. Synergistic and/or competing NFR softgoals (and thus NFR requirements) can be identified.

- **Support argumentative style of modeling:** Claims for or against modeling choices are documented through argumentation softgoals. This is useful to record design assumptions. Such argumentation softgoals are treated as goals, since they in turn may still need to be justified or can be refuted. Argumentation softgoals may strengthen, weaken or completely refute the contributions softgoals may have on each other.

- **Reason qualitatively to establish the degree of goal achievement:** An interactive labeling procedure is used to judge how well solution structures achieve the requirements. During this procedure softgoals are labeled in accordance to how well they have been achieved so far by the proposed solution structures. These labels are then propagated upwards in the goal graph, while taking into account the different types of contribution and correlation links that relate goals in the goal graph.

- **Identifying type and topic in goals:** While representing a non-functional requirement as a softgoal, analyze it into a type and a topic. The NFR type denotes the quality desired (such as performance, optimal utilization), while the NFR topic denotes an existing or intended structure or behavior that should exhibit that desired quality (software processes, memory, data structures). The identification of topic allows specifying to what part or aspect of the intended system the quality requirements apply.

- **Elaborate the system under design:** System design is described in terms of the incremental elaboration of the functional structure. The elaboration is guided by the operationalizing softgoals described in the goal graph. Alternative operationalizing softgoals relate to alternative functional decompositions during system design. Functional goals are used as focal points for specifying such alternatives functional decompositions.

4. Illustrating a requirement driven approach to design patterns

This section illustrates how our approach is used to represent, analyze and then apply the Observer design pattern during design.

Consider typical design setting in which the problem discussed by the Observer pattern arises: A software designer may have in her current design several objects such as a spreadsheet a bar chart and a pie chart object. All objects manipulate and display their data that need to be kept consistent among each other. Figure 3 describes such a design situation.

Let us now demonstrate how an NFR goal graph is built when analyzing the pattern text, and refer to the first paragraph in the Motivation section of the pattern (italics added to emphasis relevant NFRs):

> A common side-effect of partitioning a system into a collection of cooperating classes is the need to *maintain consistency* between related objects. You don't want to achieve consistency by making the classes *tightly coupled*, because that reduces their *reusability*.

From the above paragraph we define the NFR softgoal consistency[object_data]. The type of the NFR softgoal is the required property, *consistency*. The topic of the NFR softgoal is *object data*. Together they express the requirement (i.e. design goal) of achieving consistency among object data. Further requirements are that objects should be reusable - reusability [objects] - and loosely coupling – loose_coupling [objects]. This latter design goal is inferred from the wish to avoid tight coupling among objects. The paragraph argues that tight coupling would reduce reusability. From this we infer that loose_coupling[objects] contributes positively to reusability[objects]. We show this by creating a contribution link of type "Help" from loose_coupling[objects] to reusability [objects], which denotes a positive contribution but not sufficient by itself to achieve reusability[objects]. Other mechanisms such as standardized interfaces are needed to sufficiently achieve the reusability goal of objects. This can be shown by defining the standardized_interface [objects] NFR softgoal, and creating from it a contribution link of type "Help" to reusable[objects]. This additional NFR softgoal is not explicitly mentioned in the pattern text. It, however, is the justification for the abstract classes Subject and Observer in the Observers pattern solution in figure 2, that provide a standardized interface for the publish-subscribe mechanism among Observer and Subject objects.

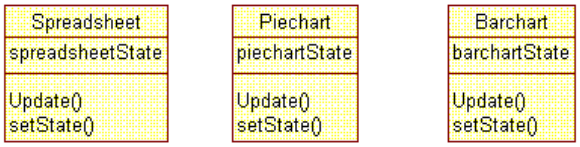

Fig. 3: Current design state: How to make those objects work together?

In order to clarify the relationship among the solution components of the observer pattern and the NFR softgoals identified so far, let us refer to a subsequent paragraph in the pattern text. After introducing Subject and Observer objects and describing how they collaborate, the pattern text further explains that (bold text in the original, italics by us):

> This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. *Any number of observers* can subscribe to receive notifications.

First we define the operationalizing softgoal observer_pattern[objects] to denote the solution structure applied to objects in the domain. We then define the operationalizing softgoals publish_subscribe[objects] and abstract_classes[objects] to denote the publish-subscribe and the abstract classes solution components respectively, and make them subgoals of observer_pattern[objects]. This is shown by defined two contribution links of type "and" to the observer_pattern[objects]. Having the two components of the observer pattern solution made explicit, we can describe their respective contribution to the NFR softgoals identified so far. The publish_subscribe design technique sufficiently achieves reducing the coupling among objects and maintaining consistency among object data. This is shown by creating contribution links of type "Make" from publish_subscribe[objects] to the reduced_coupling[objects] and consistency[object data]. Similarly, a contribution link from abstract_classes[objects] to standardized_interface[objects] of type "Make" shows that introducing abstract classes sufficiently achieves a standardized interface for objects.

The italicized phrase "*Any number of observers*" in the above diagram suggests that the ability to add further observers to the design, without the need to change the subject object is another NFR that the pattern addresses. We define the extensibility[system_functions] NFR softgoal to denote this requirement. Since both the publish subscribe mechanism and abstract classes together achieve sufficiently the extensibility of system functions, we define a contribution link of type "Make" from observer_pattern[objects] to extensibility[system_functions].

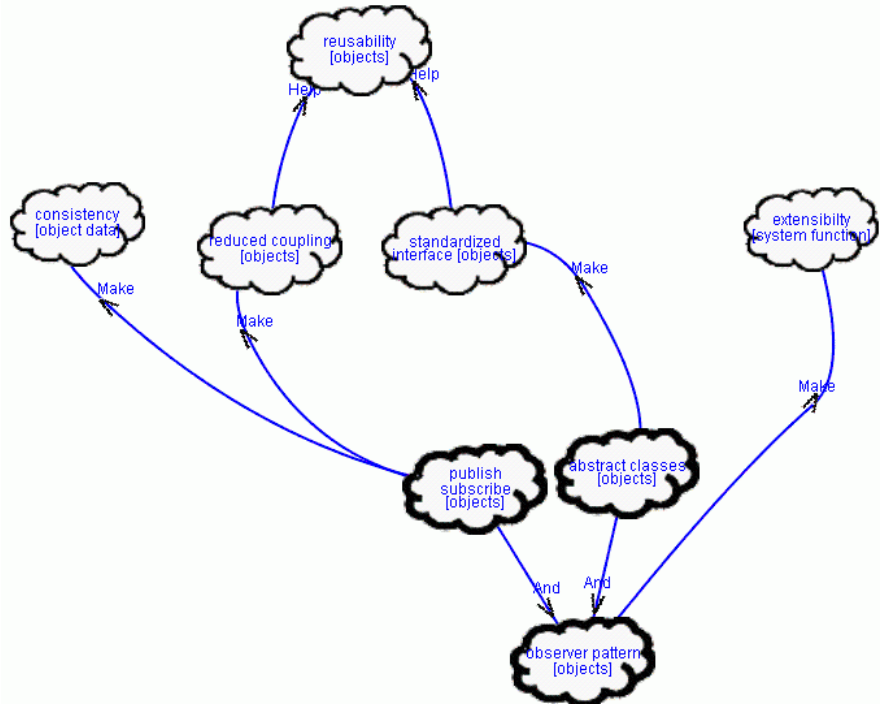

Fig. 4: Representing the properties of the Observer pattern solution structure as an NFR goal graph

Figure 4 shows the NFR goal graph we have defined. NFR softgoals are denoted by the light solid-line clouds. Operationalizing softgoals are denoted as clouds with thick solid borders. The links between them are the contribution links discussed. The contribution types defined in the notation are "Make" which means sufficiently achieves the parent goal, "Some+", there is some positive contribution of unknown extend towards the parent goal. The and refinement into two sub-softgoals means both sub-softgoals are needed to be achieved in order to achieve the parent goal. "Break", "Hurt" or "Some-" denote negative contributions. These contribution types are used to help propagate the achievement status of the goals through the network inter-relationships using an interactive qualitative reasoning process as described in [4]. Please refer to the appendix for a full legend of the notation.

Let us now turn to the tradeoffs discussed in the observer pattern. First we define the alternative solution rejected by the pattern, to have objects collaborate directly. This is denoted by the direct_collaboration[objects] operationalizing softgoal. From the pattern text we know that objects collaborating directly are tightly coupled. This is shown through a contribution link of type "Hurt" from direct_coupling[objects] to reduced_coupling [objects].

We can now analyze whether other NFR softgoals are impacted negatively through a tightly coupled solution structure. Since extensibility is also inhibited through tight coupling we create another contribution link of type "Hurt" between the direct_collaboration[objects] and the extensibility [system_functions].

