INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

A DESIGN PROCESS BASED ON PATTERNS AND NON-FUNCTIONAL REQUIREMENTS

By
Iván J. Araujo

A thesis
submitted to the School of Computer Science
and the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

March 15, 2002

© Copyright 2002, Iván J. Araujo

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**385 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**385, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72077-2

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis.

A Design Process Based on Patterns and Non-Functional
Requirements

submitted by

Iván J. Araujo

in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Michael Weiss. Ph.D.
(Thesis Supervisor)

Frank Dehne. Ph.D.
(Director. School of Computer Science)

Carleton University
March 15, 2002

Abstract

This thesis introduces a design process that combines the top-down analysis of non-functional requirements with the bottom-up application of patterns, reusable solutions to common design problems.

These two approaches are intimately tied together. A formalization of patterns using the non-functional requirement (NFR) framework leads to a richer, more structured way of expression for patterns, so that both their applicability and interrelationships become clearer.

Hence, the design process presented in this work serves as an initial approach to take advantage of both methods. Based on a new representation for patterns, requirements are iteratively considered as appropriate patterns are selected and applied to design. Thus, software architecture results from the fulfillment of both functional and non-functional requirements, as best practices (patterns) are applied.

Moreover, by identifying possible relationships among patterns, the proposed design approach also provides support for devising pattern languages.

Acknowledgements

The ideas presented in this thesis are the result of the combination of previous efforts on patterns and the NFR framework.

Herr Dr. Michael Weiss war ausserordentlich unterstützend in der Konzeptgebung und Entwicklung dieser Arbeit. Seine Leitung war entscheidend für das Verständnis aller fundamentalen Themen und den Erfolg des Ganzen.

It is also important to highlight the contribution of Mitel Networks (in particular, Dr. Gunter Mussbacher) in coming up with an interesting topic for the case study. Dr. Mussbacher was very instrumental in providing the call center documentation and clarifying aspects inherent to it.

Deep appreciation is expressed to Ildemaro Araujo, M.C.S., for his role in reviewing this thesis during final preparations. His observations were very helpful in enhancing the readability and understanding of the issues included in this work.

With utmost gratitude this thesis is dedicated to Mercedes de Araujo, my mother. Without her support, her encouragement, her love this work would not have been a reality.

Iván J. Araujo

Contents

Abstract	iii
Acknowledgements	iv
1 Introduction	1
1.1 Motivation	2
1.2 Thesis Goal	3
1.3 Objectives	3
1.4 Contributions	4
1.5 Scope	6
1.6 Thesis Structure	6
1.7 Notation	8
2 Background	9
2.1 Systems: The Result of the Integration of Several Elements	9

2.2	Non-Functional Requirements: Nature and Significance	12
2.3	Patterns: A Universe of Reusable Solutions	14
2.4	Non-Functional Requirements and Patterns: Two Linked Approaches	17
2.5	Related Work	18
2.6	Chapter Summary	19
3	The NFR Framework	20
3.1	Using the NFR Framework	20
3.2	The NFR Framework: A Goal-Driven Process	28
3.3	Chapter Summary	29
4	Patterns	31
4.1	Patterns and Pattern Languages	31
4.2	Patterns and Frameworks	33
4.3	Pattern Languages: A Lingua Franca for Design	33
4.4	Representing a Pattern	34
4.4.1	The Alexandrian Form	35
4.4.2	From Architecture to Software Engineering: Expanding the Alexandrian Form	37
4.5	Organizing and Selecting Patterns	38
4.5.1	Pattern Catalogs	38

4.5.2	Pattern Catalog Usefulness	40
4.5.3	Pattern Decision Aid and the Need for a New Approach	41
4.6	Patterns: A Solution-Driven Approach	42
4.7	Chapter Summary	43
5	A New Design Process	44
5.1	Combining Patterns and Non-Functional Requirements	45
5.1.1	Forces and Trade-offs	45
5.1.2	Force Hierarchies: NFR-Based Representations for Patterns .	48
5.2	Defining a New Design Process	53
5.2.1	Approaching the New Design Process	53
5.2.2	The Design Process Steps	54
5.3	Chapter Summary	57
6	Case Study: A Call Center Architecture	59
6.1	Analyzing and Designing a Call Center	59
6.2	Applying the Proposed Design Approach	61
6.2.1	Outlining the Application of the Design Process	62
6.2.2	Iterating on Steps 3 and 4 of the Design Process	64
6.2.3	Instantiating Patterns	67

