# Linking Patterns and Non-Functional Requirements *

Iván Araujo and Michael Weiss

School of Computer Science, Carleton University, Ottawa ON K1S 5B6, Canada

## 1 Introduction

In Software Engineering, representing and selecting patterns remains an empirical task [23]. Essentially, current pattern representations follow the format originally proposed by Christopher Alexander [1]. Although these formats have the advantage of preserving the *generative nature* of patterns, it is still difficult to recognize when to apply a given pattern, as well as the consequences its application could generate. This situation poses a challenge for most pattern users: How to select an appropriate pattern for a design decision from a broad and growing diversity of choices [20].

Given a single coherent pattern language, Alexander offers a straightforward procedure for choosing relevant patterns to be compressed into the decision maker's particular design [2]. However, with the adaptation of the pattern approach to software design and given the yearly deliberations to introduce new patterns [9][16][19][28] the number and diversity of patterns available to the designer continues to grow. Hence, it is not always practical to scan and choose from among a list of pattern titles as originally suggested by Alexander.

The *non-functional requirements (NFR) framework* [7] help us represent the relationships between design decisions and non-functional requirements explicitly. In [14], Gross and Yu examine the importance of non-functional requirements analysis in the pattern approach. They explore the applicability of the NFR framework to representing and applying patterns. Thus, Gross and Yu extract non-functional requirements and their contributions on each other from the problem section of a pattern. Patterns are applied by relating possible development techniques for achieving the corresponding non-functional requirements to the functional elaboration of the system under development.

We can go significantly further by making use of the NFR framework in understanding the forces and context of the problems that give rise to the proposed solutions. This is the main idea behind this work. The level of structuring provided by the NFR framework can help us represent patterns more formally and decide which patterns to apply in a given design context.

The overall goal of this paper is to propose a new and complementary representation for patterns, based on the analysis of non-functional requirements supported by the NFR

* This research was supported by CITO and Mitel Networks.

framework. The motivation of this new format is to promote a better understanding and structuring of patterns. Accordingly, this new pattern representation provides support for devising pattern languages.

The rest of this paper is organized as follows. The conceptual background is established in sections 2, 3, and 4. In Section 5, the proposed approach is explained. Section 6 presents a case study to illustrate the use of the new pattern representation. The paper closes with a discussion of the benefits of the new approach and some suggestions for future work.

## 2 Motivation

Applying patterns is intimately linked to the notion of pattern languages. A very simple example, building a porch [1], serves to illustrate this point. Alexander uses a set of ten patterns to satisfy a need: build a porch onto the front of a house. Besides the design of a porch, the process itself results in a pattern language. He devises a structure by starting off with the solution for a simple and broad situation (Private Terrace On The Street) and adding more and more patterns until several details in mind are considered and the given need is satisfied. The way patterns are selected allows us to intuitively understand how they are interconnected.

As Alexander states, "each part of the environment is given its character by the collection of patterns we choose to build into it." Consequently, he proposes a procedure for deriving a pattern language [1] that can be summarized as follows [12]:

1. Choose the patterns which best describe the overall scope of the project.
2. Go to the end of every pattern, where it refers to the smaller scale patterns which supports it, and make a list of the patterns which seem to apply to the project.
3. For each pattern selected in step 2, repeat step 2, and also examine the larger scale patterns at the beginning of each pattern, adding all relevant patterns to the list.
4. Repeat steps 2 and 3 until the whole list of patterns is covered.
5. Adjust the list of patterns by adding own material, either by modifying existing patterns so that they are more relevant to the current situation, or by creating new ones.

The following issues are involved in this procedure:

- A pattern user needs to scan the list of patterns to derive a preliminary subset of patterns (a pattern language). This implies reading them thoroughly until grasping the way patterns may be applied.
- It demands to pattern user several skills that depend on his/her expertise, intuitiveness, creativity, and other abilities.

As pointed out in the latter, steps 2 and 3 of the Alexander's procedure are based on designer's talent, this is a core part of his/her contribution. Moreover, this issue is pervasive in the whole process and bestows the intuitive nature to the pattern approach. On the other hand, the first issue demands great attention, particularly if the universe of patterns is broad and diverse, as currently occurs in Software Engineering.

A key issue regarding this process is the way patterns and their involved elements are described. In [10], Coplien represents forces explicitly. This constitutes a basic template for other pattern representations [6][23]. Thus, a pattern description includes a bulleted list of forces and some details about their implications. Together with a context and its corresponding solution, a discussion of forces constitutes an essential component of any pattern. Consequently, the Coplien representation provides a pattern user with an important help by describing the forces involved and supporting the pattern applicability, as well as its effects.

However, forces in a pattern context do not act in isolation. They pull and push each other until the outcome of this "conflict" is achieved, the main contribution of a pattern. Although we make the forces involved in a pattern noticeable, the following aspects usually remain unanswered:

- How are the forces related?
- In terms of resources, what are the possible trade-offs behind conflicting forces?
- What are the issues that may remain unsolved after applying a pattern?

Back to the "building a porch" example, we observe the following issues in the description provided by Alexander [1]:

- Individuality and communality needs are solved by PRIVATE TERRACE ON THE STREET (140).
- SUNNY PLACE (161) serves to emphasize the special (social) character of the place.
- The need of being covered and at the same time enjoying the open space sensation is provided by the OUTDOOR ROOM (163).
- Progressive physical development of spatial distribution is addressed by SIX-FOOT BALCONY (167), PATHS AND GOALS (120), and CEILING HEIGHT VARIETY (190).
- Some fundamental physical elements that affect the spatial distribution are a FRONT DOOR BENCH (242) and the COLUMNS AT THE CORNERS (212).
- And finally, some decorative elements that satisfy personal needs are RAISED FLOWERS (245) (olfactory and visual), as well as DIFFERENT CHAIRS (251) (visual).

As observed in this example, these issues are implied in Alexander's description. They are clear to our minds because of the simple and familiar situation addressed, and also thanks

to the nice, solid, and powerful description of patterns provided by Alexander. However, this is not the case for all situations addressed in Software Engineering, whether due to the description of the patterns involved, or due to the lack of familiarity of the designer with the problem under analysis.

The fact is that a pattern user must scrutinize the contents of every pattern before deciding to apply it. Pattern writers tend to discuss the benefits and liabilities of applying their patterns, which are identified as consequences in some pattern formats. By reading the consequences, a pattern user can determine, by subjectively trading them off, the pattern with more perceived merit [6]. Again, with the number of possible candidate patterns contending for the attention of a pattern user, this approach may become impractical [20].

Thus, we need to go further in understanding the implications of forces in pattern descriptions. We need to consider, in a more explicit way, the forces involved in a pattern description and their repercussions. In order to understand the way a given pattern works and the consequences of its application, it would be useful to count on a mechanism to represent the interaction of forces and their effects.

## 3 Related Work

This paper is linked to two topics that concern the pattern community: pattern organization and formalization.

### 3.1 Organizing Patterns

In Software Engineering, several efforts have been made for representing and organizing patterns [6][9][13][17][22][23]. In general, pattern descriptions are inspired by the original format followed by Christopher Alexander [1], namely (with minor variants), a narrative containing a context, a description of a problem, and a satisfactory solution.

The most common scheme for organizing patterns is a pattern catalog. Among others, some pattern catalogs we find in the literature available are the following: the pattern map [9][17], the pattern system [6], and the pattern roadmap [22].

However, a natural consequence of pattern organization is the conception of pattern languages as originally devised by Alexander [1]. From the first attempts to use the notion of patterns in software design, the importance of pattern languages has been emphasized [23].

### 3.2 Formalizing Pattern Descriptions

Besides the Gross and Yu's initiative [14], other efforts to treating patterns more formally are the following:

In [15], Gullichsen and Chang provide probably the earliest attempt at formalizing patterns. The authors describe patterns as morphological laws that define a permissible set of structural arrangement of the elements of architectural forms within a given context. The context of a pattern is represented by the set of patterns immediately above and below it in the pattern hierarchy. The problem is only provided in text form, and herein lies a weakness of the formalization. Finally, the solution is represented as a set of procedures and additional text for explanatory purposes.

Borchers provides a formal syntactic notation for a pattern language [4]. It represents a pattern language as a directed acyclic graph whose nodes are patterns, and whose edges reference from one pattern to another. The set of edges pointing to a pattern constitutes its context. Each node of the pattern language is itself a set of the name, ranking, illustration, problem with forces, examples, solution, and diagram of a pattern. This formal notation is used to clarify the structure of the language, and to support computerized tools for authoring and browsing pattern languages. An example of such an instrument is the *Pattern Editing Tool (PET)* developed by the author.

## 4 Patterns and Non-Functional Requirements

In this section, we examine the relationship between patterns and non-functional requirements. Initially, we will briefly describe the NFR framework, an approach very instrumental in formalizing non-functional requirements and considering their influence in design. This allows us to go further in devising the connection between patterns and non-functional requirements, as well as in exploring a way to describe pattern influence and consequence through the use of non-functional requirement analysis.

### 4.1 The NFR Framework

An observable trend in non-functional requirement initiatives is to model the chain of non-functional requirements and their contributions to particular goals. The best expression of this tendency is the *non-functional requirements (NFR) framework* [7].

In the NFR framework, non-functional requirements like interoperability, security, performance and others are used to manage the overall design process. Non-functional requirements are represented as *soft-goals*. The term "soft-goal" expresses the notion that non-functional requirements are often subjective in nature. They are usually achieved not

in an absolute sense, but to a sufficient or satisfactory extent. The NFR framework refers to the process of accomplishing the stated soft-goals as *satisficing*. This term was used by Herbert Simon in the 1950s to refer to satisfying at some level a variety of needs, without necessarily optimizing results [25].

Soft-goals are incrementally refined, and *trade-offs* are made as conflicts and synergies are discovered. The entire representation of soft-goals plus their decomposition is known as *soft-goal graph*. A *soft-goal graph* consists of one or more *soft-goals*. Every soft-goal can be decomposed into one or more *sub-goals* which, in turn, represent refinements of "parent" soft-goals, or *interdependency links*. Irregular curvilinear shapes are used to represent soft-goals and sub-goals (see Figure 1). Depending on the way an offspring positively or negatively supports the achievement of its parent soft-goal, different types of contributions are defined: *Break*, *Hurt*, *Some-*, *Help*, *Make*, and *Some+* (see figures 1 and 2).

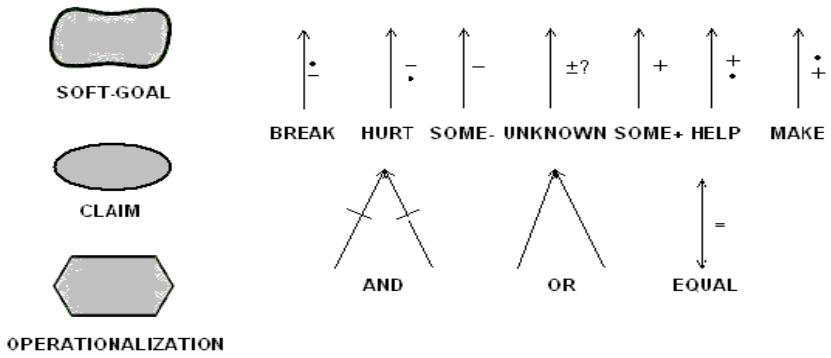



**Fig. 1.** Graphical representation for some NFR framework components and contribution links

- *Make* means that the goal sufficiently contributes to the goal at the end of the link (the parent goal). *Help* indicates a positive contribution, but not sufficient on its own. *Some+* says that there is some positive contribution to the parent goal, either *Help* or *Make*.
- Similarly, *Break* means that the offspring will break the meeting of the parent soft-goal; *Hurt* indicates partial negative support to the parent; and *Some-* indicates negative contribution, either *Break* or *Hurt*.

There are also *And* and *Or* refinements. They indicate, respectively, that all sub-goals must be achieved for the parent goal to be accomplished, or that achieving any of them is sufficient for satisficing the parent goal. These types are used to help propagate the degree of achieving soft-goals through the contribution links using a qualitative reasoning process. Figure 1 shows all the possible contribution links involved in a soft-goal graph.

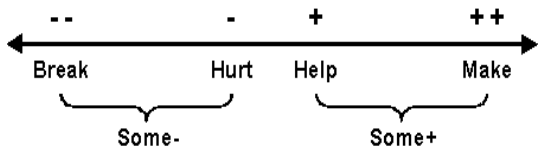



**Fig. 2.** Distinction among contribution types (adapted from [7])

Other elements of the notation are *claims*. They justify the choice of contribution type on a link and are represented as ellipses (see Figure 1). At some level, when a soft-goal is sufficiently refined, *operationalizations* come up. *Operationalizations* are possible development techniques for achieving the corresponding non-functional requirements. They are represented by hexagons (see Figure 1). By enumerating all the possible operationalizations, the developer may choose specific solutions for the target system.

Soft-goal graph representation allows designers to evaluate the impact of selecting certain operationalizing soft-goals. The evaluation works towards the top of the graph, determining the impact on higher-level soft-goals. This process is initiated by assigning labels that indicate whether a operationalization is selected (satisficed) or not (denied). In this way, a positive contribution of the soft-goal propagates the label to its parent. Hence, a satisficed soft-goal with positive contribution results in a satisficed parent, a denied soft-goal with positive contribution results in a denied parent, and so on. It is also possible to identify a weak positive or negative support for a soft-goal parent. When it is not possible to establish a positive or negative support, parent goal is assigned the undecided label. Figure 3 shows different assigned and resulting labels used during the evaluation process.

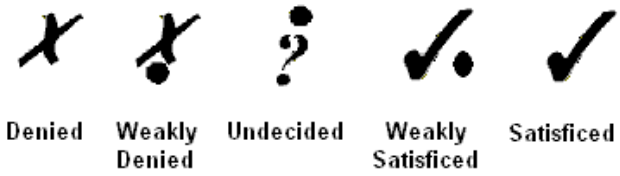



**Fig. 3.** Labels used during the NFR framework evaluation process [7]

Thus, the operation of the NFR framework is the result of the incremental and interactive construction, elaboration, analysis, and revision of soft-goal graphs.

### 4.2 Patterns, Non-Functional Requirements, and the NFR Framework

As Gross and Yu point out in [14], non-functional requirements are omnipresent in descriptions of patterns. They play a key role in understanding the problem being considered, the subsequent trade-offs, and the solution proposed. Thus, when analyzing non-functional requirements in a given context, the idea of several conflicting forces pulling and pushing emerges. By considering the components involved in determining a solution that resolves the conflict, the concept of *pattern* arises.

The NFR framework makes it possible to systematically treat non-functional requirements and ultimately use them as a criterion for making selections among competitive design alternatives. It constitutes a *top-down*, goal-oriented approach to dealing with non-functional requirements [7].