Let us look for tradeoffs not discussed in the pattern text. We can, for example, ask what positive properties does a design have in which objects are directly collaborating. Two properties come to mind: having direct collaboration helps the performance of the system since less method invocations are performed – performance[system_processes] – and making software code less abstract makes it easier to understand – understandability[system]. The publish-subscribe mechanism, however, is hurting performance and makes the software code difficult to understand.

We can add such explanations to justify the choice of link types in the goal graph. This is done through argumentation softgoals that argues for or against the link type. Figure 5 illustrates the argumentation softgoal "abstract code is difficult to understand" which *supports* the contribution link of type "Hurt" from publish_subscribe[objects] to understandability [system]. This support is expressed through a contribution link of type "Make" from the argumentation softgoal to the contribution link.

Understandability of the software code has in turn positive impact on the maintainability of the system – maintainability [system], which is another relevant NFR softgoal we can now identify. Since performance and understandability are not primary concerns of the pattern but rather "side-effects" they are defined as correlation links. These appear as dotted line links in the NFR goal graph. Figure 5 summarizes the tradeoffs discussed.

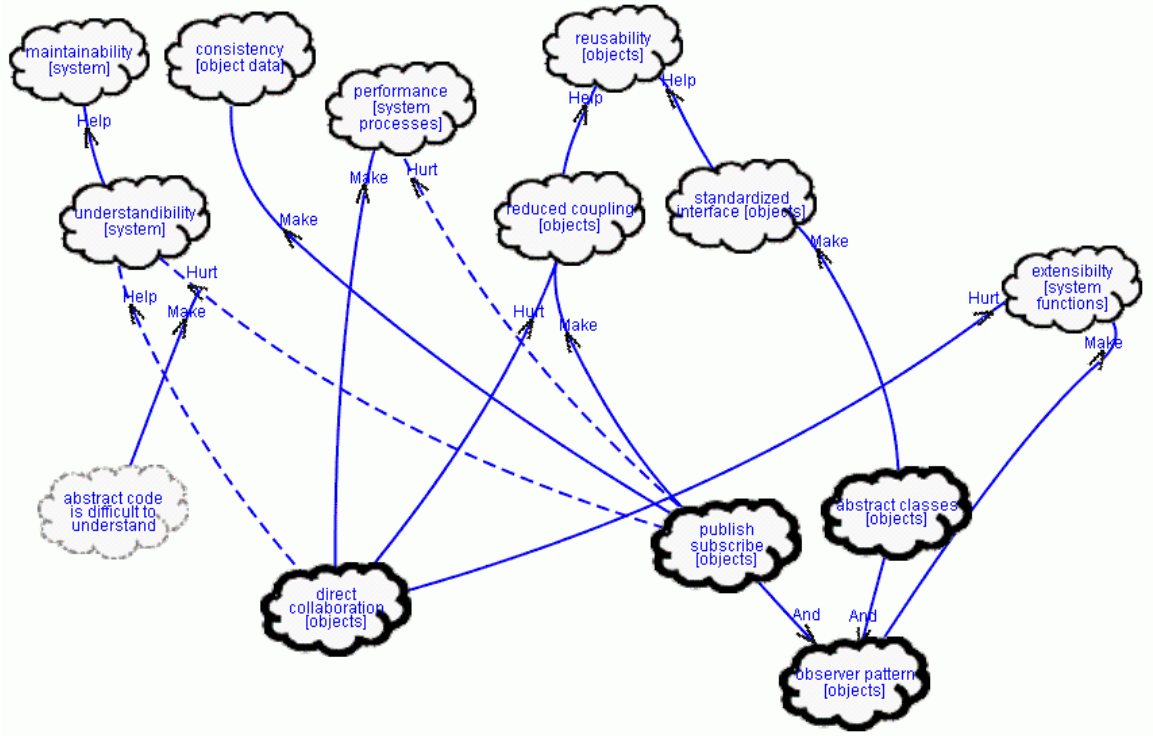

Fig. 5: Representing the reasoning structure of the Observer Pattern as an NFR goal

We now apply the Observer pattern during design. This is done by relating the operationalizing softgoals in NFR goal graph to a functional elaboration structure for the system under development.

Figure 6 shows on the left hand side a "collapsed" version of the observer pattern. Only the alternative operationalizing softgoals and several pertinent NFR softgoals are shown. On the right hand side we see a fragment of a functional elaboration of the software system under

development. The manage_views function is further elaborated into the functional goal view_objects_updated. A functional goal is used to express the possibility of alternative choices for elaborating functional structure, while a task represents a particular way of achieving a goal. The alternative tasks are linked to the goal through "means-ends" links, denoting the different means that can be used to achieve the goal. The means-ends reduction in the functional elaboration are the concrete manifestations of the corresponding operationalizing softgoals in the NFR goal graph. The means-ends links are related to the operationalizing goals through "design justification links". Design justification links with a "Make" contribution denote that the alternative task "implements" sufficiently the technique represented by the operationalizing goal, while justification link with an "and" contribution (not shown in figure 6) denotes that the alternative task partly implements the technique represented by the operationalizing goal. In order to fully implement that operationalizing softgoal, another "and" contribution from a task would be needed.

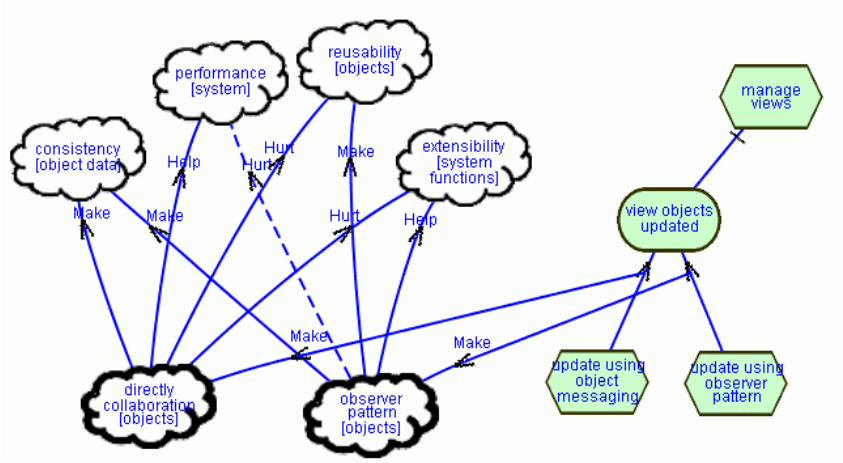

Fig. 6: Applying the Observer pattern during design

This functional goal is a focal point for two alternative functional elaboration, that corresponds to the two alternative solution structures discussed in the design pattern. Updating the view of an object can occur by having objects directly collaborate through messaging, or by using the Observer pattern. Each alternative is linked to the corresponding operationalizing softgoal through the "design justification" link. Operationalizing softgoals are in turn connected to the NFR softgoals relating alternative functional aspects of the system to operationalizing softgoals. This allows showing the tradeoffs, rationales and justifications that went into their design.

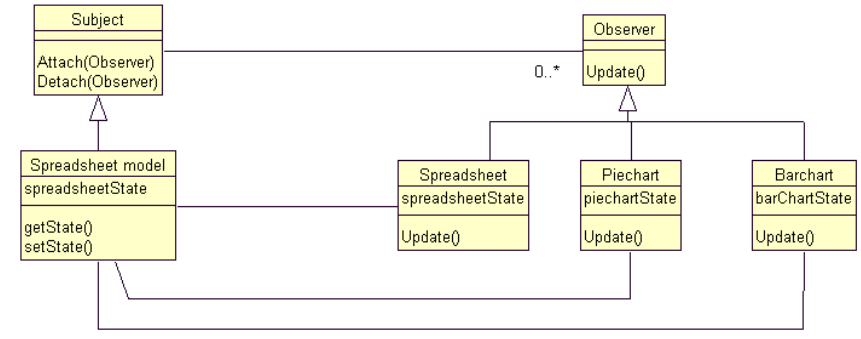

Fig. 7: Solution structure after applying the Observer pattern

Finally, the functional elaboration structure in figure 6 refers to the object oriented diagram shown in figure 7. The functional elaboration structure serves as a focal point for alternative functional elaboration, and justification of design in terms of Patterns and NFRs they address.

The object diagram represents one particular functional and structural design that corresponds to choosing one particular alternative branch in the functional elaboration structure. In this way many "justified" alternative object diagrams can be generated from a functional elaboration structure.

5. Example: the "TeleLarm AB" alarm system

5.1. Introduction

The previous section introduced our approach for representing design patterns and using them during design. In this section will take a more complex example to illustrate how to express the NFR related reasoning that a designer needs to do when applying several patterns during design.

The example patterns are taken from a real-life project for designing real-time alarm systems as reported by Molin & Ohlsson in [22]. The patterns reported address a variety of requirements and design issues relevant to that domain. These include, among others, requirements such as performance optimization and optimal utilization of limited memory, reliability (such as fault tolerance), maintainability, and portability.