6.3	Devising a Pattern Language	74
6.4	Chapter Summary	77
7	Conclusion	78
7.1	Review of Objectives	79
7.2	Summary of Contributions	79
7.3	Potential Benefits and Applicability	81
7.4	Future Directions	82
A	A Catalog of Patterns for Call Center Design	84
A.1	Pattern Principles	84
A.2	Catalog of Patterns	87
B	Case Study: Conceptual Definition	119
B.1	Call Center Functionality	119
B.2	Non-Functional Requirements	124
B.3	Force Hierarchy for the Application Domain	126
C	CTI Terminology	130
	Bibliography	134

List of Tables

A.1	Patterns contained in the catalog and their corresponding source . . .	87
B.1	System-level requirements	121
B.2	Agent Desktop requirements	122
B.3	Computer-Telephony integration requirements	122
B.4	System administration requirements	123

List of Figures

2.1	Software patterns are reusable solutions to recurring design problems	15
3.1	An example of an NFR framework representation	21
3.2	Distinction among contribution types (adapted from [12])	22
3.3	Two alternatives in considering the number of tellers of a bank	27
3.4	Non-functional requirements influence and guide the development process (adapted from [12])	29
5.1	An example of a force hierarchy, showing the base force hierarchy and the pattern solution force hierarchy	50
5.2	The force hierarchy representation for the Deviation pattern	52
5.3	Requirement analysis sheet	54
5.4	Steps of the new design process	55
6.1	A basic structure for a call center application (adapted from [35]) . .	60
6.2	Call centers viewed as embedded real-time systems (adapted from [39])	61

6.3	Validating the results of pattern instantiations	64
6.4	Considering the overall system architecture	64
6.5	Force hierarchy for the Microkernel pattern	65
6.6	Considering the Call Control API for switch – application interaction	66
6.7	Force hierarchy for the Indirection/Binding pattern	67
6.8	Considering the architecture of agent desktop applications	68
6.9	Force hierarchy for the Broker pattern	69
6.10	Microkernel pattern instantiation for the call center	70
6.11	Indirection/Binding pattern instantiation for the call center	71
6.12	Broker pattern instantiation for the call center	72
6.13	Structure of the call center application after incorporating patterns into design (three iterations)	73
6.14	An overview of the call center structure (actual design). Components partially addressed in the case study are shaded	74
6.15	An initial attempt to devise a pattern language	75
6.16	Completing the pattern language with known relationships (patterns added are shaded)	76
A.1	Force hierarchy for the Alternate Route pattern	89
A.2	Force hierarchy for the Balancing pattern	91
A.3	Force hierarchy for the Batching pattern	93

A.4	Force hierarchy for the Binding/Indirection pattern	95
A.5	Force hierarchy for the Blackboard pattern	97
A.6	Force hierarchy for the Broker pattern	100
A.7	Force hierarchy for the Coupling pattern	102
A.8	Force hierarchy for the Deviation pattern	105
A.9	Force hierarchy for the Fast Path pattern	107
A.10	Force hierarchy for the First Things First pattern	109
A.11	Force hierarchy for the Flex Time pattern	111
A.12	Force hierarchy for the Layers pattern	113
A.13	Force hierarchy for the Microkernel pattern	115
A.14	Force hierarchy for the Slender Cyclic Functions pattern	118
B.1	Force hierarchy for Total Satisfaction	127
B.2	Force hierarchy for Performance	127
B.3	Force hierarchy for Scalability	127
B.4	Force hierarchy for Efficiency	128
B.5	Force hierarchy for Security	128
B.6	Force hierarchy for Maintainability	128
B.7	Force hierarchy for Reliability	129
B.8	Force hierarchy for Transparency	129

Chapter 1

Introduction

Every system is intended to fulfill a need. In Software Engineering, that need is usually expressed in terms of *requirements*. In general, the most visible aspect of a system is the functionality it provides, i.e., its *functional requirements*. However, a system is only fully and comprehensively described if we also include in our analysis features that are not covered by a functional conceptualization. These aspects are referred to as *non-functional requirements*.

Non-functional requirements, also known as *quality requirements* [12], are at the heart of design trade-offs when a designer must choose among several competing alternatives implementing the same functionality. Typical non-functional requirements are reliability, usability, maintainability, and so on.

Patterns have received significant attention in the software design field. This concept offers a body of empirical solution templates whose usefulness has been previously shown and that can be used to explore new problems in design efforts. Successful application of patterns is intimately tied to the fulfillment of both functional and non-functional requirements.

1.1 Motivation

Addressing non-functional requirements during design had not received extensive consideration until recent years. The *non-functional requirements (NFR) framework* [12] is a significant step towards making explicit the relationships between non-functional requirements and intended design decisions.