On the other hand, when considering the nature of the pattern approach, we observe that a pattern solution may have effect on certain resources (possibly constrained), as well as on some forces (related to the resources affected). Often, a solution provided by a pattern is the result of balancing/trading-off certain resources. Thus, viewing the pattern implementation at the bottom side of the picture, and the resulting context (forces) at its upper side, the use of patterns in software design constitutes a *bottom-up* approach to addressing design issues. As a solution-driven method, pattern application determines a satisfactory outcome that results from re-adjustment of both resources and involved forces.

In [14], Gross and Yu take advantage of this fact: the pattern approach and the NFR framework are complementary. Thus, they combine them by using the NFR framework for extracting non-functional requirements soft-goals and their contributions on each other from the problem section of a pattern. Consequently, patterns are applied by relating operationalizations in the soft-goal graph to the functional elaboration of the system under development.

However, in their approach, a soft-goal hierarchy is developed for each pattern (presumably from scratch), rather than for a project or domain. Hence, the scalability of the pattern description is limited because it does not take advantage of the connective rules among patterns in a pattern language.

We may go beyond Gross and Yu's idea by using the NFR framework for describing a design context, as well as a set of related patterns. As a consequence, several design issues regarding system architecture may be addressed by the integration of several patterns. In turn, a set of patterns may be reused to solve other problems found under the corresponding application domain. All of this involves the use of a consistent structure for pattern representation: the force hierarchy.

# 5 Representing Patterns Via Force Hierarchies

In this section we introduce the proposed pattern format. This new representation is intended to complement the current formats used to describe patterns. Thus, we are prepared to present the notion that sustains the new pattern representation: the *force hierarchy*.

## 5.1 What is a Force Hierarchy?

Under a pattern-based perspective, a solution is the consequence of solving several forces in conflict. Here is the base for linking the notion of forces in patterns to non-functional requirements and constraints on resources; as several design alternatives are considered, and the more adequate, the one proposed by a pattern, is applied.

A *force hierarchy* is a hierarchical graph of forces and their contributions on each other. Force hierarchies are inspired by, and can be modeled as, *NFR soft-goal hierarchies* [7]. Thus, in the proposed pattern representation, forces are expressed as a combination of a small set of constraints on common resources: time, space, computation, money, effort, and so on. In this way, trade-offs between the top-level non-functional requirements can then be translated into trade-offs between associated resources.

In [20], an intent to support pattern selection is proposed by establishing a *hierarchy of criteria* where many principles can be categorized or generalized. Thus, pattern forces are organized into a hierarchy rooted on *total satisfaction*. Following the same principle, *total satisfaction* is at the top of any force hierarchy, as the highest level soft-goal to be achieved by any system. Under a given domain, total satisfaction results in part of the positive contribution of one or more sub-goals. Likewise, there may exist some sub-goals with negative contributions. Hence, the achievement of the total satisfaction soft-goal results from the combination of several forces, some of them having positive contributions, others with negative effects. Thus, we may say that total satisfaction may be expressed as a balance resulting from several conflicting forces and the trade-off of a number of related resources.

Let us illustrate the use of the notion of force hierarchy via an example based on a problem described by Keshav in [18]. A bank is considering the appropriate number of tellers it needs in order to keep an acceptable performance and maintain its profitability. Two possibilities are considered:

- A scenario in which each arriving customer had his/her own teller, and never had to wait in line. This solution would minimize the response time and maximize the customer satisfaction. However, it would also be extremely expensive and the bank might go out of business.
- Instead of following the one-teller-per-customer scheme, the bank may consider how a single teller can assist multiple customers. In this way, the bank cannot only solve the

problem but also save money by sharing (multiplexing) a single teller among multiple customers, at the expense of building a waiting area and the time its customers waste by waiting in line.

Figure 4 shows a force hierarchy that represents the discussion of these two alternatives. Note that the forces involved (total satisfaction, profitability, performance) are placed at the upper level of the force hierarchy. Their decomposition, the following level in the hierarchy, encompassed the associated resources (money, space, and processing capacity). Subsequently, other soft-goals regarding the resource decomposition may be included until reaching the possible implementation techniques (operationalizations). In this example, for the sake of simplicity, the operationalizations considered, one teller or multiple tellers, follow the level comprising the resources involved.

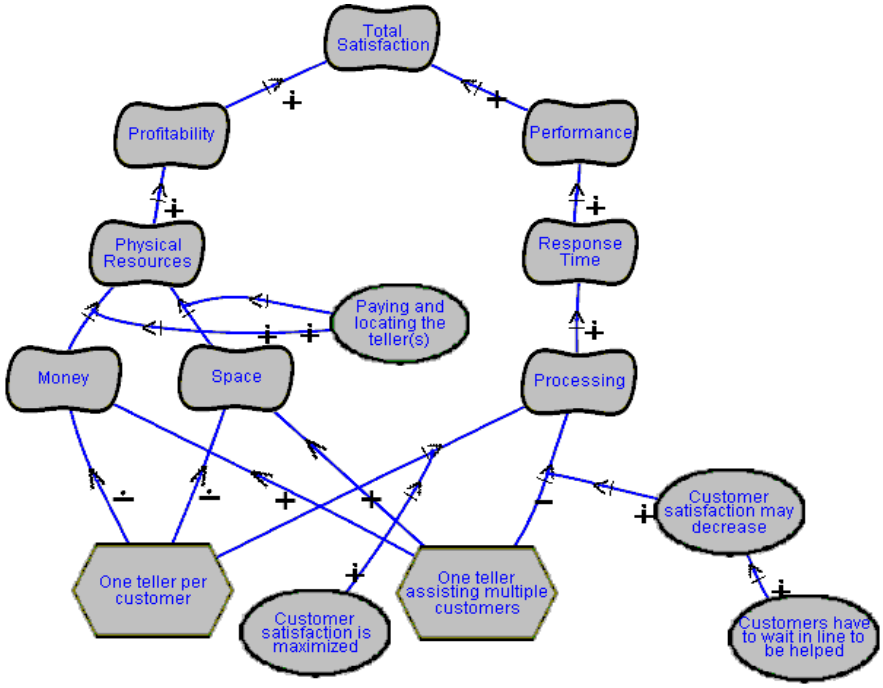



**Fig. 4.** A force hierarchy showing the alternatives in considering the number of tellers of a bank

By incorporating the evaluation process provided by the NFR framework, we can get an idea of the impact of selecting each alternative (see Figure 5). Choosing the first alternative leads to maximizing the customer satisfaction (performance is satisficed) by paying a high price (profitability is denied). The second choice implies tolerating an acceptable reduction in performance to obtain the essential profitability. It is important to highlight that the sec-

ond choice uses the Multiplexing technique [18]. This method implies sharing an expensive or scarce resource provided that users are guaranteed a certain degree of satisfaction.

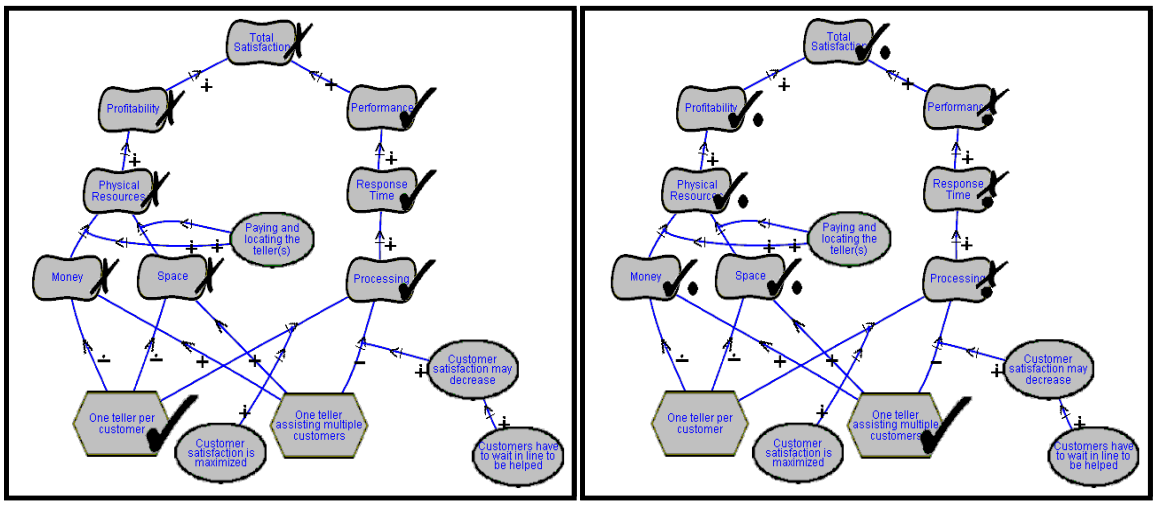



**Fig. 5.** Evaluating the impact of choosing each alternative in considering the number of tellers of a bank

There are two scopes in which we may use force hierarchies: *application domain* and *pattern solution*.

### 5.2 Application Domain Force Hierarchy

In system design, we find several design contexts, or *application domains*, such as distributed system design, real-time system design, control system design, and so on. Given an application domain, its force hierarchy consists of a representation of the top-level non-functional requirements corresponding to the design context, and the contributions of the associated resources. This establishes a starting point for representing associated patterns.

Thus, on the top of the application domain force hierarchy we find the total satisfaction soft-goal. In turn, this soft-goal is decomposed into one or more forces that describe the design context under analysis. All the resources associated to those forces comprise the next lower level in the hierarchy.

Let us go deeper in examining this structure via an example: Constructing a force hierarchy for distributed system design. First, we need to identify the forces that encompass this context. Table 1 shows some forces we have found as of significant impact on distributed system design. Likewise, we need to identify the resources related to the set of forces already established. Table 2 contains a list of resources related to these forces.

| **Forces** | **Brief description** |
|---|---|
| Performance | Extent to which system activities are accomplished promptly. |
| Scalability | Ability to provide satisfactory performance as greater demands are placed upon it. |
| Efficiency | Degree at which all the resources are employed when the system goals are achieved. |
| Maintainability | Ability to provide an adequate framework for problem fixing. |
| Transparency | Degree at which a given component does not explicitly need to know or define the status of other component when communicating (it encompasses location and time transparency). |

**Table 1.** Forces involved in distributed system design

| **Resource** | **Brief definition** |
|---|---|
| Computation | Processing that can be done in a unit time (capacity). |
| Effort | Human effort required to design and build a system. |
| Money | Cost to develop or produce a system. |
| Scaling | Minimization of centralized elements (number of additional users, features, requests, components without sacrificing acceptable performance). |
| Space | Limit on the memory available. |
| Time | Time taken to respond to an event, time-to-market, medium time between failures. |

**Table 2.** Resources involved in a distributed system design context

In order to encompass a very important issue regarding distributed system design, we have included modifiability, the ability to minimize side effects on other components, and testability as attributes that affect maintainability. Moreover, we added interoperability, as an attribute that affects scalability, in the sense that access to externally visible functionality and data structures is provided without imposing unacceptable performance degradation.

The next step involves establishing the relationships among the forces and the total satisfaction soft-goal, as well as among resources and forces in the design context. It is simpler to try to establish positive contributions, and subsequently make any necessary adjustment. In our example, we observe that all the identified forces have some positive contribution on the total satisfaction soft-goal. Likewise, we may define positive contribution links among the resources and their associated forces. Figure 6 shows the resulting force hierarchy corresponding to distributed system design.

We have to note several important aspects regarding this force hierarchy:

- No evaluation process is used [7] since we do not have the whole picture yet. We need to consider different design alternatives to examine their possible effects. Here is where pattern analysis takes place, as we will see later.
- Although the total satisfaction soft-goal appears, in practice we do not include it, as it is implied. Moreover, to avoid making the diagrams busy, we also omit the some-positive contribution symbols.
- In no way, constructing a force hierarchy for an application domain means that a perfect combination of forces, resources, and contributions coming up with total satisfaction has to be found. Ongoing refinement is always necessary as the application domain is better understood and further improvements are incorporated.

### 5.3 Pattern Solution Force Hierarchy

In constructing a pattern language, the first step involves choosing the patterns which best describe the overall scope, or *application domain*, of the project [1]. Later, we take the application domain force hierarchy and incorporate the contributions of every pattern to the involved resources, obtaining the *pattern solution* force hierarchies.

Thus, on the top of the pattern force hierarchy, we find the forces it addresses. They comprise a subset of the forces corresponding to the design context(s) where a given pattern is applicable. All the resources associated to those forces encompass the next level in the hierarchy. From that level on, a subsequent decomposition follows. This allows the pattern representation to hold the reasoning on the effects of the application of the pattern.

To avoid making the diagrams busy, in the *pattern solution* force hierarchy we only show the top-level non-functional requirements and resources actually affected by the pattern.

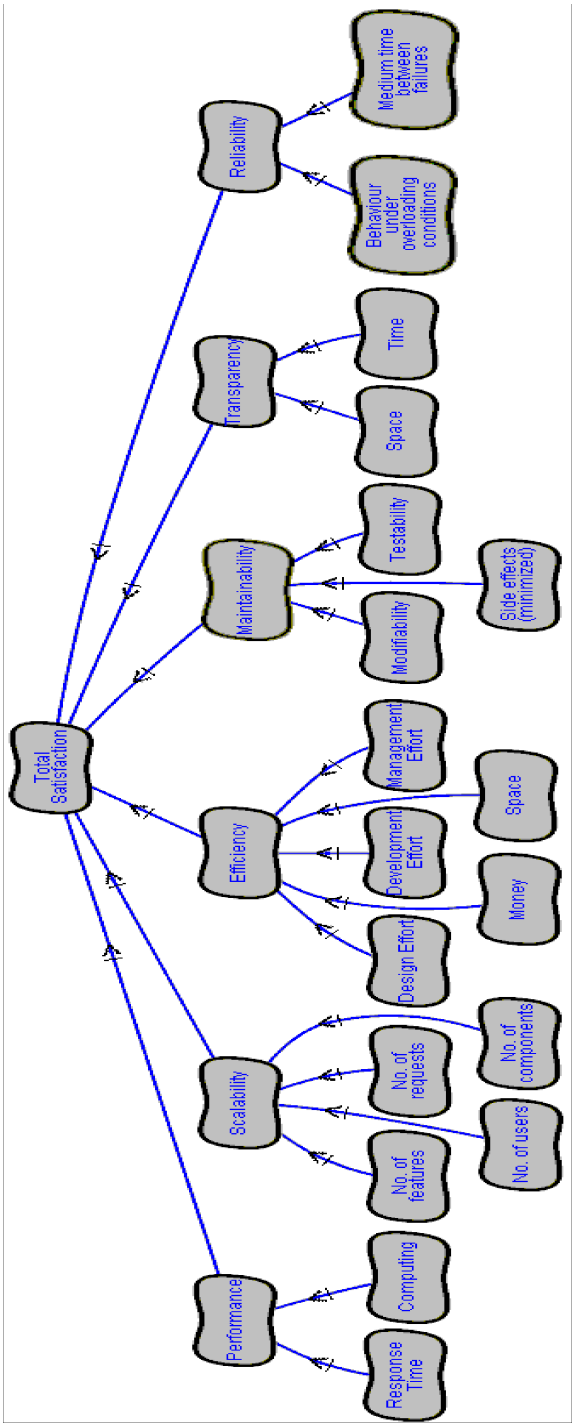



**Fig. 6.** Force hierarchy for distributed system design

As an illustrative way of explaining this issue, let us examine the force hierarchy corresponding to the Flex Time pattern [26]. This pattern is concerned with situations in which processing is required at a particular frequency or at certain time of the day. For instance, when many reports are required at approximately the same time of day. In that case, the surge in demand results in slow responses.