We choose this examples because it reports a real-life effort in using patterns to document the requirements and architectural design features of a commercial line of products. Although the patterns are domain specific, their textual representation is a typical example how patterns are documented in the pattern literature.

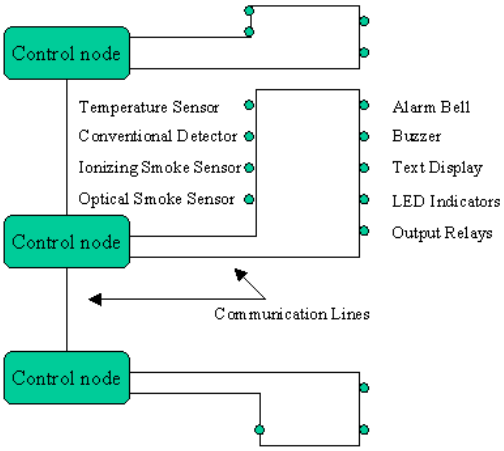

Fig 8. Overview of a fire alarm system (figure adapted from [22])

This section will present the Deviation and the Point pattern which are the first two (out of the six) design patterns discussed in [22]. Since we wish to demonstrate how patterns are successively applied during design, we will keep the presentation of the Deviation pattern short and elaborate more on representing and applying the Point pattern.

This section will first introduces the design context that gives rise to the problems the Deviation and the Point patterns come to address. It then gives a brief overview of the Deviation pattern, its structure, how it is applied during design, and the resulting design context. Then the Point pattern is illustrated, presenting its structure, how it is applied during design, and the resulting design context. Finally, we will illustrate what additional NFR related analyzes is needed when applying the Point pattern after the Deviation pattern during design.

Figure 8 shows how a fire alarm system can be described by its physical entities and their connections (adapted from [22]). The control nodes are autonomous processors. Each control node has a set of input sensors and/or output devices connected. Sensors and output devices are connected to control nodes through communication lines. Control nodes are also interconnected between themselves through communication lines.

Figure 9 shows an object diagram that describes the design of a control node in the above system. The diagram shows that an Alarm_Handler is responsible for surveying the environment. The Device_Poller object retrieves input status of all input devices connected to the control node, and stores all inputs as System_Status objects. An Alarm_List object keeps track of all the alarms that the locally attached output devices are responsible for. This is done by managing a mapping between the locally attached output devices and the subset of System_Status objects that are needed to check for the occurrence of an alarm. The Alarm_List object is also responsible for checking, on behalf of the Alarm_Handler object, the system status with respect to each defined alarm. Finally, the Alarm_List object writes to Output_Device objects in order to write to the locally attached output devices to either report the current system state or activate actuators in the environment to perform emergency actions. Note that the Device_Poller object and then Alarm_Handler objects are working concurrently within the control node.

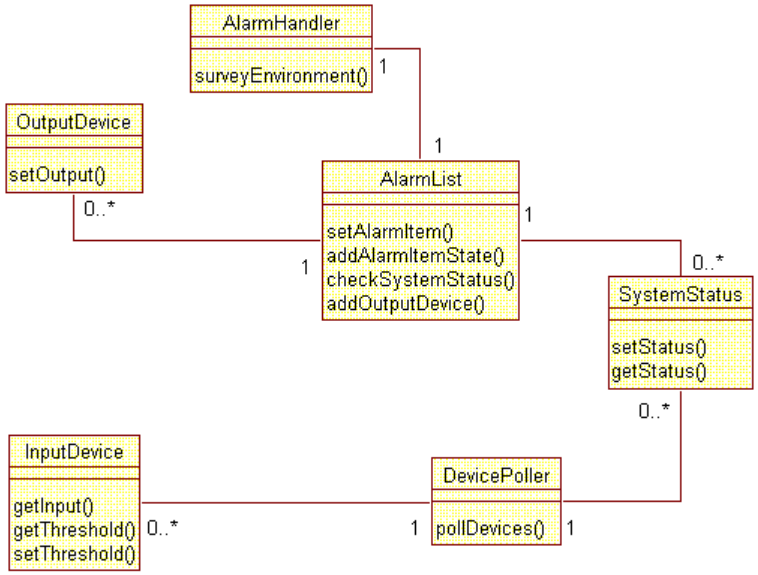

Fig. 9: Solution structure of one control node in the alarm system

There are two principal problems with the above design that arise when considering that we deal with a distributed processor environment, and that input and output devices exhibit great variability in the type of sensors, their detection algorithms, access protocols, and physical packaging. The following section will discuss how the Deviation pattern addresses the first problem. Section 6 will discuss how the Point pattern addresses the second problem.

5.2. Understanding, analyzing & applying the Deviation pattern

Dealing with a distributed environment means that an output device may be dependent on a set of input devices, no matter where the input devices are physically located in the distributed system. An input device might be connected to the same control unit as the output device, or it may be connected to another, remote, control unit. The "mapping" between outputs devices and system status items defined in the Alarm_List object (for all output devices locally available), may need to refer to System_Status objects that are locally available and to System_Status objects that

reside on other control nodes, and that need to be fetched from there. A key question is how to make remote system status locally available such that the performance of the system is not compromised during an alarm situation, when the Alarm_List object needs to traverse all subsets of System_ Status objects, while at the same time also keeping on eye on other required system qualities such as memory utilization and the complexity of design.

This question is addressed by the Deviation pattern. Figure 10 is an NFR goal graph that represents the argumentation structure of the Deviation pattern. The NFR goal graph was constructed in a "middle out" manner, similar to how we have demonstrated it with the Observer pattern. Some NFR softgoals and contribution links were explicitly mentioned as forces and relationships among forces in the pattern text, while others NFR softgoals and contribution links were identified while further analyzing the pattern text and clarifying its (often implicit) argumentation structure.

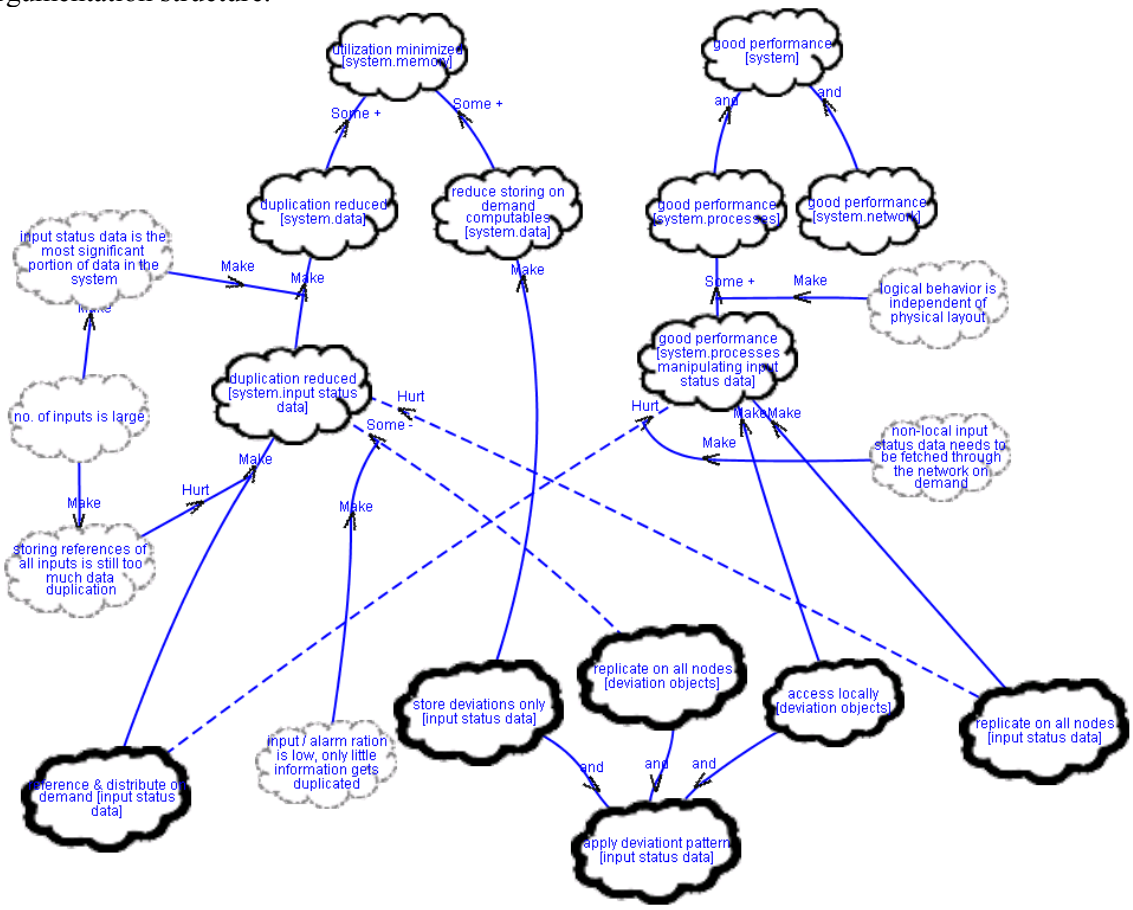

Fig. 10: The reasoning structure of the Deviation pattern

The top part of figure 10 shows that the pattern author has put up-front two major requirements during the design – memory utilization minimized in the system, and achieving good performance of the system. In pattern terminology these two top-level goals in the graph indicate that the basic question the designer faces is how to resolve the forces of minimizing memory utilization, while at the same time achieve good system performance.