On the other hand, a very important issue related to system design is how to express a solution so that anybody can easily grasp the idea behind it, and reapply its principles to new problems. In [17], Erickson points out the need for a common language: A language that allows all people involved in analysis and design (current and/or potential users, as well as any other person possibly implicated in the project) to actively participate in all analysis and design stages without missing fundamental details. By “fundamental” is meant letting people with no background in system development come up with their own ideas. At the same time, that language should provide an adequate platform for describing and communicating solutions. Patterns seem to fulfill this need.

In Software Engineering, representing and selecting patterns remains an empirical study. On the one hand, *pattern representation* essentially follows the format originally proposed by Christopher Alexander [2]. Although this format has the advantage of preserving the *generative nature* of patterns, it is still difficult to recognize when to apply them. On the other hand, a *pattern user* has to face a broad and growing diversity of choices when selecting a pattern as a design decision is taken.

In [29], McPhail presents the *Pattern Decision Aid*. This approach suggests how to select a pattern based on a ranking of forces. The way the ranking is built is based on the notion of *force hierarchy*. This approach facilitates the selection of patterns while forces, an inherent component of patterns, determine the most satisfactory criteria for designers.

Hence, with the aid of the NFR framework we may formalize the representation of

patterns by using the notion of force hierarchy. In this way, the existing gap between the ‘classical’ pattern representation and the notion of hierarchy of forces is filled. As a consequence, we have a means for supporting *pattern selection*.

1.2 Thesis Goal

The overall goal of this thesis is to propose a *design process* based on the addressing of *functional* and *non-functional requirements*, as well as *patterns* through a new representation based upon the NFR framework.

The main idea is to apply patterns as non-functional and functional requirements are considered. The proposed design approach differs from others in that non-functional requirements associated with the functional issue addressed determine the pattern selection through the use of a *new pattern representation*. As a result of the application of this new design process, not only a *system architecture* may be obtained, but also possible new *pattern interrelationships* may be explored, and consequently, a *pattern language* may be derived.

1.3 Objectives

The objectives of this thesis are focused on two areas: *Non-functional requirements* and *patterns*.

Non-Functional Requirements. This thesis elaborates upon the NFR framework, an approach proposed by Chung et al. in [12], as well as upon its applicability to pattern representation initially devised by Gross and Yu [20]. The NFR framework represents a significant step in formalizing non-functional requirements and in addressing their achievement during design. This thesis is aimed to provide a *design*

approach centered on a new pattern representation based on the NFR framework. This new design process is intended to carefully consider the non-functional requirements inherent to a given problem, and to address their fulfillment.

Patterns. Non-functional requirements are intimately tied to the notion of forces in patterns. This work is intended to provide a *new representation for patterns*. This representation will emphasize the forces and other elements involved in a pattern solution under a given context, or application domain. It must also offer support for selecting and applying patterns, as well as for exploring possible pattern interrelationships.

1.4 Contributions

We can point out the following issues as the contributions of this work:

- It provides a means for improving the understanding of patterns. A typical pattern description may consist of several pages of dense explanations about the intricacies of the solution presented. However, to get the picture several readings are often needed. How about counting on a means that allows pattern readers to catch the idea behind the solution that a given pattern offers? Through the use of the NFR framework, it is possible. This work elaborates on this important issue through the development of a new and complementary representation for patterns.
- It promotes the use of patterns. By their own merit patterns have become part of a very important trend in Software Engineering. One of the most promising aspects of patterns is the possibility that a solution found in a given field be also applicable in a very different domain. The approach presented in this work promotes the more extensive use of patterns through making the understanding of pattern contributions easier and clearer for pattern readers.

- It provides a consistent platform for ongoing refinement of pattern descriptions. The approach proposed in this work allows designers to count on a consistent way of representing patterns. It also provides support for understanding possible side effects, identifying ‘hidden’ trade-offs, and clearly establishing the impact of certain design decisions.
- It supports the understanding of patterns’ portability across application domains. A typical situation observed in Software Engineering is that a given pattern may receive several names and be represented in different ways across diverse analyses. The consistency that our pattern representation provides also helps reduce the confusion that all those different descriptions may generate.
- It proposes a design process. In terms of software design, the most significant contribution of this work is the outline of a design process entrenched in functional and non-functional requirements, and patterns. The way these three factors are addressed during design, with emphasis on non-functional requirements, may help obtain a satisfactory software architecture in terms of the elements involved in system development.
- It offers support for devising pattern languages. With the support of the new pattern representation, the way several patterns may be interrelated through common contributions and trade-offs can be considered. After addressing their instantiation, a more complete view of pattern interrelationships may be derived and it may be possible to devise new pattern languages, or to understand new connections among known pattern languages.
- It provides a pattern representation that may serve as a support to use the Pattern Decision Aid. With a new pattern representation built on the notion of force hierarchy, a direct connection can be established with the Pattern Decision Aid. Hence, we may count on a way of expressing patterns that can be used to address their selection with the advantage of considering the large spectrum that the Pattern Decision Aid supports.