A brief description of this pattern can be found in Section 9. In distributed system design, it can be used to address performance issues. Thus, we may use the force hierarchy defined for distributed system design (see Figure 6) for deriving a force hierarchy for this pattern. Three forces are involved: performance, scalability, and efficiency. Figure 7 shows the corresponding force hierarchy.

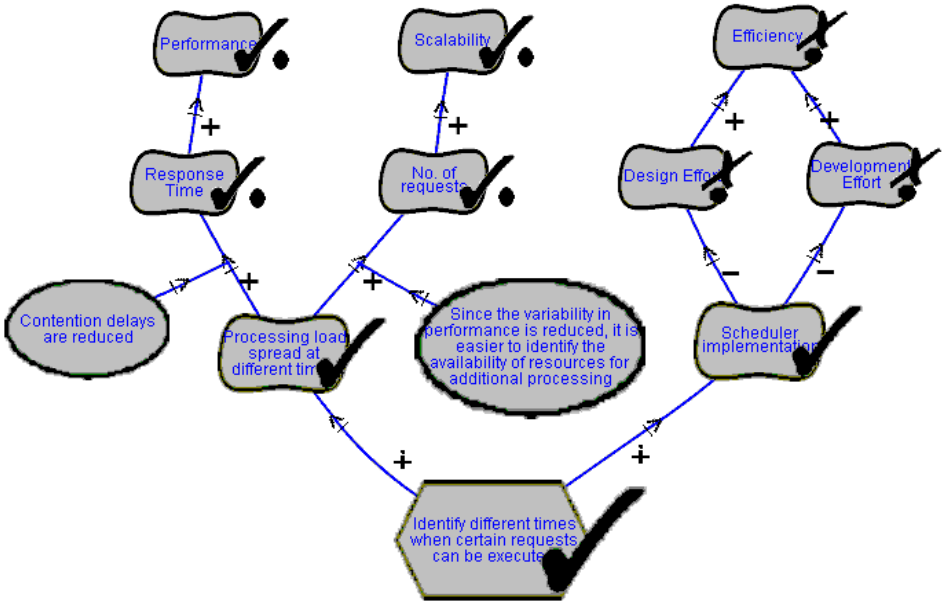



**Fig. 7.** Force hierarchy for the Flex Time pattern

The evaluation process [7] is included to show the possible effects of the application of this pattern. When considering several patterns, a pattern user is now able to compare different alternatives, the possible trade-offs they impose, and determine the most appropriate for the design issue under analysis.

### 5.4 Redefining the Roles in Pattern Application

The writer and user roles involved in pattern application may be redefined as follows:

- **Pattern writer.** S/he outlines the force hierarchy for describing an application domain. Later on, s/he uses that force hierarchy as a base for deriving the force hierarchies for all the patterns considered under the given application domain.
- **Pattern user.** S/he identifies the application domain corresponding to the system in design. This provides her/him with a set of patterns and their corresponding force hierarchies. Thus, s/he uses the pattern force hierarchies to select the appropriate ones in addressing functional issues regarding the system under design.

Pattern users and writers may also refine the force hierarchy of a pattern, or even the one corresponding to the application domain, as new relationships or forces, and possible omissions in force hierarchies become apparent when applying patterns.

## 6 Using Force Hierarchies for Pattern Application

In this section, we will explore the possible use of the new pattern representation in selecting patterns through a case study: devising the architecture of a call center.

### 6.1 Case Study Overview

As part of this research, a case study was conducted at Mitel Networks to re-examine several design issues concerning a call center project, as outlined in system documentation.

A call center is a distributed system intended to provide an effective way of connecting customers to service. It must support the distribution of incoming calls to different services or support agents [3]. Call center design involves the interaction of several components loosely coupled. These components usually cooperate in a heterogeneous distributed environment. This offers the option of examining a domain where applications may change in number of users, platforms, and components. Consequently, scalability and performance are emphasized.

The application domain that corresponds to call center design is distributed system design, with emphasis on responsiveness, given the embedded real-time nature of these applications [26]. Figure 8 shows a general representation of a call center. To avoid suggesting a specific design, elements commonly found in call centers, such as ACDs and/or PBXs [3], were not explicitly included in this depiction.

Call center system documentation contains a preliminary view of the system architecture. It also includes a definition of the functional requirements, as well as several design issues involved in their achievement. The functional requirements addressed by the call center project are the following:

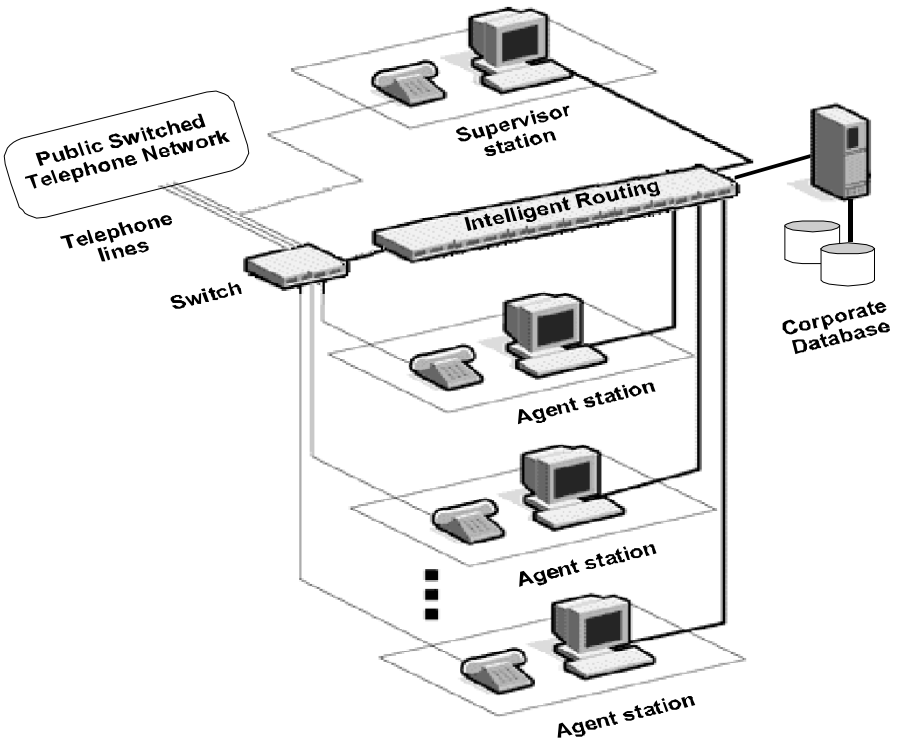



**Fig. 8.** A basic structure for a call center application (adapted from [24])

- **Computer Telephony Integration.** It provides links from telephony switches (PBXs) to customer applications.
- **Agent Automation.** It offers real-time statistics information to the agent on the current call, on his/her current performance and gives the agents easy access to interact with the call center Supervisor and other agents.
- **Intelligent Routing.** It provides customizable routing of calls based on caller inputs and system values.

For the purpose of the case study, we will limit the analysis to the first two functional requirements by addressing three design issues as follows:

- **Overall system architecture.** This design issue defines the primary structure and establishes the basis for considering the functional requirements involved in the project, as outlined above.
- **Computer Telephony Integration.** This issue is concerned with the first functional requirement. We will discuss two alternatives of design typically found in call center architectures: Using Computer Telephony Integration (CTI) vs. Flash and Dial integration, i.e., providing the same essential services by configuring telephony switches.

– **Agent station application structure.** This subject involves the second functional requirement by defining the structure that supports the implementation of agent station applications.

### 6.2 Outlining the Pattern Selection Process

We associate the impacted top-level non-functional requirements and related resources with each design issue. As there are no hard and fast guidelines on how to go about extracting the forces from the requirements, it will be often easier to start by identifying the resource constraints (e.g. reduce effort, or save money), and then the top-level non-functional requirements that will be affected. The resulting forces define the context for selecting patterns.

Thus, based upon the impacted forces and resources identified for each design issue, we select applicable patterns based on how closely they match the forces. This "best-fit" approach is currently performed by scanning the forces resolved by each pattern.

Once all patterns have been selected, they can be incorporated into the current design. Except for small designs that designers can still manage in their heads, we should defer incorporating patterns until all requirements have been considered. The reason lies in the generative nature of patterns; we have to ensure that we proceed in the appropriate order of abstraction implied by the connective rules.

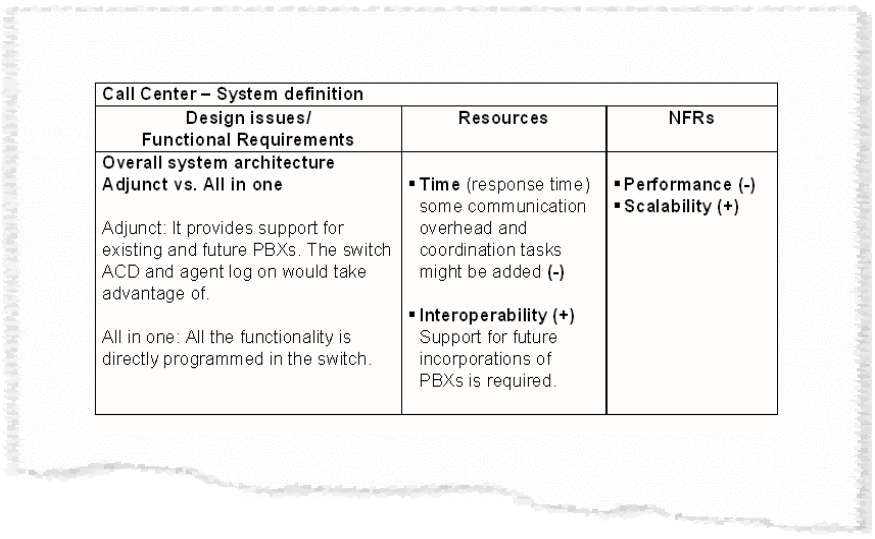
### 6.3 Addressing the Pattern Selection

Initially, 14 patterns documented in [6], [18], [22], and [26] were identified as potentially applicable to distributed and embedded real-time system design. Using the base force hierarchy we developed for distributed system design (see Section 5.2), we derived the force hierarchies for these patterns. Section 9 contains the description of these patterns along with their force hierarchies (added to every pattern description as a new section: *NFR-based representation*).

**Overall system architecture.** The first design decision we will consider is the following: *Should the functionality of the call center be provided on the Switch or in Adjunct Servers?*

Figure 9 shows the aspects concerning this issue, as well as involved resources and non-functional requirements.

From the pattern catalog the best choice seems to be the Microkernel pattern (as suggested by the context of the functional issue considered, as well as its emphasis on scalability). Figure 10 shows its corresponding force hierarchy.



| Call Center – System definition | | |
|---|---|---|
| **Design issues/ Functional Requirements** | **Resources** | **NFRs** |
| **Overall system architecture**<br>**Adjunct vs. All in one**<br><br>Adjunct: It provides support for existing and future PBXs. The switch ACD and agent log on would take advantage of.<br><br>All in one: All the functionality is directly programmed in the switch. | ▪ **Time** (response time) some communication overhead and coordination tasks might be added **(-)**<br><br>▪ **Interoperability (+)** Support for future incorporations of PBXs is required. | ▪ **Performance (-)**<br>▪ **Scalability (+)** |

**Fig. 9.** Considering the overall system architecture

A close alternative to the Microkernel pattern is the Indirection/Binding pattern [18] (its force hierarchy is shown in Figure 23, Section 9), a pattern that emphasizes maintainability and is concerned with situations when a preliminary structure has already been conceived. However, the most important point for the design issue addressed here is how to provide an efficient support for adding components. This is what the Microkernel pattern addresses as its force hierarchy highlights. It is mainly concerned with scalability and efficiency, and this is the reason why we chose it.

The Microkernel pattern separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a platform for plugging in these extensions and coordinating their collaboration. Effort and time are traded-off against scalability (here in the sense of allowing us to add components to the system). Consequently, performance and certain degree of efficiency are traded-off against scalability.

**Computer Telephony Integration.** The second design issue discusses the *trade-offs between Computer Telephony Integration (CTI) and "Flash and Dial" integration* (see Figure 11 for details of this design issue).

The decision is made in favor of an extra level of indirection between call center applications and the switch, thereby improving the maintainability and scalability of the system, while slightly giving up on performance. Since the call center product needs to interoperate with a broad spectrum of switches, future changes to the switch platform should not necessarily affect call center applications. Modifications should be possible as their effects on

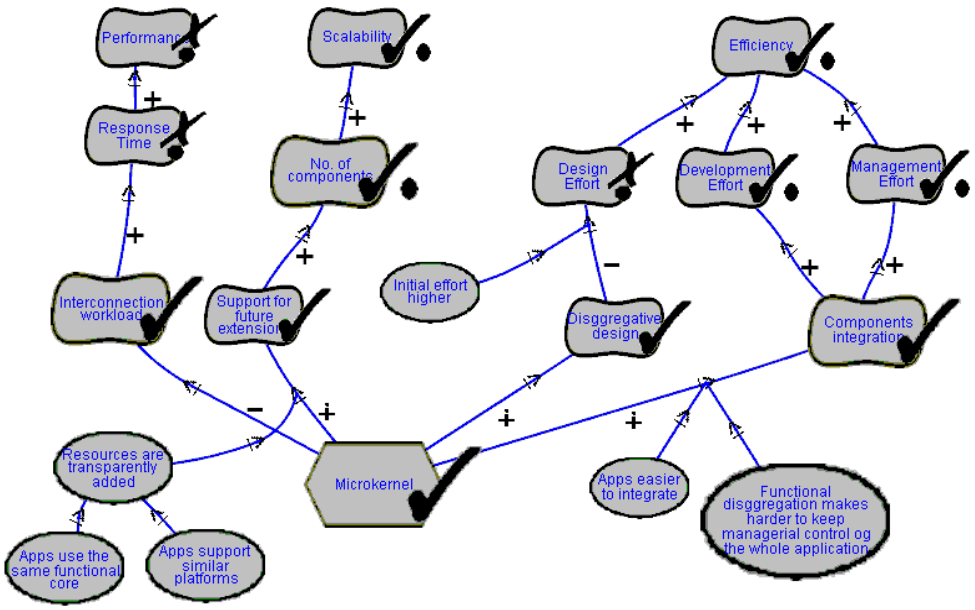



**Fig. 10.** Force hierarchy for the Microkernel pattern

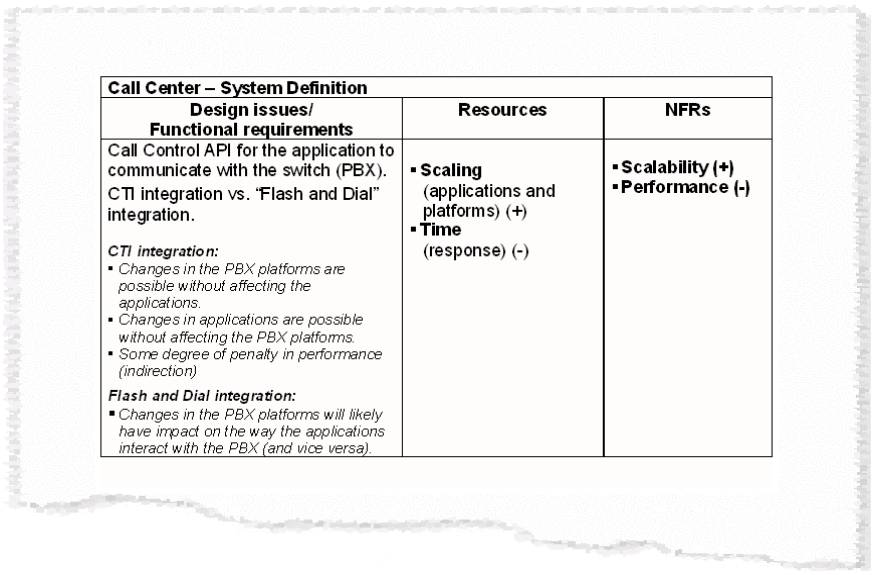
other call center components are minimized. Likewise, changes to the applications should not necessarily impose adjustments to the switch.