At the bottom of figure 10 we can see the three alternative solution strategies that are discussed in the pattern. a) to store in each control node references to system status information that is stored in a remote control node (i.e., introduce a proxy SystemStatus object). b) to duplicate all system

status information to all control nodes in the system (i.e., introduce a replication mechanism for SystemStatus objects). c) to duplicate only status information that is deviant from its normal value to all control nodes in the system (introduce a replication mechanism but for deviant status information objects only). The components of the third solution are further elaborated, to clarify its contributions to pertinent NFRs in the goal graph. All three alternative solutions are denoted by the operationalizing softgoals at the bottom of figure 10.

The NFR softgoals between the top part and the bottom part of figure 10 represent the argumentation structure of the pattern. In principle the pattern argues that adopting the first solution, to reference remote system status data, would reduce data duplication and thus memory utilization needs in the system. However, performance of the system during alarm situations would not be optimal. Also, in this particular design context (i.e. in an embedded system), storing references for all remote input status data is too expensive in terms of memory utilization. This solution is rejected. The second solution duplicates all input status data to all control nodes, which would provide optimal performance during alarm situations. However, memory utilization would not be minimized. This solution is also rejected. The third solution is then adopted as the pattern solution, to duplicating only deviant objects. The pattern further argue in support for this solution in this particular design context, since the ratio of sensors vs. the number of sensors participating in an alarm is rather low. Therefore there is not a lot of deviant input status information that needs to be duplicated, which mitigates the memory utilization concern. Having all deviant input status data duplicated in all control nodes, makes then locally accessible to all the local processes that access them during an alarm, which optimizes their performance.

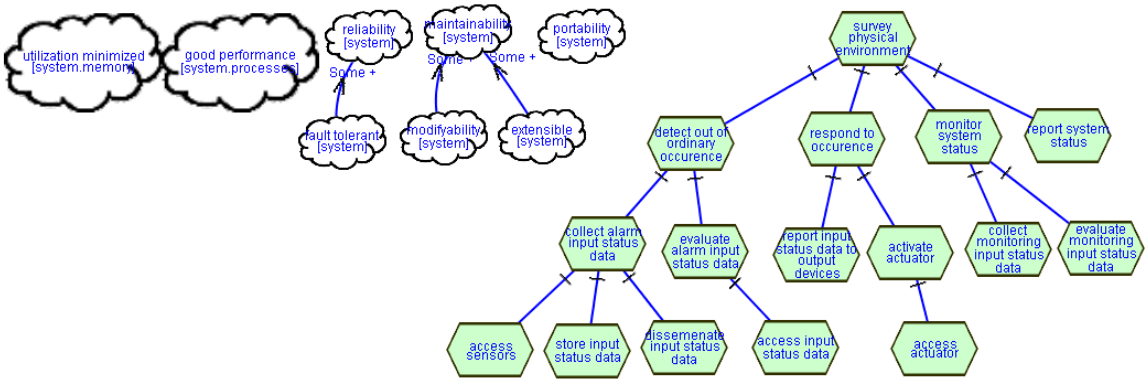

Fig. 11: Functional and non-functional aspects of the alarm system

Let us now turn to applying the pattern during design. Figure 11 shows the kind of requirements the intended alarm system should meet. It includes the non-functional requirements mentioned previously (by the clouds denoting softgoals on the left-hand side), together with a first elaboration of the functions the system should be capable of performing. Tasks on the right hand side refers to objects and methods defined in the object diagram. These include the ability to survey the physical environment by detecting out-of-ordinary occurrences, responding to those occurrences, monitoring and reporting facilities of the system status. Each one of these sub-functions is refined into more particular functions. During design the designers now asks: how can the system be further designed so that the non-functional requirements mentioned will be met, and how does that design relate to further refinements of the functional and structural aspects of the system.

Figure 12 shows how the deviation pattern is applied and what impact it has on the functional structure of the system. To avoid cluttering the diagram, figure 12 hides some of the argumentation goals shown in figure 10. On the left-hand side we can see the three solutions

discussed in the pattern text, denoted by the reference_&_distribute_on_demand [input_status_data],apply_deviation_pattern[input_status_data], and replicate_on_all_ nodes [input status data] operationalizing softgoals. On the right hand side we can see how each one of the solution strategies generates alternative refinements to the input_status_data stored, input_status_data_disseminated and input_status_data_accessed functional goal. We have replaced the functional tasks store_input_status_data, disseminate_input_status data, and access_input_status_data in figure 11 with corresponding *functional goals*. Since through refining and analyzing how to achieve the non-functional part of the system, the designer arrived at the insight that storing, disseminating and accessing input status data may be done in more that one way.

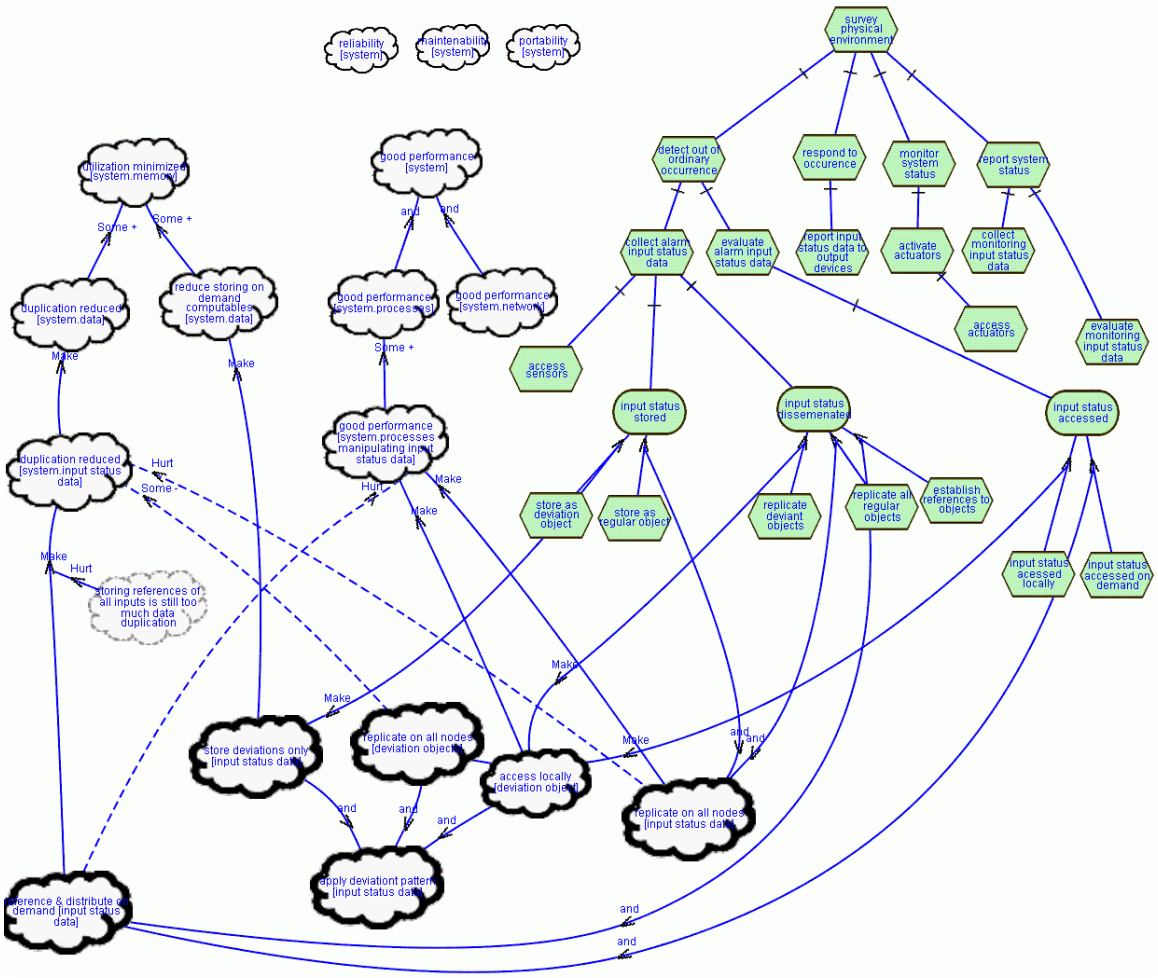

Fig. 12: Applying the Deviation pattern during design

Figure 13 shows the solution structure after applying the deviation pattern. It shows that instead of storing every input status as System_Status objects, we only store Deviation objects, which represent input status information that exceeds certain threshold limits. A Replication_Handler object was also added to replicates a copy of Deviation objects to all control units. The Alarm_List object is now aware that it refers to Deviation objects rather than to all System Status objects. It can, however, deduce system states that do not point to a Deviation object, are not exceeding a threshold and are thus in a normal state.