After matching the non-functional requirements related to this issue with the forces addressed by the Indirection/Binding pattern [18] (see its force hierarchy in Figure 23, Section 9), this pattern results in the "best fit" from the patterns contained in our catalog. This pattern also promotes the reuse of resources, a key aspect of phone services.

**Agent automation.** This time, we want to devise the *agent station application structure*. System documentation states that it has to follow the client-server model. We also want to consider the non-functional requirements associated with this design issue (see Figure 12) to define a structure that satisfies them at a reasonable degree.

Here the priority is given to the transparency (in terms of interoperability) that an agent station application requires in interacting with other software components (e.g., client-server interaction, Data Collector - agent station application interaction, and so forth). Scalability (in terms of providing support for more applications) is also a requirement. Satisfying these requirements implies giving up on performance.

Since the Broker pattern [6] follows the client-server model and addresses the non-functional requirements related to this design issue, it results the "best fit" from our pattern catalog (Figure 25, in Section 9, shows its corresponding force hierarchy).



| Call Center – System Definition | | |
|---|---|---|
| **Design issues/ Functional requirements** | **Resources** | **NFRs** |
| Call Control API for the application to communicate with the switch (PBX).<br>CTI integration vs. "Flash and Dial" integration.<br><br>*CTI integration:*<br>▪ *Changes in the PBX platforms are possible without affecting the applications.*<br>▪ *Changes in applications are possible without affecting the PBX platforms.*<br>▪ *Some degree of penalty in performance (indirection)*<br>***Flash and Dial integration:***<br>▪ *Changes in the PBX platforms will likely have impact on the way the applications interact with the PBX (and vice versa).* | ▪ **Scaling** (applications and platforms) (+)<br>▪ **Time** (response) (-) | ▪ **Scalability (+)**<br>▪ **Performance (-)** |

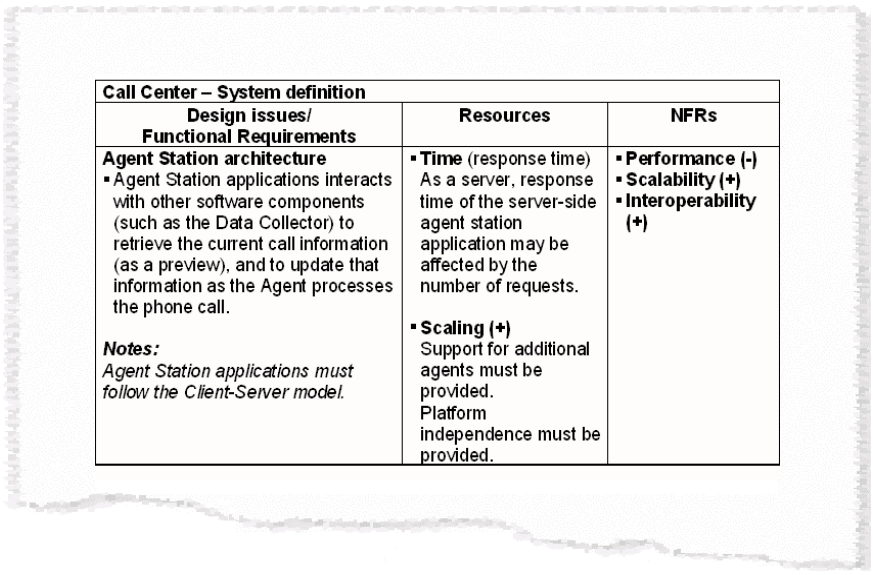
**Fig. 11.** Considering the Call Control API for switch – application interaction

### 6.4 Addressing the Pattern Application

**Microkernel.** Instantiation of the Microkernel pattern for the call center is shown in Figure 13. Its application supposes a functional disaggregation into three basic elements: the Microkernel (corresponding to the PBX switch), an adjunct component (to support any internal application added to the PBX switch), and a component that handles features such as ACD (Automatic Call Distribution) to provide support for intelligent routing. In this context, an additional component (Adapter) appears in the resulting structure. This component is not filled yet by any object (this is fairly typical for architectural patterns such as the Microkernel pattern).

**Indirection/Binding.** The Indirection/Binding pattern instantiation for the call center is shown in Figure 14. For the sake of call center implementation, this pattern imposes a 'mediator' among the switches and all the potential functions that comprise the call center applications. The role of 'mediator' is played by a CTI adapter, which provides the services required by CTI client applications. The Indirection/Binding pattern allows call center architecture to provide the core (switches) with a certain degree of independence with respect to call center applications.

The client component in this structure may be filled by the Data Collector. The Data Collector is the component responsible for gathering information from CTI components.



| Call Center – System definition | | |
|---|---|---|
| **Design issues/ Functional Requirements** | **Resources** | **NFRs** |
| **Agent Station architecture**<br>▪ Agent Station applications interacts with other software components (such as the Data Collector) to retrieve the current call information (as a preview), and to update that information as the Agent processes the phone call.<br><br>***Notes:***<br>*Agent Station applications must follow the Client-Server model.* | ▪ **Time** (response time)<br>As a server, response time of the server-side agent station application may be affected by the number of requests.<br><br>▪ **Scaling (+)**<br>Support for additional agents must be provided.<br>Platform independence must be provided. | ▪ **Performance (-)**<br>▪ **Scalability (+)**<br>▪ **Interoperability (+)** |

**Fig. 12.** Considering the architecture of agent station applications

**Broker.** The instantiation of the Broker pattern is shown in Figure 15. This pattern establishes a client-server structure for the agent station applications. In addition to this feature, an intermediate component (Broker) is located in the middle and two proxies (one for each side of the client-server structure) act as delegates for handling the client-server connections and message exchange.

**Consolidating a view of the call center architecture.** So far, we have obtained a separate view of each pattern instantiation. Figure 16 shows the resulting structure of the call center application after incorporating the three patterns into design and integrating their instantiations for the call center.

Although we still do not have information about the Data Collector design in detail, we may anticipate that it will interact with the server-side of agent station applications. It also remains to define the structure of the Data Collector and its possible interaction with the agent station server.

**Comparing the results with the actual design.** A general view of the high-level architecture description provided by the system design documentation is shown in Figure 17.

Regarding the overall system architecture, the actual designer's decision implies leaving a minimal expression of functionality in the switch and to build on separate functions

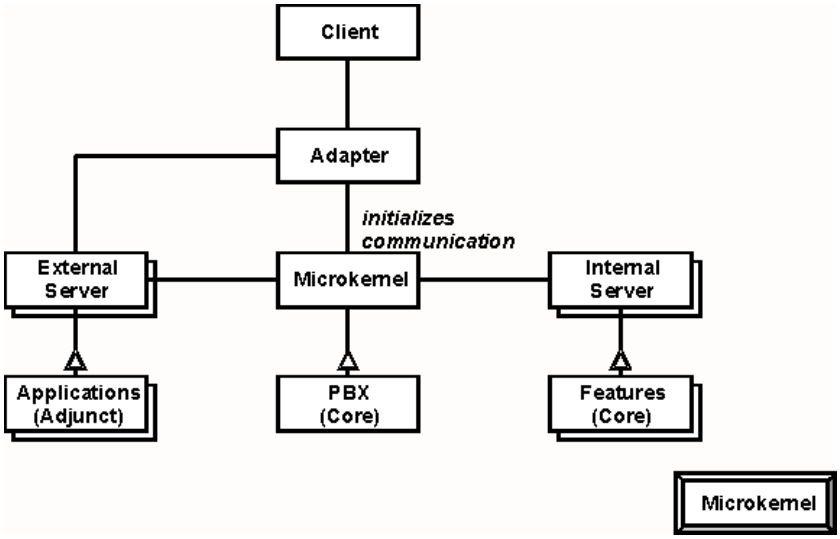



**Fig. 13.** Microkernel pattern instantiation for the call center

into other components. We observe that the core and the functions have been separated (the switch representing the core, the intelligent routing as an adjunct function). Thus, additional switches, features, and applications can be incorporated in the future. It may also be possible to make changes in call center components in run-time. This conforms to the Microkernel model predicted by the pattern instantiation.

As for computer telephony integration, the actual designer's decision entails the use of a CTI component. The main responsibility of this component, as is usual for applications based on CTI [3], is to provide an interface with a set of services for call center applications. As for the instantiation of the Indirection/Binding pattern, this introduces an intermediate layer between the switches and call center applications. This also imposes a certain degree of acceptable performance degradation. However, this penalty is paid off since, as required, changes in switches do not affect call center applications, and vice versa.

The system documentation states that the structure of the agent station application has to follow the client-server model. In addition to this, it is also established by the designer's decision that the interaction between clients and servers is handled through Remote Method Invocation (RMI) [27]. It makes possible that an agent station client application calls a remote component in a server, and that an agent station server application may also be a client of other remote objects (if needed).

For the pattern instantiation, the supporting structure (Client-side Proxy, Broker, Server-side Proxy) provides the desired scalability required for agent station applications. It also discharges agent station applications from including complex mechanisms for enabling interconnection. In this way, the Broker pattern matches the use of RMI.

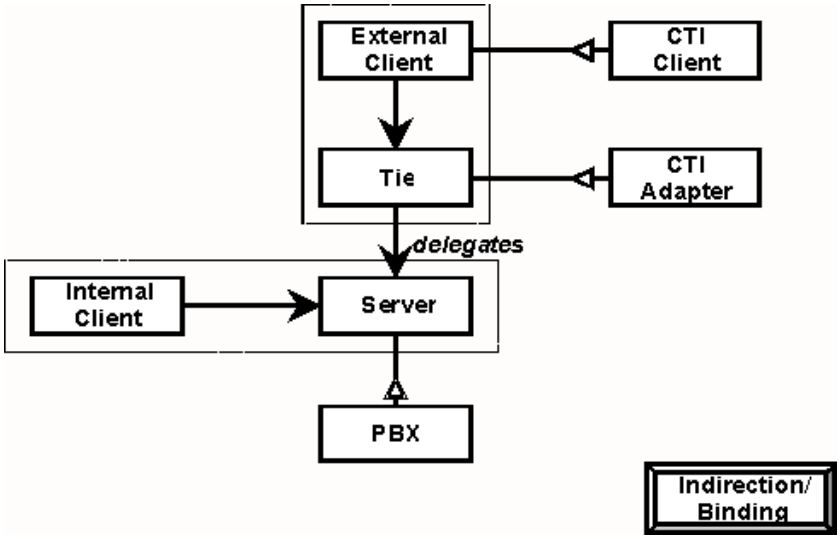



**Fig. 14.** Indirection/Binding pattern instantiation for the call center

### 6.5 Devising a Pattern Language

We could consider how the process applied to call center design might help us establish certain relationships among some patterns from our catalog. We have successively applied three patterns and derived a structure that satisfactorily conforms to the basic requirements established in the system documentation. This leads us to the conclusion that a relationship exists among the Microkernel, the Binding/Indirection, and the Broker pattern. Likewise, a sequence of pattern applications has been established. Hence, we have derived a backbone for a call center architecture and we can build on this initial interrelationship to devise an mini pattern language (see Figure 18).

Although the relationships among these three patterns were established as a consequence of their application to the case study, and this constitutes a backbone for a design scheme based on these patterns, we could also consider two possible paths to incorporate more patterns to our initial mini pattern language:

- Seek out loose ends in the new pattern representation and address their solution.
- Complete our mini pattern language by applying other patterns defined in the pattern literature.

Let us consider the first possibility. When analyzing the force hierarchy for the Microkernel pattern, a subject that demands further attention is pointed out by one of its claims: *Functional disaggregation makes harder to keep managerial control of the whole applica-*

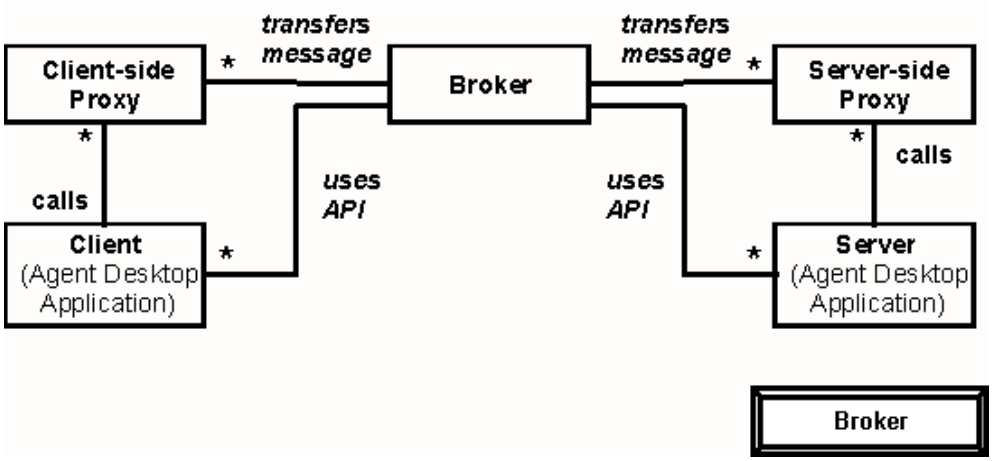



**Fig. 15.** Broker pattern instantiation for the call center

*tion.* In other words, several heterogeneous components working together might cause some difficulties for managing their common state.

This is a crucial aspect of call center design. This consequence of the application of the Microkernel pattern is implicitly established in the system documentation when stating that the entire control of a call center configuration and management may be delayed until the second phase of the project in favor of the time-to-market constraint. We can address this issue by applying the Deviation pattern [21] (to consider how to keep a compact representation of the state of the system) and the Blackboard pattern [6][11] (to determine how to establish a coordination mechanism among the components). The fact that these two patterns can be combined in this fashion to address similar problems is stated in [5].

Now let us consider the second path. In [6], it is stated that the implementation of the functional components derived from the Microkernel pattern could be addressed by applying the Layers pattern. For the purpose of call center design, this would imply using the Layers approach to implement the CTI component. Since the CTI component resulted from applying the Binding/Indirection pattern, the Layers pattern is placed after Binding/Indirection in our mini pattern language.

Considering these well established relationships, we can incorporate these patterns (Deviation, Blackboard, and Layers) to our language. The resulting language is shown in Figure 19.

We might continue to incorporate more patterns, whether by considering previous well established relationships, by figuring out aspects that still deserve attention (after analyzing the pattern force hierarchies) and the corresponding selection of more patterns.

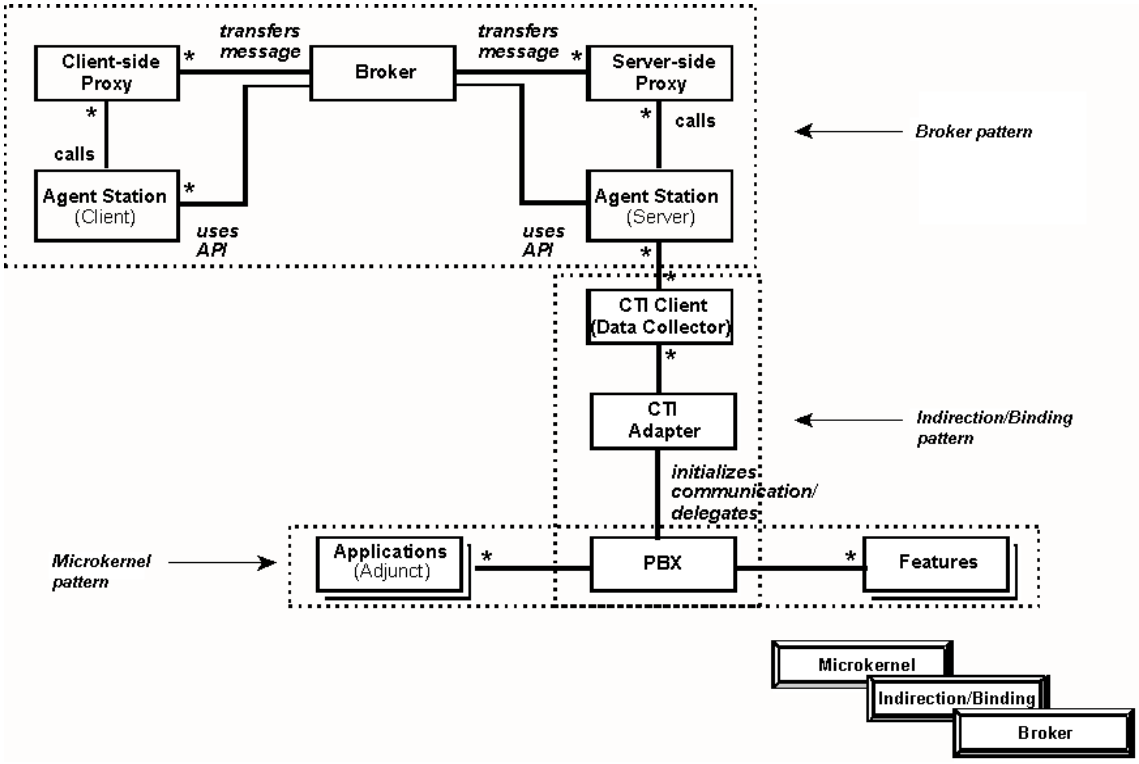



**Fig. 16.** Structure of the call center application after incorporating patterns into design

## 7 What Could Go Wrong?

Some methodological restrictions of our approach are as follows:

- Dependency on designer's expertise when the application domain force hierarchy is composed. It involves the selection of the relevant forces, as well as the associated resources and their corresponding constraints and trade-offs.
- Dependency on designer's judgement when patterns are selected.

Conceiving force hierarchies demands ongoing refinement due to the following reasons:

- At early design stages, designers might not be able to identify all the possible non-functional requirements involved.
- Designers may take into account the non-functional requirements most relevant to initial stages of design. Incorporation of the remaining non-functional requirements would be the result of further analysis on the design process.

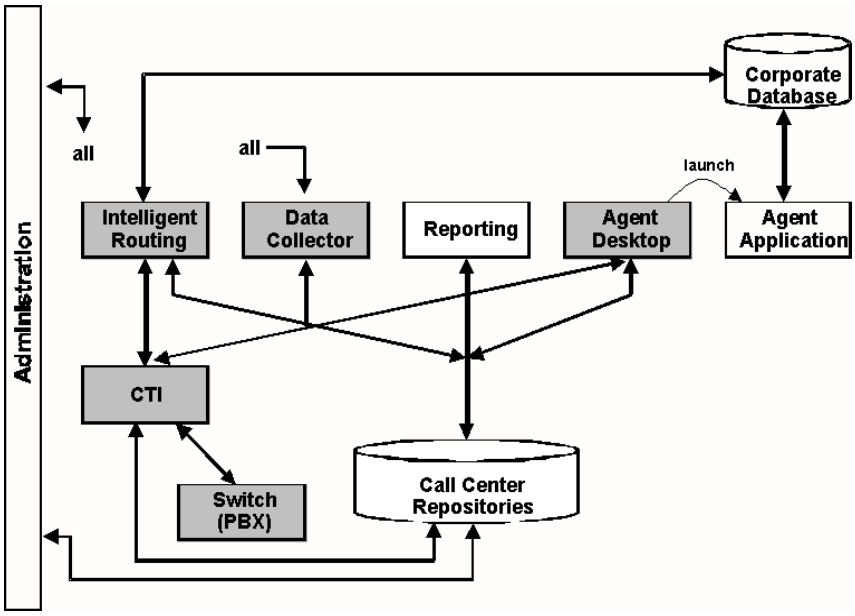



**Fig. 17.** An overview of the call center structure (actual design). Components partially addressed in the case study are shaded

Hence, there is no pre-defined procedure we could state for deriving an absolute force hierarchy for a given application domain for certain design project.

## 8 Concluding Remarks

An explicit representation of forces involved in patterns and their interrelationships is a key aspect in deriving pattern languages. It may determine the understanding of patterns interconnection in a language, as well as the consequences implied in their progressive application to any problem.

Pattern representation here proposed does not replace any of the current pattern representations. It is intended to complement them by providing pattern users with a better understanding of pattern implications and consequences in terms of forces involved. In turn, this approach also provides a mechanism for representing the same issues, forces, involved in the context of any addressed problem.

In terms of progressive refinement and possible incorporation of patterns, the nature of the approach presented here provides a way for systematic refinement of the force hierarchy of both an application domain and its associated pattern solutions. Hence, it is possible to count on a consistent platform for ongoing refinement of pattern descriptions.

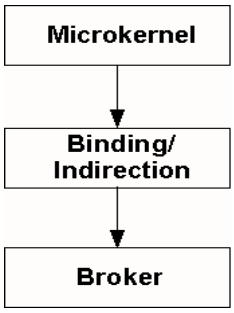



**Fig. 18.** An initial attempt to devise a pattern language

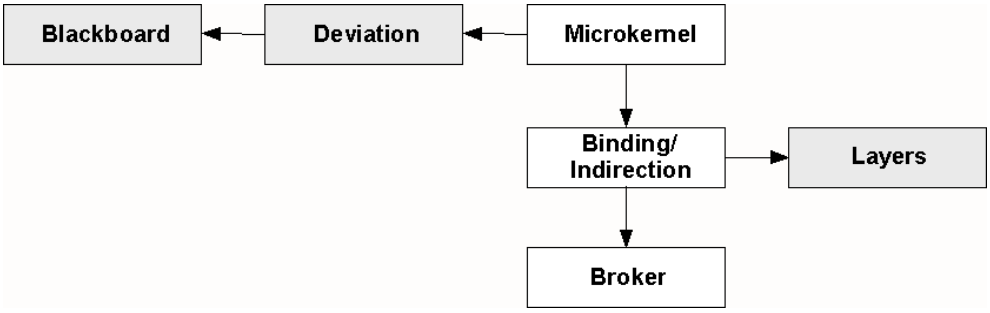



**Fig. 19.** A map of the extended mini pattern language (patterns added are shaded)

With the support of the new approach, the way several patterns are interrelated was devised in the case study. Together with previous knowledge of other interrelationships in distributed system design, a mini pattern language was conceived based on the result of the pattern instantiations resulting from the case study.

### 8.1 Potential Benefits

Some benefits of the proposed approach are as follows:

- To have a pattern representation that highlights forces, resources, and pattern contributions. This promotes understanding of patterns, and consequently, the reuse of best ideas in addressing design issues.
- To discover pattern interrelationships provides the opportunity of creating new pattern languages. This offers unlimited possibilities to search and exploit integrated solutions.
- To support understanding of patterns portability across application domains. Since the notion of force hierarchy comprises application domains, pattern contributions may be considered under different views. This helps pattern portability across several contexts, and consequently, reusability is promoted.

- The use of the NFR Framework helps designers make informed decisions and anticipate design issues for future application releases.
- Users may be encouraged to get actively involved in the analysis and design phases, by drawing on patterns and pattern languages in a more expressive way.

## 8.2 Future Directions

Our next steps may be oriented towards these directions:

- Use the proposed pattern representation in pattern application to an an ongoing project. It would also allow us to confirm its suitability in a different context and provide the opportunity of feedback and refinement.
- In a more extensive way, continue to consider the analysis of each pattern representation to devise possible consequences of pattern application. In this way, the mini pattern language considered in the case study may be extended.

## 9 Catalog of Patterns

This section presents a condensed version of a set of patterns applicable to distributed system design. In order to encompass the embedded real-time nature of the subject of the case study, some patterns with emphasis on performance were included [26].

Table 3 shows the pattern names and their corresponding source of documentation.

| **Pattern name** | **Source of documentation** |
|---|---|
| Alternate Routes | [26] |
| Balancing | [18], [26] |
| Batching | [18], [26] |
| Binding/Indirection | [18] |
| Blackboard | [6], [11] |
| Broker | [6] |
| Coupling | [26] |
| Deviation | [14], [21] |
| Fast Path | [26] |
| First Things First | [26] |
| Flex Time | [26] |
| Layers | [6] |
| Microkernel | [6] |
| Slender Cyclic Functions | [26] |

**Table 3.** Patterns contained in the catalog and their corresponding source

**Alternate Routes.** This pattern spreads the demand for high-usage objects spatially; that is, to different objects or locations. The net result is to reduce contention delays for the objects.

**Problem.** This problem occurs frequently in database systems when many processes need exclusive access to the same physical locations, usually to execute an update. When many processes must have access to the same physical location, the requests must serialize, causing delays for waiting processes.

This problem also happens when several processes must coordinate with a single concurrent process.

**Solution.** The solution for this problem is to find an alternate route for the processing. There are several strategies for finding alternate routes.

In the database access situation, we need to find a way for the accesses to go to different physical locations. Depending on the nature of the problem, there are some solutions to address this situation, such as hashing algorithms, a different key assignment strategy, accessing different tables as the updates are executed, and so forth.

An alternate route solution for the process coordination problem would be to have multiple downstream processes, each one updating its own region of the database.

These solutions spread the load spatially by routing database accesses to different areas of the database and routing processing tasks to different processes (coordination problem).

**NFR-based representation.** Three non-functional requirements are implied in this solution. This pattern trades off effort (in defining the strategy for spreading the load) with performance (responsiveness and scalability). Figure 20 shows the force hierarchy for the Alternate Route pattern.

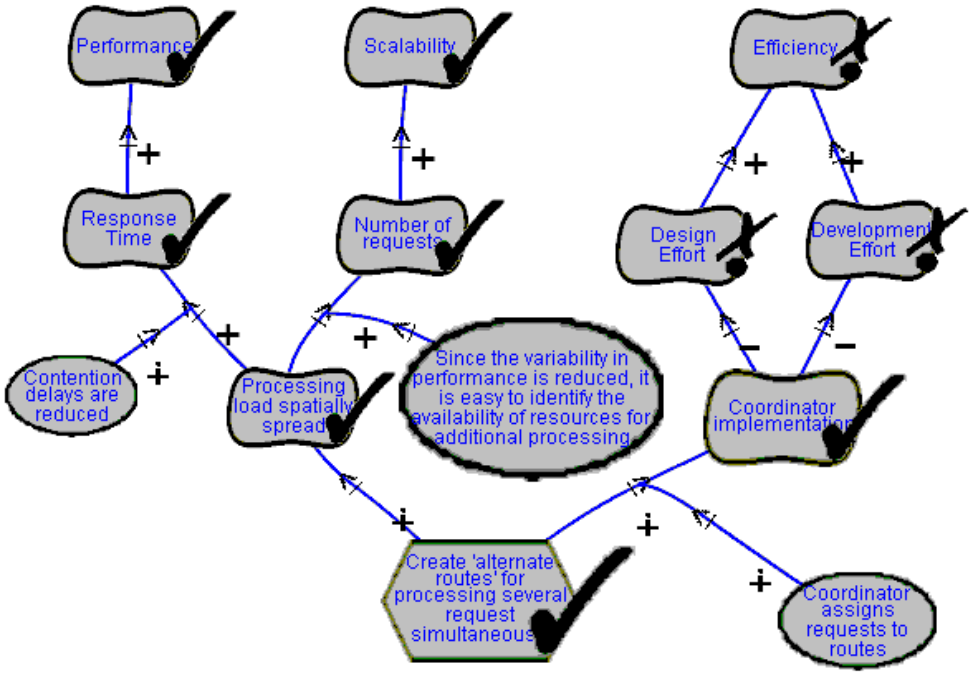



**Fig. 20.** Force hierarchy for the Alternate Route pattern

---

**Balancing.** This pattern is concerned with the removal of bottlenecks one by one until all resources are equally constrained.

---

**Problem.** Assume that we want to increase the speed with which we store data on a tape drive attached to a computer through an I/O bus. The three components affecting the speed are the CPU, the I/O bus, and the tape-drive write mechanism. Measurements may show that the slowest component in the system is the tape drive. Then, no matter how fast the CPU or the I/O bus is, system performance will not improve unless the tape drive's performance does.

Suppose we now replace the drive with a faster one. Then, we may find that the I/O bus is too slow to match the tape drive. Thus, we have a new bottleneck: the I/O bus. In order to remove the new bottleneck and get a balanced system we must consider another approach.

**Solution.** The bottleneck device is the one with the highest utilization. This is the device that limits the scalability.

In general, this solution implies identifying the component that will saturate first. It is the resource with the highest demand. By progressively identifying and eliminating bot-

tlenecks, a balance is reached. It occurs when all resources are equally constrained and performance goals are achieved.