5.3. Understanding, analyzing & applying the Point pattern

This section illustrates two further advantages when making NFR explicitly for describing and for applying patterns during design. First, it illustrates how to express patterns that make use of other patterns in order to address particular sub-problems in their proposed solution structure. Second, it illustrates the NFR related reasoning a designer needs to make when applying several patterns during design.

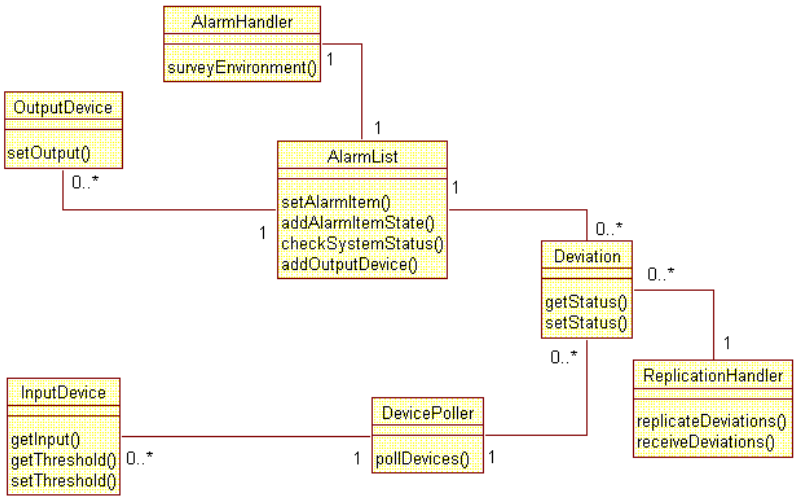

Fig. 13: System solution structure after applying the Deviation pattern

The point pattern addresses the problem of variability of I/O devices in the type of sensors, their detection algorithms, access protocols, physical packaging, and the means by which sensors, actuators and other output devices are connected to the system. However, despite the variations in the make-up of those devices, their logical behavior (such as requesting their status or activating actuators) are similar over all devices, sensors, actuators and output devices. The point pattern problem statement therefore is: "How can the logical behavior of the system be separated from the variation among input sensors and output actuators?" In other words, how can a standardized interface be established which separates the logical behavior from the device variations.

The Point pattern discusses two alternative strategies for achieving a standardized service interface. One approach is to create a standardized interface to all devices that are attached to the systems control units. The interface would be defined as an abstract base class, corresponding to an abstract device, and variations of the device implementation would be defined in corresponding subclasses. This would be the most natural approach to establish the logical device behavior within the base class and then have each sub-class implement its own way of relating to its device. It is, however, inadequate since in this particular problem context, the amount of input sensors or actuators that may be connected to one such device varies as well. This would introduce another element of variability that can not be accommodated in a stable manner within the abstract base class representing a device.

The other approach that the Point pattern proposes, and adopts, is to establishing a standard “logical” service interface for logical input and output relevant to the problem domain, such as for input sensors or logical output units. To this end the Point pattern suggests defining an abstract base class called point. A device is then covered by a set of points.

The Point pattern uses another pattern, the Bridge pattern [10] in order to decouple the Point abstraction from the Point implementation code. The bridge pattern introduces another layer of indirection (polymorphic invocation) between the clients interfacing and the implementation

code. This has some negative effect on the performance of the system when accessing I/O devices (such as accessing the sensor state information), this is not mentioned in the pattern text description but can be identified with our "middle out" approach of analysis and described in the NFR goal graph.

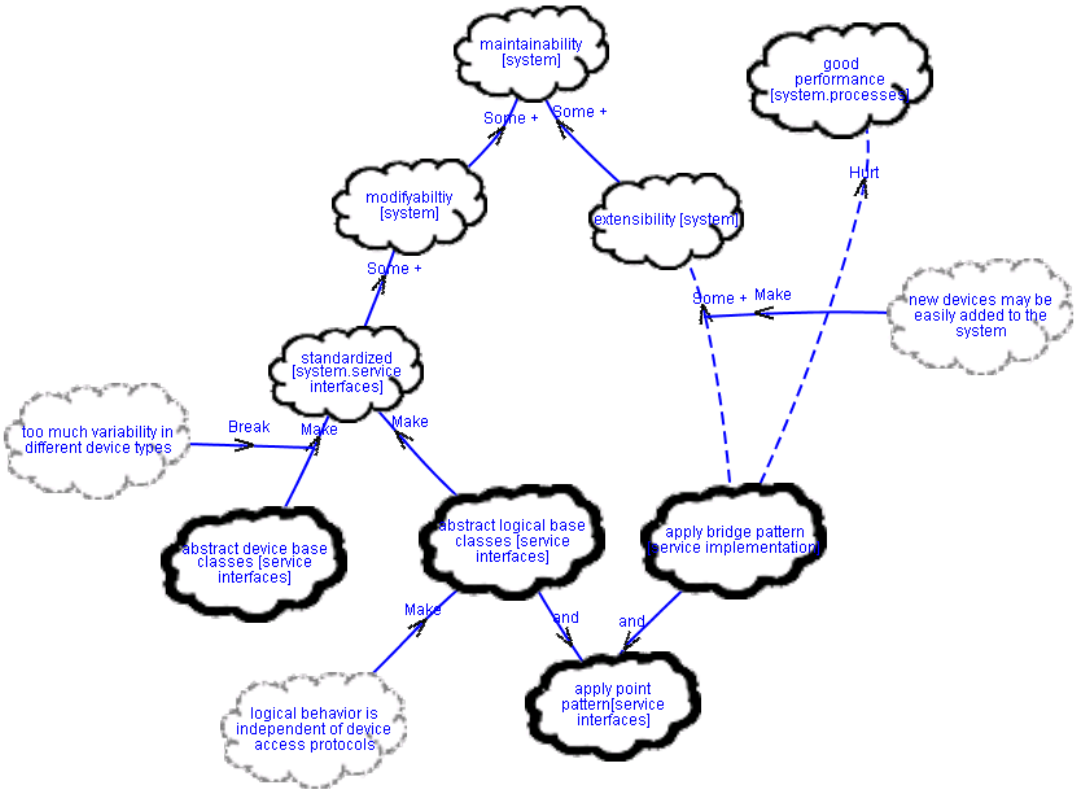

Fig. 14: The reasoning structure of the Point pattern

Figure 14 summarizes the forces and solution discussed in the Point pattern. The figure shows that the Point pattern is in fact addressing two distinct issues. It addresses the additional variability of devices by suggesting a logical rather than device oriented abstract service interface. This is addressed by the abstract_logical_base_classes[service_interface] operationalizing softgoal. The pattern also addresses the issue of separating implementation code from interface code through the bridge pattern. This is a related but different concern, and is shown by the apply_bridge_pattern [service_implementation] operationalizing softgoal. The bridge pattern addresses extensibility issues, allowing for new type of devices to be added without affecting the point "client" code.

Figure 15 illustrates how the Point pattern is applied during design. As described in the Point pattern text, the diagram shows that accessing input sensors (and output actuators) can be done in two different ways. One is through the base class abstracting the notion of device. This is denoted by the access_sensors_through_device_abstraction task. The other alternative is through the base class abstracting the notion of a logical device. This is denoted by the access_sensors _through_point_abstraction task. Both alternative tasks are linked through "means-ends" link to the sensors_accessed functional goal. Note again the switch from accessing_sensors (and accessing_actuator) task to the sensors_accessed (and the actuator_accessed) goal symbol to indicate the availability of functional alternatives, as described in the Point pattern. The means-ends links connecting the alternative tasks to their respective goals, are then related to the corresponding operationalizing goals through design justification links.

Figure 15 illustrates how applying the Point pattern may have impact on non-functional requirements ("forces") already addressed by having applied previous patterns ("the deviation pattern"). Even though having made design choices for optimizing performance (by duplicating deviations), there is no guarantee that performance would not be adversely affected during subsequent design. The use the point pattern makes of the bridge pattern may still adversely affect the overall system performance. Thus, when applying subsequent patterns, the designer needs to analyze whether and in what way already met non-functional requirements may still be negatively impacted.