Several approaches can be followed: changing the software, using a faster device, or adding more devices. For the example, we must improve the I/O bus to match the drive speed, perhaps by using a different I/O bus technology. This process continues until we meet the performance target or run out of resources. Ideally, the I/O bus, drive, and CPU will simultaneously reach their maximum performance, so that the system is balanced.

**NFR-based representation.** Figure 21 shows the force hierarchy for the Balancing pattern. Observe that the effect on efficiency is undecided. This is not clear yet and depends on specific implementations.

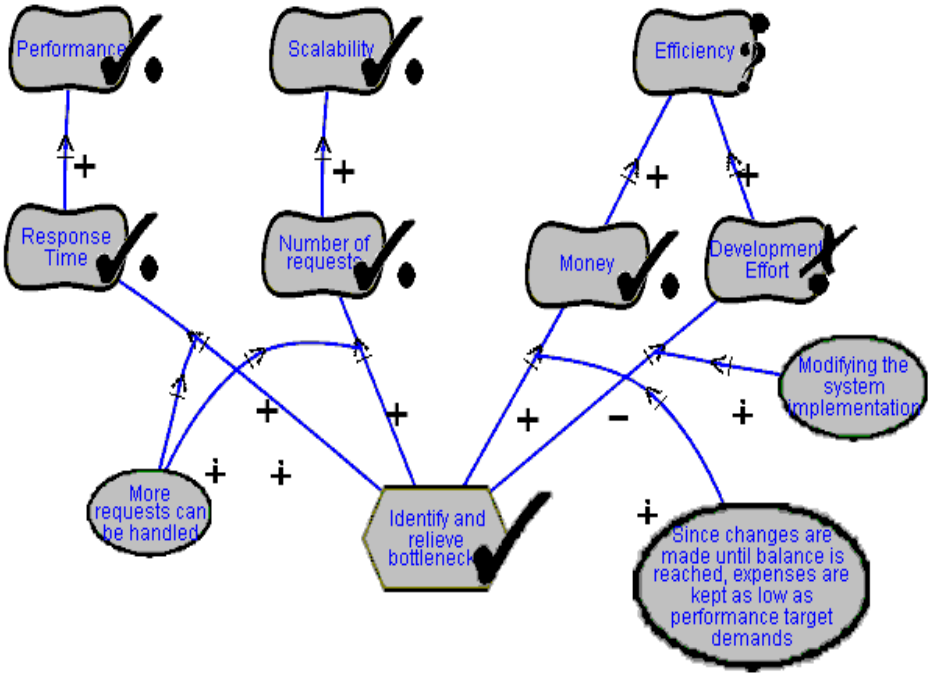



**Fig. 21.** Force hierarchy for the Balancing pattern

**Batching.** This pattern combines frequent requests for service to save the overhead of initialization, transmission, and termination processing for the requests.

**Problem.** The problem occurs when requested tasks require considerable overhead processing for initialization, termination, and, in distributed system design, for transmitting data and requests. For very frequent tasks, the amount of time spent in overhead processing may exceed the amount of real processing on the system.

**Solution.** The solution to this problem is to combine the requests into batches so the overhead processing is executed once for the entire batch instead of for each individual item.

Batching can be performed by the sender or the receiver of the request. With sender-side Batching, the sender collects requests and forwards the collection in one batch. With server-side Batching, the receiver collects requests and, when a batch is received, begins the processing for all of them.

The Batching pattern minimizes the product of the processing times the frequency of requests. Batching does slightly more work per execution, but reduces the frequency of execution, thereby yielding a smaller product (than individual requests).

**NFR-based representation.** This pattern trades off effort (in designing and developing the batcher) with performance (responsiveness and scalability). Figure 22 shows the corresponding force hierarchy for the Batching pattern.

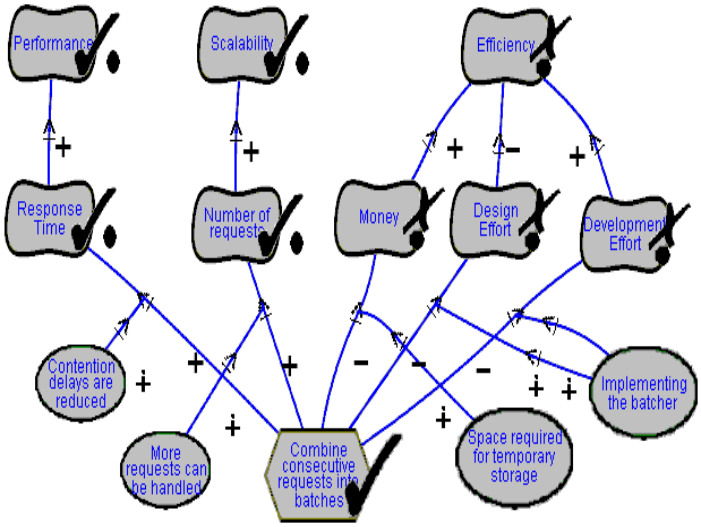



**Fig. 22.** Force hierarchy for the Batching pattern

**Binding/Indirection.** This pattern handles situations in which several components may change their location.

---

**Problem.** Suppose we have a friend, John Doe, who frequently changes e-mail addresses as he moves from one Internet provider to another. Hence, keeping in touch with John represents a problem because whenever we want to send him an e-mail we have to find his current address.

**Solution.** A system can be described at several levels of abstraction. The greater the level of abstraction, the more the generality of the description. However, at some point, we need to go from the general to the specific. This is called binding an instance to an abstraction.

If we store the translation from an abstraction to its current instance in a well-known location, the system can use this information to automatically dereference the abstraction. This is called indirection.

We could save some trouble if we had an alias for John, say "jdoe", and updated an alias file whenever John's address changed. Thus, we bound an alias to the current e-mail address for John Doe. Indirection allows us to always send mail to "jdoe" as the system translates this automatically to the current e-mail address.

Indirection allows us to dynamically associate instances with abstractions by modifying the translation stored in a well-known location.

**NFR-based representation.** Figure 23 shows the corresponding force hierarchy for the Binding/Indirection pattern.

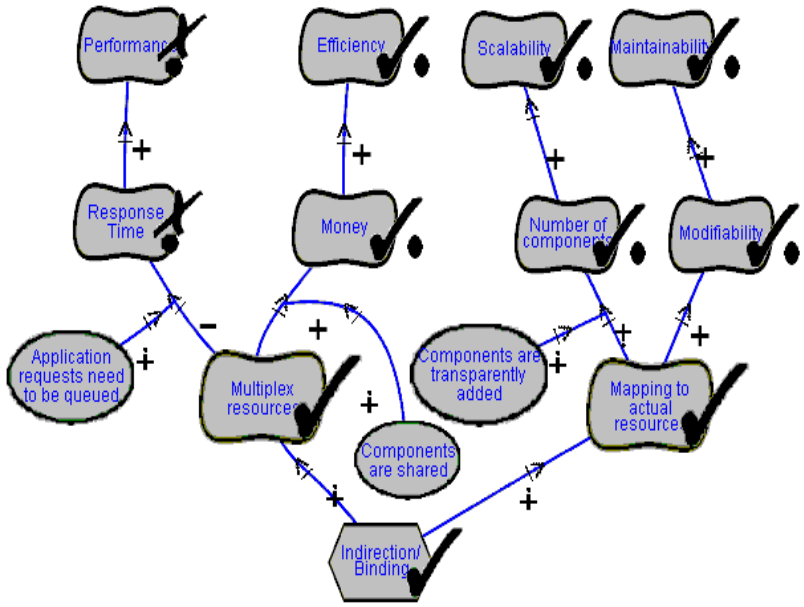



**Fig. 23.** Force hierarchy for the Binding/Indirection pattern

---

**Blackboard.** A set of agents specialized in performing different tasks is intended to perform a particular subtask of an overall task. This pattern provides a means that supports monitoring and building on mutual progress.

---

**Problem.** Agents need to collaborate to perform complex tasks that extend beyond their individual capabilities. A way to enable collaboration is to hard wire the relationships between agents. However, this is difficult to achieve if the locations of the collaborating agents are not fixed; some agents have not been created at the time an agent wants to interact with them; or if agents are mobile.

The coordination protocol needed for agent collaboration can be expressed using the data access mechanisms of the coordination medium. That means that the coordination logic is embedded into the agents. Although a logical separation between algorithmic and coordination issues would provide more flexibility, the cost of a more complex coordination medium is considered too high. Keeping the interface to the coordination medium small allows agents to be easily ported to use another coordination medium.

Hence, the problem to solve is how to ensure the cohesion of a given set of specialized agents under the forces described above.

**Solution.** The solution involves a blackboard to which agents can add data and which allows them to subscribe for data changes in their area of interest. Agents can also update

data and erase it from the blackboard. The agents continually monitor the blackboard for changes and signal when they want to add, erase, or update data.

When multiple agents want to respond to a change, a supervisor agent decides which specialist agent may make a modification to the blackboard. The supervisor agent also decides when the state of the blackboard has sufficiently progressed and a solution to the complex task has been found. The supervisor only acts as a scheduler for the specialists, deciding when and whether to let a specialist modify the blackboard. It does not facilitate their interaction with each other.

**NFR-based representation.** Figure 24 shows the force hierarchy for the Blackboard pattern.

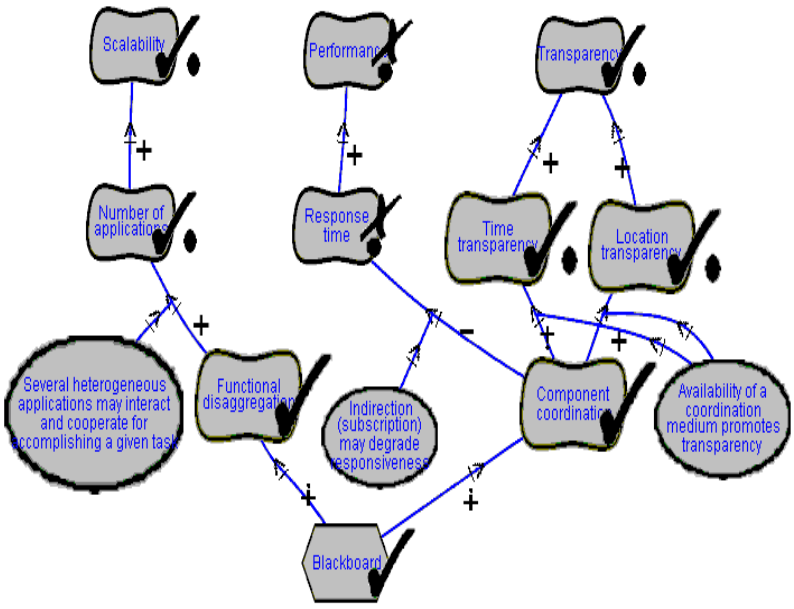



**Fig. 24.** Force hierarchy for the Blackboard pattern

**Broker.** This pattern is concerned with the structuring of distributed software systems with decoupled components that interact by remote service invocations.

**Problem.** By partitioning functionality into independent components a system becomes potentially distributed and scalable. However, when distributed components communicate with each other, some means of inter-process communication are required. If components handle communication themselves, the resulting system faces several dependencies and limitations.

Services for adding, removing, exchanging, activating and locating components are also needed. Applications that use these services should not depend on system-specific details to guarantee portability and interoperability, even within a heterogeneous network.

**Solution.** Introduce a broker component to achieve better decoupling of clients and servers. Servers register themselves with the broker, and make their services available through method interfaces. Clients access the functionality of servers by sending requests via the broker. A broker's tasks include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.

By using the Broker pattern, an application may have access to distributed services simply by sending message calls to the appropriate object, instead of focusing on low-level inter-process communication. In addition, the Broker architecture is flexible, in that it allows dynamic change, addition, deletion, and relocation of objects.

The Broker pattern reduces the complexity involved in developing distributed applications, because it makes distribution transparent to the developer.

**NFR-based representation.** Figure 25 shows the force hierarchy corresponding to the Broker pattern.

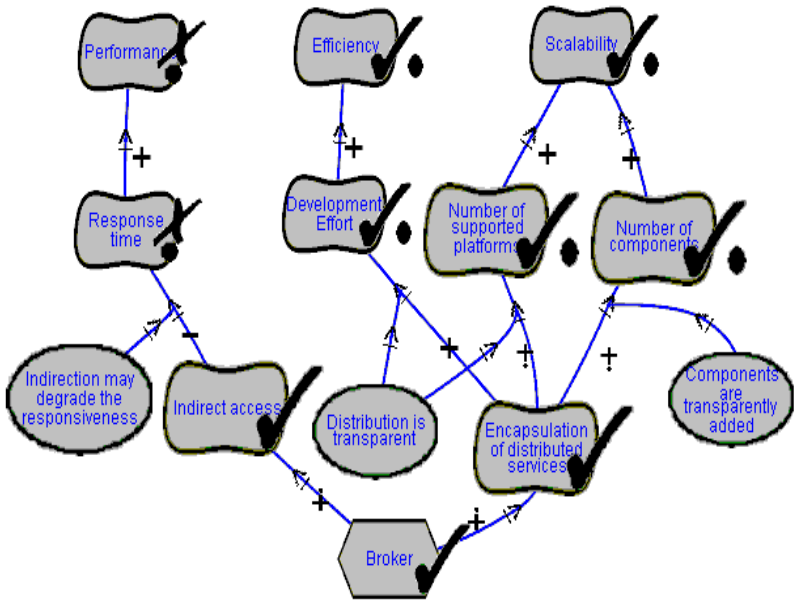



**Fig. 25.** Force hierarchy for the Broker pattern

---

**Coupling.** The Coupling pattern is concerned with the elimination of frequent requests for small amount of information.

---

**Problem.** Consider a distributed application that uses a "back-end" relational database. A way to provide an object-oriented interface to this database is to use a class to represent each table. Each attribute in the table has corresponding accessor functions (e.g., *getX* and *setX* operations) in class interface. Thus, the class structure mirrors the physical database schema.

However, this approach may lead to some performance problems. With an object for each table, the interaction between that application and the database is fine-grained and, therefore, frequent. If the database is on a different node, this generates a large number of requests; each for a relatively small amount of data. These remote requests are expensive, degrading both performance and scalability. Since any change to the database schema may have an impact on the class structure, this approach also affects negatively the system maintainability.

**Solution.** The solution is to use more coarse-grained objects to eliminate frequent request of small amounts of information. These coarse-grained objects are aggregations of the fine-grained objects that mirror the database schema.

How to construct the aggregation? This is determined by the access patterns for the data. Data that are frequently accessed at the same time should be grouped into an aggregation.

The Coupling pattern identifies interfaces that should be streamlined and combines information likely to be needed together. It also minimizes the total processing required for the interface.

**NFR-based representation.** Figure 26 shows the force hierarchy for the Coupling pattern.

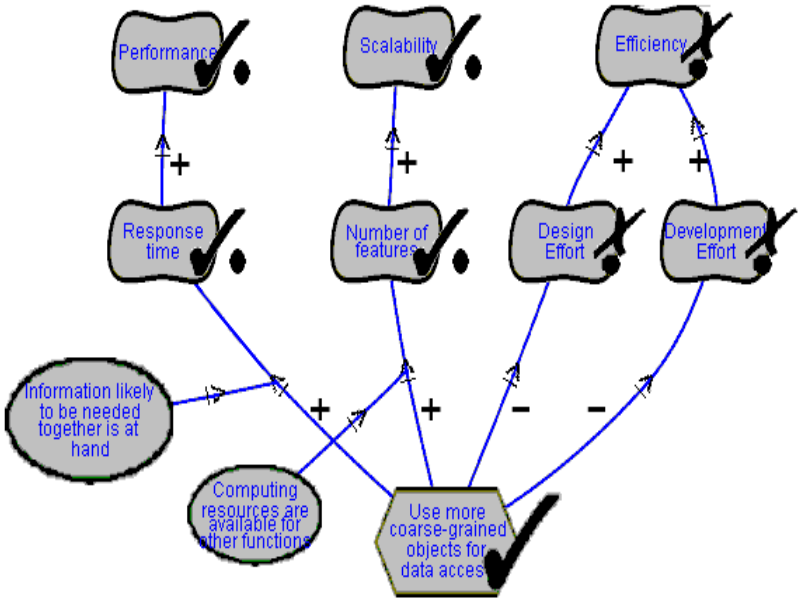



**Fig. 26.** Force hierarchy for the Coupling pattern

**Deviation pattern.** The Deviation pattern provides a compact representation of the state of the system.

---

**Problem:** The purpose of a fire alarm system is to survey a plant, a building, or some smaller unit. The system makes use of a number of sensors distributed across the area being surveyed. Each sensor is connected to one of several control units. The control units, each of which is an autonomous computing node, are all connected to a common communication network and they normally operate as a single integrated system. The key function of the system is to detect when something out of the ordinary occurs, e.g. a fire or an indication thereof, and generate an alarm.

Although a fire indication in the environment is the main thing to be detected by the system, it also monitors itself continuously for abnormal internal conditions. When faults and disturbances are detected, they cause actions to be taken in a manner similar to that of fire alarms.

The main problem to be addressed is how to implement the dependencies and information flow between alarm detection and actuators, user interfaces, and other outputs.

An alternative would be for each control unit to have only a proxy for remote inputs. Each proxy would consist of a system-wide reference, and any request would be forwarded to the control unit where the input actually existed. However, even storing as little as one reference per input in each control unit requires a great deal of memory space.

On the other hand, the ratio of alarms, faults and disturbances to the total number of inputs is very low. Most of the sensors are in a normal state most of the time. All inputs need not therefore be known at each control unit since an unknown input can be assumed to be in a normal state. Only information about deviations from the normal state must therefore be globally accessible from each control unit.

**Solution:** Represent each detected deviation from the normal state as a Deviation object. Use Deviation subclasses, such as Alarm, Fault and Disturbance to represent different kinds of deviations. Let deviations be the unit of distribution in the system in the sense that all deviations are replicated to all control units. Since the set of deviations defines the complete system state, this is immediately available on all nodes.

**NFR-based representation.** The force hierarchy for the Deviation pattern is shown in Figure 27.

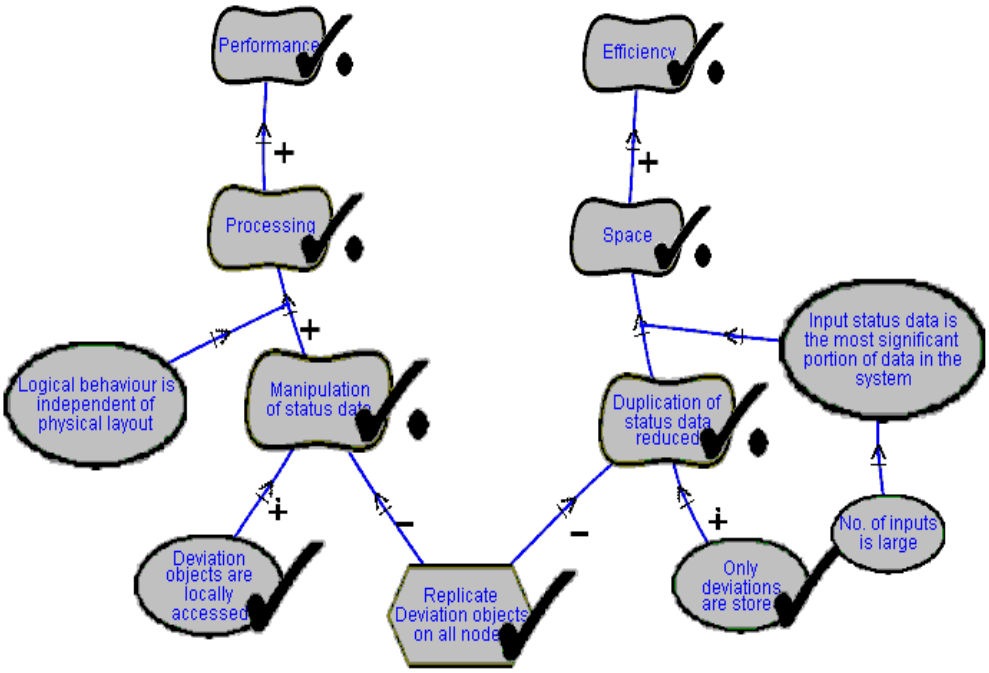



**Fig. 27.** Force hierarchy for the Deviation pattern

---

**Fast Path.** This pattern is concerned with improving responsive times by reducing the amount of processing required for dominant workloads.

---

**Problem.** Any application may be the result of implementing several functions. However, typically we find that only few of them are used most often. These functions comprise the dominant workload for the application, and usually account for most of the resource usage. For example, while an ATM may provide the capability of making deposits, or checking the balance, the most frequent use of an ATM is for making withdrawals from checking accounts. These dominant workload functions may have a significant impact on the overall performance of the system.

**Solution.** Minimize the processing for these dominant workload functions. A way to achieve this is by creating fast paths, alternative execution paths for these functions.

A Fast Path in a software application is similar to an express train that stops only at the most important stations along the route. If a user needs a major station, he can save time by taking an express train.

Similarly, in an ATM application, withdrawal functions are provided with a Fast Path by allowing users to press a single key to select a withdrawal of a predetermined amount, without having to enter the transaction type, account, and amount.

Any implementation of the Fast Path pattern reduces the overall load on the system and improves the responsiveness of other functions.

It is not enough to recognize the presence of dominant workload functions; it is also needed to ensure that the Fast Path created for minimizing their processing is effectively used. This means that it is also required to monitor the use of functions, and adapt the system to changing patterns of use.

**NFR-based representation.** The force hierarchy for the Fast Path pattern is shown in Figure 28.

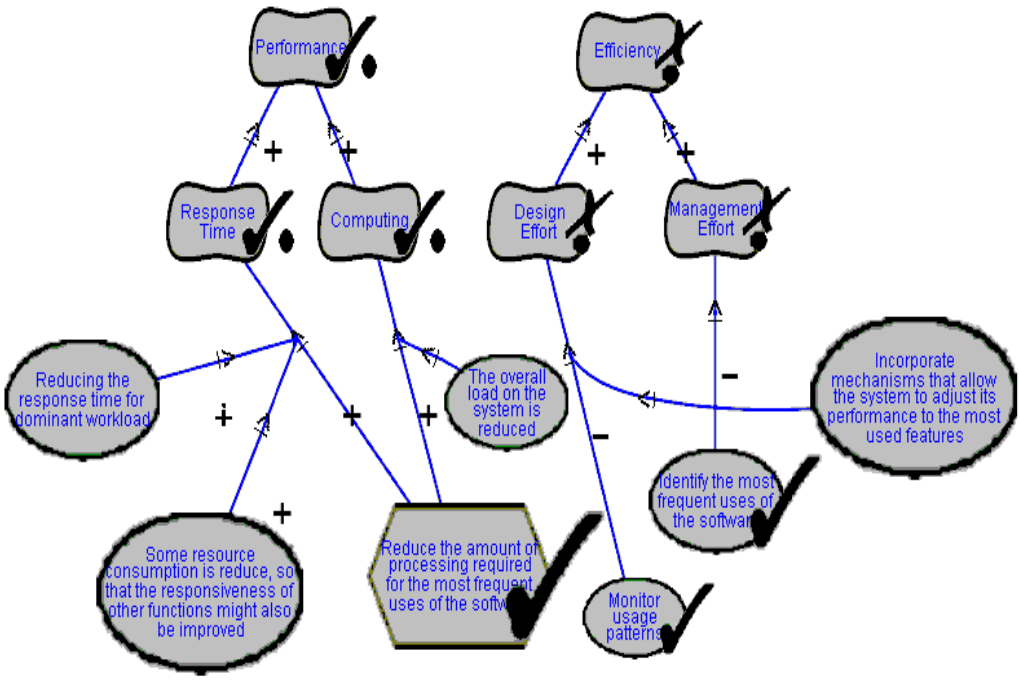



**Fig. 28.** Force hierarchy for the Fast Path pattern

**First Things First.** This pattern is based on the premise that the most important functions have to be focused and achieved. In that way, if everything cannot be completed within the time available, at least the most important tasks will be accomplished.

**Problem.** In a radar tracking system [8], tracks are processed in a first-in-first-out (FIFO) fashion. However, when track processing is exceeded, those at the end of the queue may not have been processed. Because the tracks in a sensor report usually come in the same order, whole regions of the operator display might not be updated in overload conditions. This is a potentially disastrous situation because there is no correlation between important regions of the sky and the arrival order of sensor reports.

**Solution.** The solution implies prioritizing tasks and performing the most important ones first. For the radar tracking system, applying the First Things First pattern involves computing a value for each track as it is identified. Then, by using a scheduling algorithm that maximizes the value of work completed, it is ensured that the most important tracks are up to date.

The First Things First pattern is only useful when the overload is temporary. If there are not periods of lower demand, the system will be unable to catch-up, and low-priority tasks may be ignored forever. Hence, in this and other situations when the overload is not temporary, the processing environment might require being upgraded to handle the even load and provide an adequate responsiveness.

**NFR-based representation.** See Figure 29 for the force hierarchy corresponding to the First Things First pattern.

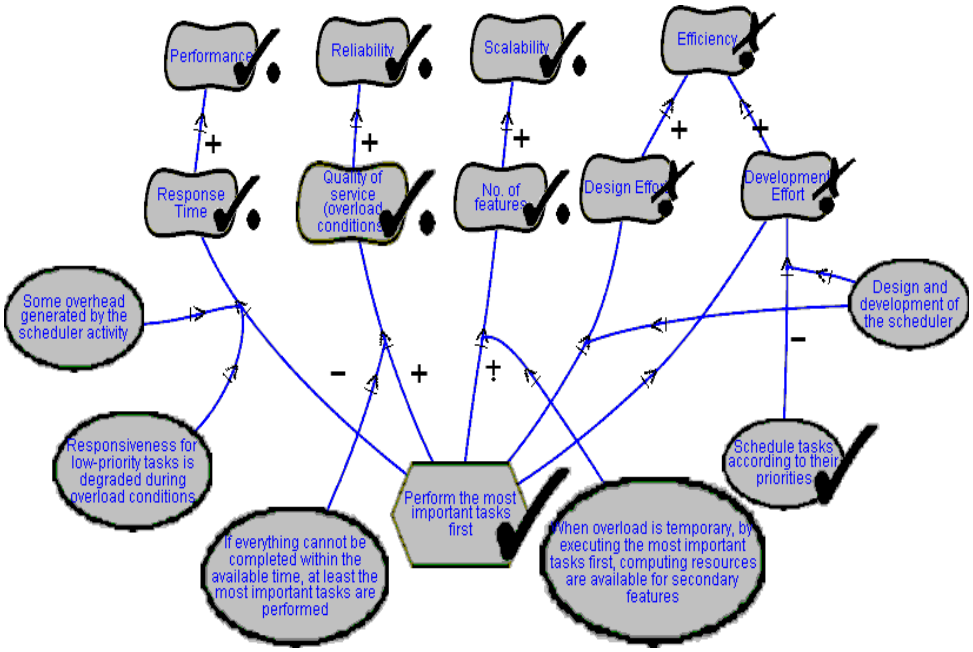



**Fig. 29.** Force hierarchy for the First Things First pattern

---

**Flex Time.** A situation addressed by this pattern typically occurs when processing is required at a particular frequency, or at a particular time of the day. When many reports are required at approximately the same time of day, the surge in demand results in slow responses.

---

**Problem.** An example of a problem addressed by this pattern is found in service centers. A service center system collects data on activity in a service system and summarizes them in a memory (RAM) data bank. Periodically, the memory data bank is copied into an archival database (in this example, every half an hour). While this archive cycle executes, postponing updates must protect the state of the memory data bank. Thus, the incoming work is temporarily halted. The problem occurs in periods of heavy loads when the queue of incoming work builds, and it takes a long time to clear this backlog and return to normal operating conditions. Hence, only a job at a time may proceed.

**Solution.** Identify the functions that are executed repeatedly at regular, specific time intervals, and modify the time of their processing. In other words, do less work more often.

In the example, archiving one-third of the data every 10 minutes could achieve this (the data structure must allow partition schemes and provide for preserving its state). Considering how the IDs of the operators and the corresponding number of calls handled

have to be archived, this solution may imply to archive one-third of the operators and their calls in one cycle. Another third of the operators is archived in the next cycle, and so on.

Thus, the load is spread temporally to reduce the congestion. In some cases, it reduces the amount of time that processes are blocked and cannot proceed. In other cases, it reduces the demand so that concurrent processes encounter fewer queuing delays for computing resources. All these results improve the overall responsiveness of the system.

**NFR-based representation.** Figure 30 shows the corresponding force hierarchy for the Flex Time pattern.

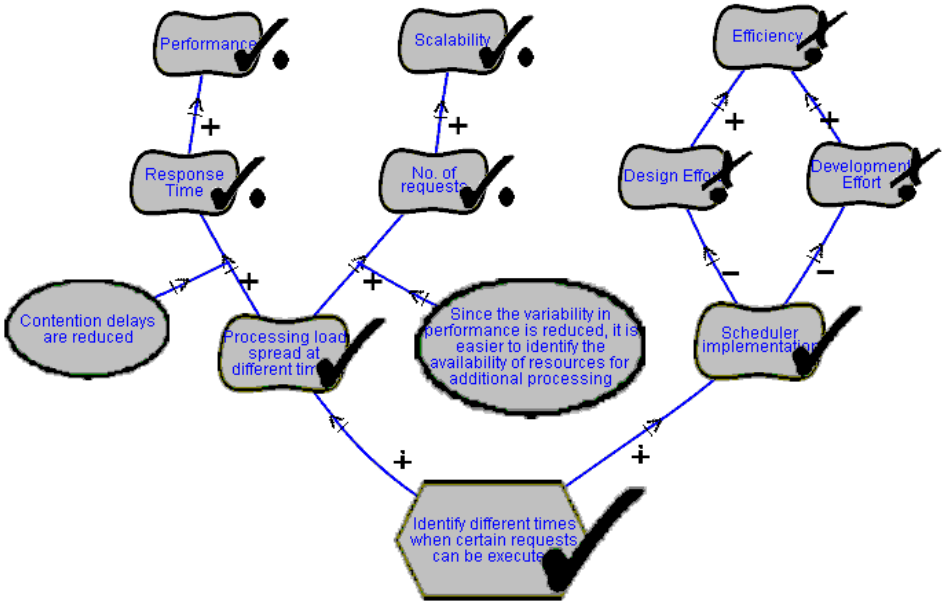



**Fig. 30.** Force hierarchy for the Flex Time pattern

**Layers.** This pattern helps structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

**Problem.** A system with a mix of low- and high-level issues is being designed. High-level operations rely on the lower-level ones. Several operations are on the same level of abstraction but are largely independent from each other. This requires some horizontal structuring that is orthogonal to their vertical subdivision.