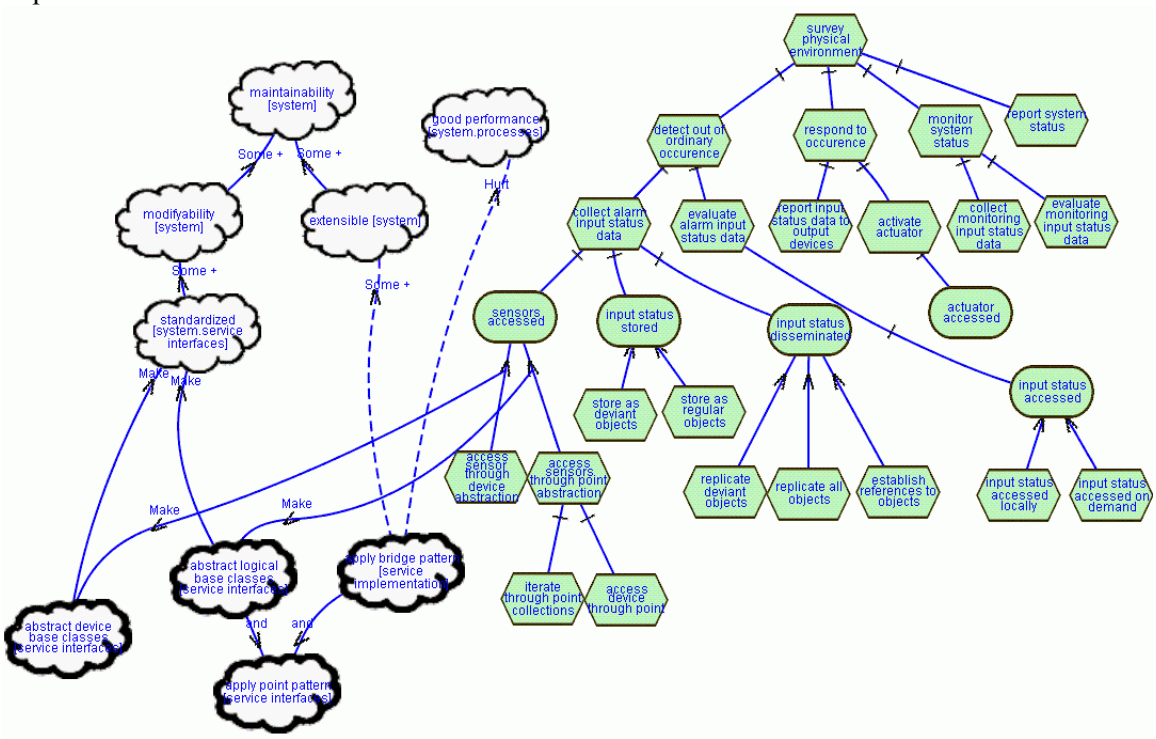

Fig. 15: Applying the Point pattern during design

Figure 16 illustrates the solution structure after applying the Point pattern. The Point object is used to access both input and output devices, and is the interface through which the Device_Poller object and the Alarm_List object access I/O devices respectively, while the PointImp object implements the different protocols to access I/O devices. There is a zero-to-many relationship between the devices and the PointImp object, which means that one input device may have several points defined, if that input device offers several sensor services to the control node.

6. Discussion

The approach presented in this paper offers a systematic way of relating non-functional requirements, through design patterns to the design of software system. It supports organizing, analyzing and clarifying NFRs in patterns, and structuring, understanding and applying of design patterns during design. NFRs that are explicitly represented in design patterns and during design aid in better understanding the rationales of design, and make the patterns approach more amenable to analysis, design guidance, and tools support. Treating functional and non-functional requirements as goals allows exploring and retaining a design history of alternative design choices, their rationales and reason why they were accepted or rejected during design.

The approach advocates a "semi-formal" style of modeling. It formalizes how (and how sufficiently) softgoals contribute to each other, provides some structuring of softgoals in terms of type and topic, and allows analyzing those contribution structures with qualitative reasoning methods. Although natural language together with informal graphical representations are powerful and intuitive in communicating design pattern intents and solution structure, providing some structuring can be very useful. Our semi-formal approach provides abilities to better understand and analyzing the complex tradeoffs that need to be made during design. It also provides abilities to better deal with changes in requirements and design assumption during the software development life-cycle and system evolution. The design history allows reasoning about the impact of making one NFR (say reusability) more important than another one (say, time to market). Design decisions that optimized for time to market can be found, and alternatives favoring reusability that were previously rejected may be reconsidered. Other changes such as design assumptions can be tracked. For example, the assumption of using one operating system (an operationalizing goal) over another would yield better market penetration (an NFR), may change or prove wrong and thus invalidate previously assumed contribution structures. Reevaluating the contribution structures would provide the basis for considering other, previously rejected design alternatives.

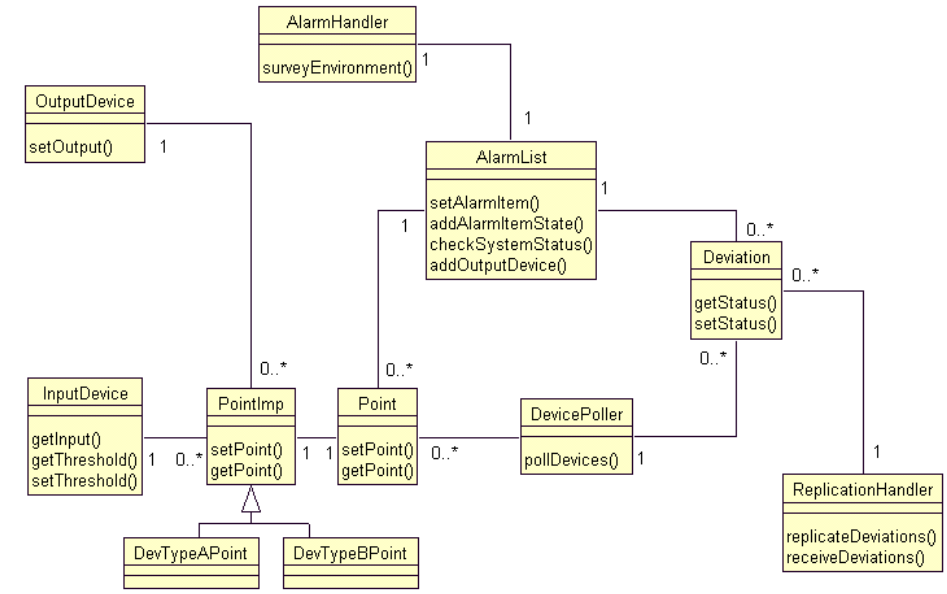

Fig. 16: System solution structure after applying the Point pattern

Such benefits of making NFRs explicit were confirmed in three case studies we performed. The case studies analyzed architectural documents of a new product provision software infrastructure, a new line of attendant console products, and the addition of a new service provision infrastructure into an existing telecommunication switching system.

Although not directly performed for patterns, the case studies pointed us to the need to consider coarser-grained constructs, such as patterns, when representing and dealing with non-functional requirements during design. In particular since we observed that designers often think in terms of solutions that address a variety of NFRs at once, rather than in terms of fine-grained (NFR) goal refinements. Furthermore, in projects were patterns were used during design it "showed up" in the NFR goal graphs, and the functional elaboration structures we produced. This motivated the approach presented in this paper, to incorporate design patterns into NFR modeling and reasoning.

The case studies showed that our approach for dealing with NFRs is applicable to complex real-life projects. Although being a simple modeling tool, and light weight in terms of formalization, NFR goal graphs, were considered very valuable by practitioners in making the argumentation structure of ongoing discussions explicit. The strength is in its simplicity, in that it allows documenting the essential NFRs when facing the tremendous complexity of telecommunication systems. Participants felt that goal graphs presented the rationales for specific choices and the arguments for them in a highly compressed and readable form. Tradeoffs could better be discussed, analyzed and clarified than when being presented in a linear text.

The need to manage change along the lines of non-functional requirements was further emphasized in the second case study. The design team learned that it was given more time than initially anticipated to complete the project. The designers were then asked to find design decision that traded-off the time to market NFR with other NFRs of the system. Without a design history as described above such information is not systematically available.

Following are further discussion points we wish to address:

Global vs. local design constraints: One of the major challenges when dealing with NFRs is that they are constraints that need to be dealt with globally, while introducing "coordinated" design techniques locally. We deal with NFR that are global constraints over the system, by carefully refining and connecting them in terms of type and topic to more local NFRs, until the designers arrives at local design techniques (operationalizing softgoals) that she can apply. Having an NFR goal graph structure allows justifying how those local techniques in fact contribute to the global constraints introduced by "global" non-functional requirements.

Generic vs. Domain specific know-how: Molin & Ohlsson [22] discuss their concern of what the best level of generality ought to be for documenting their patterns. They explain that their patterns are specific to fire alarm systems, however, most of their patterns reported are not that specific, and could be rewritten for a wide range of systems, to make them more accessible to a wider audience. However, they point out that stating the patterns in general terms makes them more difficult to understand at the level where the reader can actually apply them. The ability to provide for "topic" structures allows describing patterns in more generic terms. This more generic description can be used when searching for patterns and applying the pattern during design. We have not "generalized" the patterns to make them more widely applicable, in order to stay close to the pattern description given in Molin and Ohlssons' paper.