On the other hand, portability to other platforms is desired. Several external boundaries of the system are specified a priori, such as a functional interface to which the system must adhere. The mapping of high-level tasks onto the target platform is not straightforward, mostly because they are too complex to be implemented directly using services provided by the platform.

**Solution.** Structure the system into an appropriate number of layers and place them on top of each other. Start at the lowest level of abstraction (let us call it Layer 1). This is the base of the system. System functionality is decomposed into several levels of abstraction (layers). Thus, Layer J is put on top of Layer J-1, and so on, until the top level of functionality is reached (let us call it Layer N).

This does not necessarily prescribe the order in which to actually design layers. This just gives a conceptual view. Essentially, within an individual layer all constituent components must work at the same level of abstraction.

The services of each layer implement a strategy for combining the services of the layer below in a meaningful way. Thus, most of the services that Layer J provides are composed of services provided by Layer J-1, and Layer J's services may depend on other services in Layer J.

**NFR-based representation.** Figure 31 shows the corresponding force hierarchy for the Layers pattern.

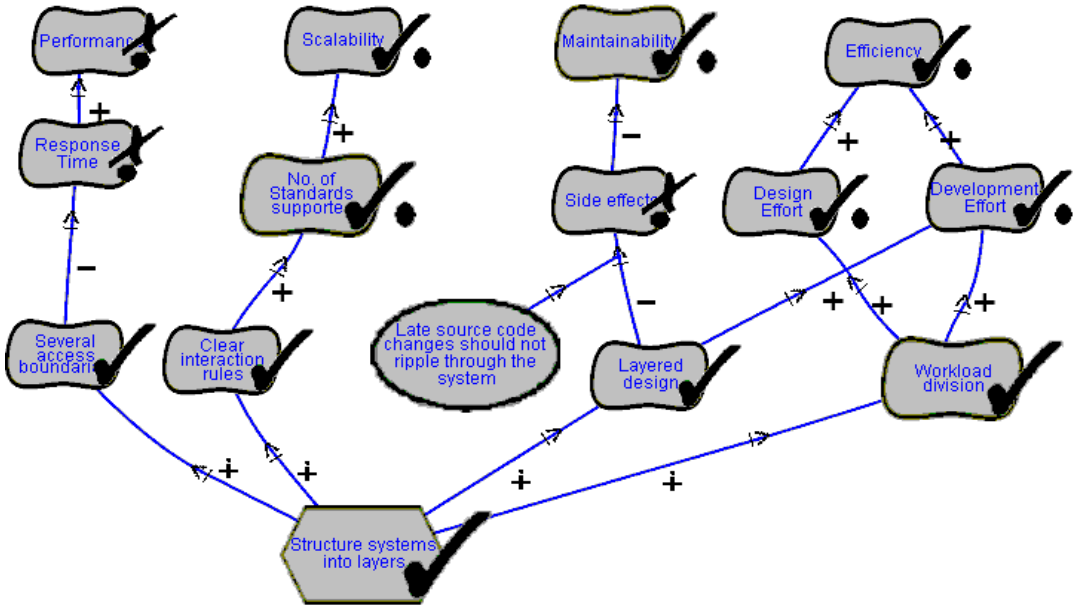



**Fig. 31.** Force hierarchy for the Layers pattern

---

**Microkernel.** The Microkernel pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts.

---

**Problem.** Developing software for an application domain that needs to cope with a broad spectrum of similar standards and technologies is a non-trivial task. The following forces need particular consideration when designing such systems:

- The application platform must cope with continuous hardware and software evolution.
- The application platform should be portable, extensible and adaptable to allow easy integration of emerging technologies.

**Solution.** Encapsulate the fundamental services of the application in a microkernel component. The microkernel includes functionality that enables other components running in separate processes to communicate with each other.

Core functionality that cannot be implemented within the microkernel should be separated in internal servers. External servers implement their own view of the underlying microkernel. To construct this view, they use the mechanisms available through the interfaces of the microkernel.

Clients communicate with external servers by using the communication facilities provided/stated by the microkernel approach.

**NFR-based representation.** See Figure 32 for the force hierarchy corresponding to the Microkernel pattern.

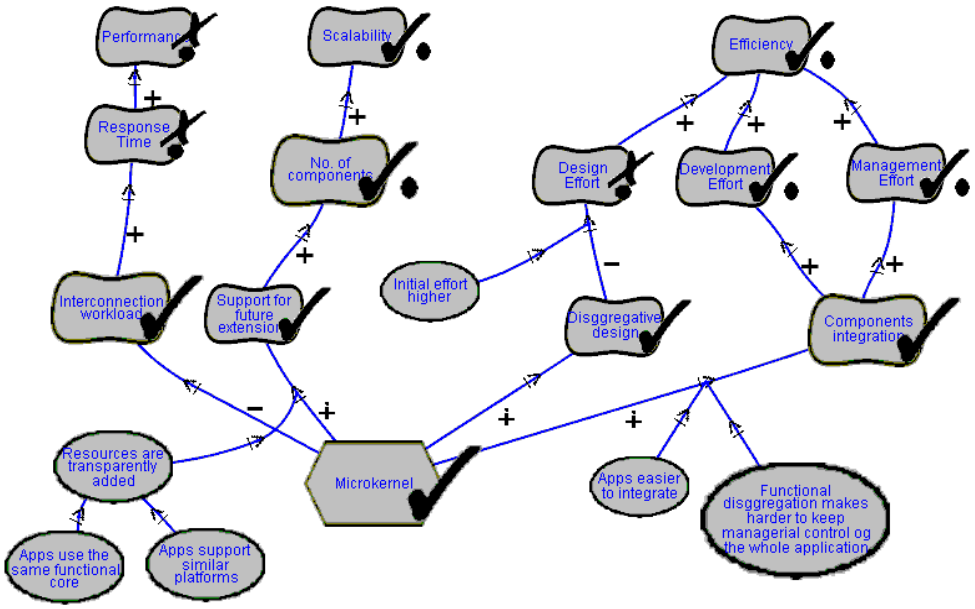



**Fig. 32.** Force hierarchy for the Microkernel pattern

**Slender Cyclic Functions.** This pattern is concerned with processing that must execute at regular intervals, such as generating reports or periodic data archiving.

**Problem.** Certain types of processing require the execution of a number of functions at regular intervals (these functions are referred to as cyclic). The problem arises when there are concurrent sources of a given event, or when there is other processing to be performed. If the deadline for handling each event is to be met, there must be sufficient slack time to allow the processing of the cyclic functions and the other events, as well as any other work that must be performed.

The example of the service center illustrates a situation where this pattern can be applied. As explained in the description of the Flex-Time pattern, a service center system collects data on activities in a service system and summarizes them in a memory (RAM) data bank. The data collected have to be periodically copied into an archival database. The processing is extensive because it is not a one-to-one copy; instead, it requires loading the data into multiple database tables and updating keys with each update. While this archive cycle executes, any other event that requires processing might clog the system capacity, or unforeseen problems might make difficult to complete the archiving on time.

**Solution.** Identify the functions that execute repeatedly at regular, specific time intervals, and minimize their processing requirements.

For the example, at the service center, we need to streamline the extensive processing required to copy the memory data bank to the archival database. One way of reducing the processing is to take a quick snapshot of the real memory contents, copy them to another location in virtual memory, and then allow other work to proceed while the copy of the memory contents is used for updating the database. Making a simple one-to-one copy takes far less time than the whole database update.

As a result of the application of this pattern, the overall processing time is increased because both the one-to-one copy and the database updates are made. However, the net effect is the reduction of the time that inbound work must wait.

**NFR-based representation.** Figure 33 shows the corresponding force hierarchy for the Slender Cyclic Functions pattern.

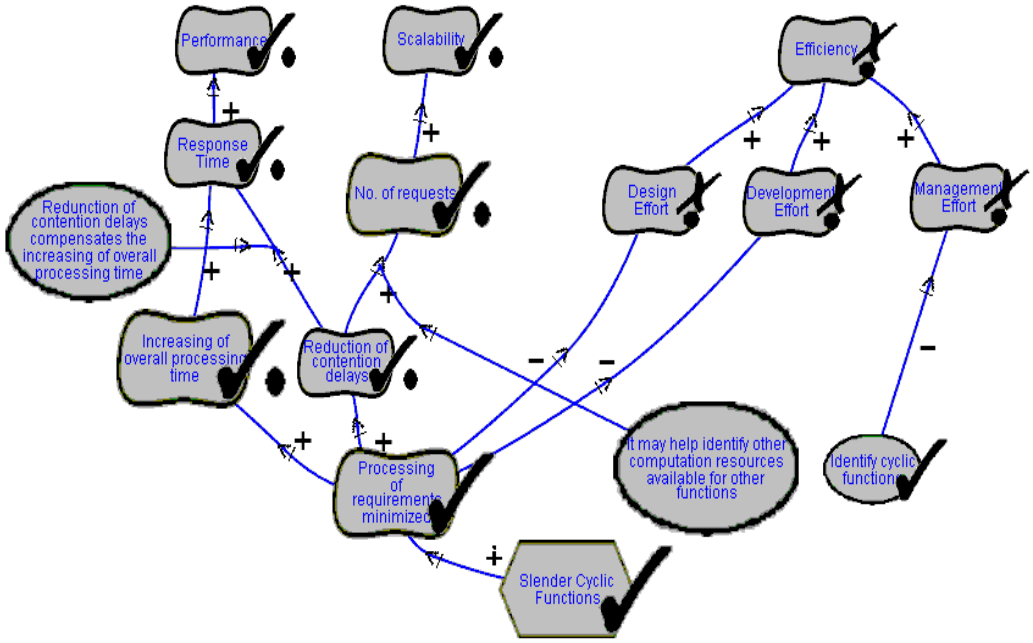



**Fig. 33.** Force hierarchy for the Slender Cyclic Functions pattern

## References


1. Alexander C.: *A Pattern Language.* Oxford University Press. New York, NY (1977).
2. Alexander C.: *The Timeless Way of Building.* Oxford University Press. New York, NY (1979).
3. Bayer M.: *CTI Solutions and Systems: How to Put Computer Telephony Integration to Work.* McGraw-Hill Companies Inc. (1997).
4. Borchers J.: *A Pattern Approach to Interaction Design.* , John Wiley & Sons, Ltd., (2001).
5. Bosch J., Molin P.: *Software Architecture Design Evaluation Transformation.* In Proceedings of the IEEE Engineering of Computer Based Systems Symposium (ECBS99), (1999).
6. Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley & Sons, Ltd., (1996).
7. Chung L., Nixon B., Yu E., Mylopoulos J.: *Non-Functional Requirements in Software Engineering.* Kluwer Academic Publishers. (2000).
8. Clark R., Jensen E. D., Kanevsky A., Maurer J., Wallace P., Wheeler T., Zhang Y., Wells D., Lawrence T., Hurley P.: *An Adaptive, Distributed Airborne Tracking System.* In Proceedings of the International Workshop on Parallel and Distributed Real-Time Systems, (1999).
9. Coplien J., Schmidt D. (editors): *Pattern Languages of Program Design.* Addison-Wesley, (1995).
10. Coplien J.: *A Generative Development Process Pattern Language.* In: Coplien J., Schmidt D. (editors). Pattern Languages of Program Design. Addison-Wesley, (1995).
11. Deugo D., Weiss M., Kendall E.: *Reusable Patterns for Agent Coordination.* In: Omicini A. (editor). Coordination of Internet Agents, Springer, (2001).
12. Erickson T.: *Lingua Francas for Design: Sacred Places and Pattern Languages.* In Proceedings of Designing Interactive Systems (DIS 2000), ACM, Brooklyn, NY, August, (2000).
13. Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, (1994).
14. Gross D., Yu E.: *From Non-Functional Requirements to Design through Patterns.* Faculty of Information Studies, University of Toronto, Toronto, Ontario, Canada, (2001).

15. Gullichsen E., Chang E.: *Generative Design in Architecture Using an Expert System.* In Proceedings of the Graphics Interface Conference. 425–433, (1985).
16. Harrison N., Foote B., Rohnert H. (editors): *Pattern Languages of Program Design 4.* Addison-Wesley, (2000).
17. Frye J., Yoder J. (editors): *The Hillside Group: The Patterns Home Page.* URL: http://hillside.net/patterns
18. Keshav S.: *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network.* Addison-Wesley, Reading, Massachusetts, USA (1998).
19. Martin R., Riehle D., Buschmann F. (editors): *Pattern Languages of Program Design 3.* Addison-Wesley, (1998).
20. McPhail J., Deugo D.: *Deciding on a Pattern.* 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2001, LNCS 2070, Springer, (2001).
21. Molin P., Ohlsson L.: *Points & Deviations - A Pattern Language for Fire Alarm Systems.* published in Pattern Languages of Program Design 3, Addison-Wesley, (1998).
22. Rogers G. F.: *Framework-Based Software Development in C++* Prentice-Hall Inc., Middletown, New Jersey (1997).
23. Schmidt D., Stal M., Rohnert H., Buschmann F.: *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* John Wiley & Sons, Ltd., (2000).
24. Sevcik P., Forbath T.: *The Call Center Revolution. Converged Networks Enable Integrated Customer Service Solutions.* Northeast Consulting. Boston, MA. (1998).
25. Simon H.: *Models of Bounded Rationality.* Vol. 2. Cambridge, MIT Press, (1982).
26. Smith C., Williams L.: *Performance Solutions. A Practical Guide to Creating Responsive, Scalable Software.* Addison-Wesley (2002).
27. Sun Microsystems, Inc.: *Java Remote Method Invocation - Distributed Computing for Java.* URL: http://java.sun.com/marketing/collateral/javarmi.html
28. Vlissides J., Coplien J., Kerth N. (editors): *Pattern Languages of Program Design 2.* Addison-Wesley, (1996).
29. Yu E., Liu L. *GRL - Goal-oriented Requirement Language, Project Home Page.* URL: http://www.cs.toronto.edu/km/ome. (2001)