7. Related work

The approach proposed in this paper is based on the NFR Framework [3,4,5]. The NFR Framework introduces the concept of softgoal for specifically dealing with non-functional requirements during requirements analysis and design. Softgoals are treated as design goals that are ill-defined and tentative, and for which no formalizable criteria of achievement exists. It further introduces the contribution types among softgoals to provide a "softer" notion for specifying and reason about goal achievement. We adopt these concepts from the NFR framework and applied them to design patterns. Design patterns become NFR goal graph fragments that represent reusable knowledge, and that can be analyzed and applied during design. This extends the NFR framework with the ability to deal with coarser-grained problem and solution structures. It further provides a richer representation for the relationships between non-functional requirements, functional requirements, and functional solution structures that get refined during design. The notation for representing the functional elaborations is adopted from Yu [23]. In his work Yu uses this notation in order to model the design of (social) actor behavior

in a distributed environment. Future work will adopt this distributed view for functional elaboration and allow clustering functional elaborations and alternative reasoning along the lines of stakeholders accountability and expertise.

Although there has been some interaction between the pattern and the requirement engineering communities, not much work has been done in using patterns to connect requirement to design, in general and the linking of NFRs to software patterns in particular. One interesting work that does picks up the idea of using patterns for addressing non-functional requirements in architectural design is Bosch and Molin [24]. In this paper the authors propose a software architecture design method that is based on iterative evaluation and transformation cycles. First a rough high-level architecture is designed. It is then evaluated on how well it meets the non-functional requirements of the system. Evaluation is done through a variety of techniques. Having identified problems in meeting non-functional requirements, transformation techniques, which are based on applying patterns, are used to optimize weak areas in the architecture to better fulfill the non-functional requirements, while making tradeoffs among others.

Although the Bosch & Molin work gives a very useful overall methodology of how to approach the design of software architectures, the method does not deal in detail with representing, analyzing the structural properties of patterns and the refining and analyzing of the evolving architecture, how they achieve or conflict with NFRs. Our approach can therefore be used to complement their general methodology and give more particular notational support and guidance for such analysis and reasoning.

Design rationale is another related area. Lee [25] presents a goal-oriented approach to facilitating decision-making processes, which in turn extends an earlier model for representing design rationale [25]. The NFR framework builds on these earlier work with a focus on dealing with NFRs. Our work adopts the distinctions between NFR and FR goals, but incorporates FR goals to establish alternative functional elaborations, and for relating operationalizing softgoals to them during design, as initially outlined in [27]. Our work therefore falls within the area of goal-oriented requirement engineering [28,29], where goals are used to identify, analyze and refine requirements and serve as selection criteria for solution specifications.

Our approach also relates to the area of requirements traceability support [30] for the development life-cycle. Our work can be seen as specializing such traceability to particularly deal with non functional requirements, design patterns and alterative functional elaborations of the system.

8. Conclusions & future work

This paper proposed an approach for better dealing with NFRs that are presented in design patterns, and when applying patterns during design. The approach in its current form proved useful to practitioners in the field for better understanding, discussing, analyzing and documenting NFRs and related design decision during design. This section discusses several areas where our approach can be further extended to provide additional capabilities.

Representing solution structures: In this paper we have represented solution structures with the UML notation [21]. The solution structure is the outcome of an argumentation and decision process. The process is made explicit by the NFR goal graphs and functional elaboration structure. A notation, such as UML, does not allow one to expressing structures (i.e. components, interfaces, and connectors) that are still under development, and where requirements have only been partly accomplished. We are currently working on a representation of solution structures

that have a richer semantics for representing a structural view of the system that is only partially developed, and where functional (and non-functional) goals have not yet been fully developed into solution structures.

Scalability and tools support: Although NFR goal graphs capture a lot of essence, they may become quite large. Current research is underway to better deal with large NFR goal graphs. This is done by clustering them along the lines of stakeholders who are responsible for their achievement. These enhancements are done within the larger research goal to establish an actor oriented approach for requirements and design. Our research group is also working on tools support. The tool can be requested at [31].

Applicability of patterns during design: Our approach recognize the need for "applicability conditions" to make the application of design knowledge (such as those captured in design patterns) more selective, and context dependent, when applied during system design. For example, let us suppose that the designer of the Point pattern wishes to emphasis that the pattern should only be applied to service interfaces of devices that have no hard real-time requirements and their sensing of the environment is not critical to the systems' survival. However, for certain alarm conditions direct access to input devices is needed to fully optimize performance. The "pattern designer" would attach this applicability condition to the patterns operationalizing softgoal, suggesting that the designer who is applying the Point pattern during design should not tradeoff performance and survivability with maintainability and extensibility of the system. Further work needs to be done to explore how applicability conditions are best specified and utilized when applying patterns.

Patterns with alternative implementations: Some patterns in the literature provide alternative implementations, where each implementation may have some impact on further NFRs. We address such issues by partitioning the design process (i.e., the NFR goal graphs, and the functional elaboration structure) into layers of design decisions. In this sense implementation decisions may be seen as addressed in a later "decision" layer after having adopted the pattern during the design process in a previous layer. In this paper we have not illustrated this feature. Further research is underway to utilize the functional goal construct in conjunction with actor related constructs for representing "delegated" decisions in patterns and during design. Alternative implementations may then be seen as providing alternative choices later in the development process, while the decision itself is deferred.

Domain-dependent specializations: Our approach is domain independent. The emphasis is on properties and structures of design elements and relationships between non-functional requirements and design elements for better representing and evaluating design alternatives rather than explicitly expressing the design elements themselves. However, similar to domain-dependent architectural descriptions, that take advantage of domain specific structuring principles (such as real-time structures), our approach could be extended to include such domain specific constructs to better represent relationship structures and better allow for evaluating alternative designs in terms pertinent to a particular domain, that are often optimized for dealing with particular NFRs.

Higher-order causal relationships and NFRs in patterns: An interesting observation pointed out by one of our reviewers was the relationship between forces in design patterns and higher-order relationships discussed in [32]. Higher-order relationships are a (cognitive) means to select among relationships that exists among objects in a domain. Design patterns reason about such relationships among design objects in terms of their NFR properties. By making NFRs explicit, we provide support for denoting the elements involved in higher order relationships. Future work

can focus on how to utilize the theoretical framework provided in [32] to enhance the analysis, reasoning and support for applying design patterns.

Improved representation of pattern languages: In the design pattern literature, several related patterns are often organized into a "pattern language". Coplien explains that "a pattern in isolation solves an isolated design problem; pattern languages build a system. It is through pattern languages that patterns achieve their fullest power."[12]. If each pattern has clearer structure, then the relationships among patterns would also become more perspicuous. For example, patterns may be related via their positive or negative forces, via the problem structure, etc. Current pattern languages are typically represented as directed acyclic graphs with a single type of link with no clear semantics. Such graphs can be enriched by other kinds of semantic relations, showing how a number of patterns together can lead to the design of whole systems.

Better retrieval of patterns and knowledge based support. With more explicit structure, patterns can be retrieved more easily from a richer catalogue. There can be more dimensions for indexing, accessing and navigating the catalogues. Future work will focus on how to index, access and navigate such catalogues and how such catalogues could server as a basis for a knowledge base [33], for storing and guiding retrieval of design know-how that addresses NFRs and related tradeoffs during design.

NFR goal graphs facilitating the reuse of design patterns: Following from above, the reuse of design patterns could be facilitated when having a more explicit structure of the patterns' design goals. Currently, patterns need to be searched in full-text with unstructured search strings. Having an explicit structure for the NFRs a pattern comes to address, would aid in searching for patterns according to the tradeoffs they make. Future work could investigate how reuse would be facilitated. This may also include investigating how the pattern problem context could be better characterized within which the NFRs play a role.

NFR goal graphs as a basis for hyperlinked textual representations: NFR goal graphs are not intended to be used in place of a textual representation of patterns. The benefits of our approach is in the ability to convey a lot of information about structure, meaning and relationships among NFRs and solution structures in a condensed manner. Additional information can be made available through hyperlinked facilities that provide textual descriptions attached to nodes and links and their respective attributes. In such a hyperlink environment the NFR goal graph can then be used to navigate through the textual representation in a non-linear manner. This may include navigating along tradeoffs, alternative solution structures, contribution types, related problems, patterns and the like.

Linking high-level business goals to system requirements. High-level organizational and system objectives may be linked to the forces discussed in design patterns, and related to when applying patterns during design. Such links may then provide an understanding of how intended software systems and its evolving artifacts in fact contribute to the high-level, and to some extent strategic directions, an organization wishes to take. Using patterns provides a more convenient language when describing how the design contributes to those higher-level objectives, such as time-to-market, cost and the like.

The NFR driven approach and the pattern approach, therefore, can be very complementary. The patterns approach needs a way to link to requirements, while the NFR approach needs a way to aggregate its fine-grained solutions The NFR approach is goal-driven, whereas the pattern approach is solution-driven. One is top-down and the other is bottom-up. The two should be

combined because the most interesting design decisions are probably happening in the interactions in the middle.

9. Acknowledgements

Financial support from Communications and Information Technology Ontario, the Natural Sciences and Engineering Research Council of Canada, and Mitel Corporation are gratefully acknowledged. We would also like to thank the anonymous reviewers for their valuable suggestions for improvement.

References

[1]Boehm BW. Characteristics of software quality. North-Holland Pub. Co., Amsterdam New York 1978.
[2]Bowen TP. Wigle GB. Tsai JT. Specification of software quality attributes (Report RADC-TR-85-37). Rome Air Development Center, Griffiss Air Force Base NY 1985.
[3]Chung L. Representing and using non-functional requirements: a process-oriented approach. Department of Computer Science University of Toronto. Toronto 1993.
[4]Chung L. Nixon B. Yu E. et al. Non-functional requirements in software engineering. Kluwer Academic,Boston 2000
[5]Mylopoulos J. Chung L. Nixon B. Representing and using nonfunctional requirements: a process-oriented approach. IEEE Transactions on Software Engineering 1992; 18(6).
[6]Chung L. Nixon B. Yu E. Using quality requirements to systematically develop quality software. Proceedings of the 4th International Conference on Software Quality. McLean, VA, USA. 1994.
[7]Chung L. Nixon B. Yu E. Using non-functional requirements to systematically select among alternatives in architectural design. Proceedings of the First International Workshop on Architecture for Software Systems. Seattle, Washington. 1995.
[8]Chung L. Gross D. Yu E. Architectural design to meet stakeholder requirements. In: Donohue P(ed.). Software architecture. Kluwer Academic Publishers. San Antonio, Texas, USA 1999. pp 545-564.
[9]Chung L. Nixon B. Yu E. Dealing with change: An approach using non-functional requirements. Requirements Engineering: Springer-Verlag(1). London, England. 1996. pp. 238-260
[10] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns : elements of reusable object-oriented software. Reading, Mass.: Addison-Wesley.
[11]F. Buschmann, Pattern-oriented software architecture : a system of patterns. Chichester, West Sussex, England ; New York: J. Wiley & Sons, 1996.
[12]Coplien O. James. Software patterns. SIGS Books & Multimedia, 1996
[13]Beck K. Johnson R. Patterns generate architectures. Proceedings of the 8th European Conference on Object-Oriented Programming. Bologna, Italy. 1994. pp 139-149.
[14]Alexander C. The timeless way of building. Oxford University Press, New York 1979.
[15]Eden AH. Gil J. Yehudai A. Precise specification and automatic application of design patterns. The Twelfth IEEE International Automated Software Engineering Conference.1997.
[16]Florijn G. Meijers M. Winsen P. Tool Support for object-oriented patterns. Proceedings of ECOOP'97. Finland.1997.
[17]Coplien J. Zhao L. Symmetry and symmetry breaking in software patterns. Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering. Erfurt, Germany. 2000.
[18]Alexander C. Ishikawa S. Silverstein M. A pattern language : towns, buildings, construction. Oxford University Press, New York 1977.
[19]Coplien O. James. A generative development process pattern language. In: Coplien O, James(eds). Coplien J, O. Schmidt, D, C. Pattern languages of program design. Addison-Wesley. Reading, Mass.1995. Pattern languages of program design. Addison-Wesley. Readings, MA. 1995.
[20] Coplien, J. O. and Schmidt, D. C. (1995) *Pattern languages of program design,* Addison-Wesley, Reading, Mass.
[21] http://www.rational.com
[22]Molin P. Ohlsson L. The points and deviation pattern language of fire alarm systems. In: Martin RR, Dirk(eds). Pattern languages for program design 3. Corporate & Professional Publishing Group. Readings MA.1998. pp 431-445.

[23]Yu E. Modelling strategic relationships for process reengineering. Ph.D. thesis, Dept. of Computer Science, University of Toronto. 1995.
[24]Bosch J. Molin P. Software architecture design: evaluation and transformation. IEEE Engineering of Computer Based Systems Symposium. 1999.
[25] Lee J. Extending the Potts and Bruns model for recording design rationale. Proc.13th International Conf. on Software Engineering. Austin,Texas.1991.pp 114-125.
[25]Potts C. Bruns G. Recording the reasons for design decisions. Proc.10th International Conf. on Software Engineering.1988.pp 418-427.
[27]Mylopoulos J. Chung L. Yu E. From object-oriented to goal-oriented requirements analysis. Communications of the ACM 1999;42(1): 31-37.
[28]E. Yu and J. Mylopoulos `Why Goal-Oriented Requirements Engineering', Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality (8-9 June 1998, Pisa, Italy). E. Dubois, A.L. Opdahl, K. Pohl, eds. Presses Universitaires de Namur, 1998. pp. 15-22.
[29]Lamsweerde A. Requirements engineering in the year 00: A research perspective invited paper for ICSE'2000. 22nd International Conference on Software Engineering. ACM Press. Limerick. 2000.
[30] Jarke. M, Special issue on traceability. Communications of the ACM.1998; 41(12)
[31]http://www.cs.toronto.edu/km/ome
[32] Gentner, D. Structure-Mapping: A theoretical framework for analogy. Cognitive Science 1983; 7:155-170.
[33] I. Jurisica, J. Mylopoulos, E. Yu 'Using Ontologies for Knowledge Management: An Information Systems Perspective' Knowledge: Creation, Organization and Use – Proceedings of the 62nd Annual Meeting of the American Society for Information Science (ASIS'99). Oct. 31 - Nov. 4, 1999, Washington, D.C. pp. 482-496.

Appendix

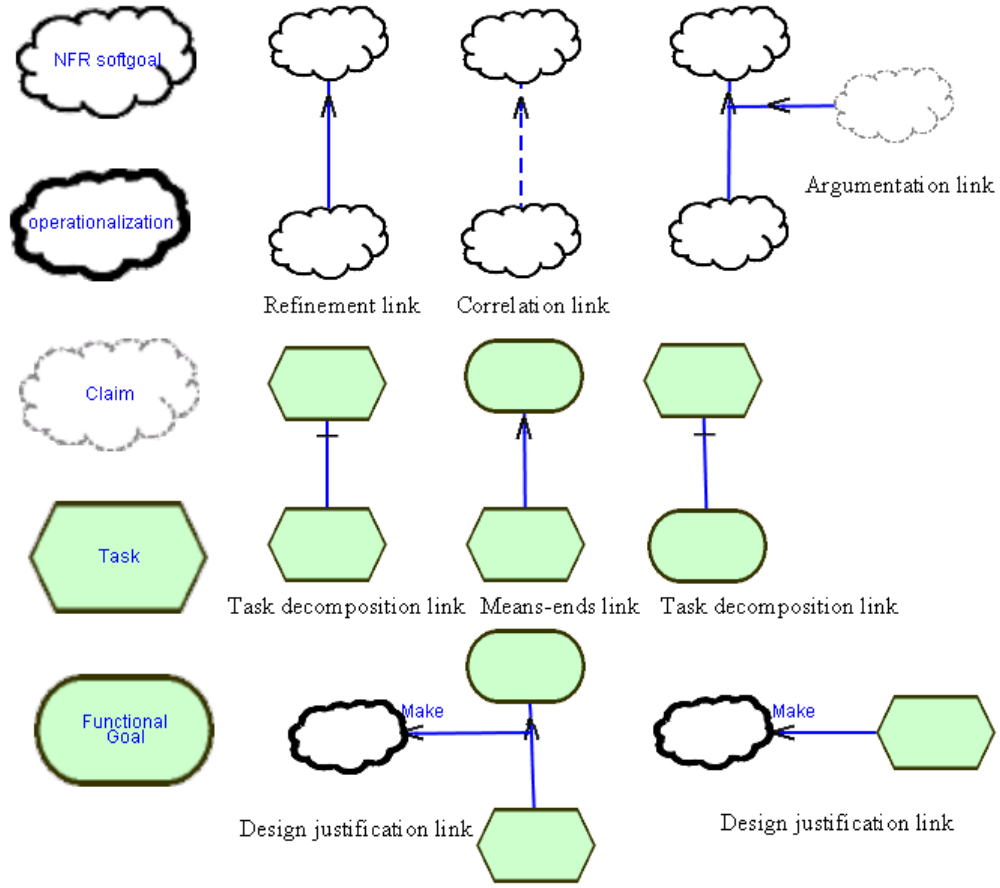

Fig. 17: Modeling Notation Legend