1.5 Scope

It is important to stress that the main focus of this work is to propose a new design process. Although a new pattern representation is presented, its purpose is to serve as an auxiliary or complementary description. It is not intended to replace either the one proposed by Alexander, or any other format based on Alexander's ideas. The pattern format presented in this thesis helps pattern selection while it also leverages the generative nature of patterns.

In the case study, the idea of re-examining the architecture of a call center is not to produce a perfect (or even a better) design, but just to intuitively show how the design process may be used. This has two direct consequences. On the one hand, a preliminary system architecture for a distributed application may be obtained. On the other hand, new interrelationships among patterns may become apparent, which brings out the opportunity of creating a pattern language.

Specific aspects related to design and implementation of call center systems have been used to illustrate the application of the proposed design process. Although a number of issues related to call center design had to be considered, a definitive structure for such an application requires more time and effort than that employed in developing this thesis and is beyond its scope.

Since ongoing refinement is at the core of the proposed design process, all the pattern representations obtained, along with the interrelationships found are not definitive in any sense. They are subject to further revision and improvement as part of the design process.

1.6 Thesis Structure

This thesis is structured as follows:

- **Fundamentals.** Chapter 1 initiates this work by presenting the main issues covered in the thesis body. Chapter 2 provides an overview of associated material relevant to the conceptual framework of our proposed approach, such as basic concepts related to system design, non-functional requirements, and patterns.
- **Conceptual framework.** This work depends on two fields: The NFR framework [12, 20], and patterns [11, 19]. Accordingly, chapters 3 and 4 describe the topics concerning the NFR framework and patterns, respectively, in more detail. All of this serves as an appropriate foundation for the proposed methodology.
- **The new design approach.** Chapter 5 elaborates on the proposed design process. It opens with a discussion of the importance of forces and trade-offs in patterns. Later on, an essential notion in the formulation of the new design approach is presented: the force hierarchy. Moreover, roles in pattern application are redefined, and the artifacts required for the application of the new methodology are shown. This chapter concludes with a discussion of the steps that constitute the proposed design approach.

Chapter 6 presents a case study to illustrate the application of the new design process. The architecture of a call center is examined and three patterns are applied as functional issues are considered. As it corresponds to the basis of this design process, the criteria for selecting patterns are based on their new NFR-based representation.

- **Concluding remarks.** Chapter 7 summarizes how the thesis meets the stated objectives and confirms the claimed contributions. The thesis finishes with a brief description of related initiatives, as well as a statement of potential areas of applicability and suggested areas for future work.
- **Supplementary material.** Accompanying material that complements some issues presented in this work is contained in three appendices.

1.7 Notation

The Organization Modeling Environment (OME) tool [45] was used to portray the force hierarchies using the NFR framework notation [12]. OME is very instrumental in helping propagate the degree of achieving soft-goals through the contribution links using the qualitative reasoning process described in the NFR framework.

The descriptions corresponding to pattern instantiations use the Unified Modeling Language (UML) [8].

Chapter 2

Background

Under the perspective of Software Engineering, there are several concepts involved in system design. This chapter outlines some of these essential subjects and provides a conceptual framework for the topics presented in this thesis.

2.1 Systems: The Result of the Integration of Several Elements

Certainly, any system is the consequence of putting together several elements and the result of their interaction. The way those elements are integrated is not random. System design responds to the analysis of the availability of *resources*, such as money, time, space, processing capacity, and so forth; as well as *constraints* on those resources. A main goal of system design is to optimize the way the constraints are handled to fulfill the *requirements* established by user needs. Starting with the concept of *system requirement*, let us establish a conceptual definition of several issues relevant to system design.

Requirements: A *requirement* is a statement, usually expressed in natural language, of the services a system is intended to provide. Requirements are translated from the abstract to the concrete world through *specifications*. Specifications are organized into a structured document setting out detailed descriptions of the system services. In turn, specifications can be broken down into several levels of abstraction. Each level of abstraction serves as a basis for a design or implementation. Regardless of their level of abstraction, requirements can be classified into two categories: *functional* and *non-functional* [10].

Functional requirements: *Functional requirements* are those directly related to the domain functionality of a system, i.e., specific functions, tasks or behaviors the system must support. In [41], a functional requirement is defined as follows:

a system/software requirement that specifies a function that a system/software system or system/software component must be capable of performing. Functional requirements are software requirements that define behaviors of the system, that is, the fundamental processes or transformations that software and hardware components of the system perform on inputs to produce outputs.

Functional requirements may be directly visible to the users of an application by means of a particular function, or they may represent aspects of its implementation, such as the algorithm used to compute a function [11]. A component, or group of software components, implements functional requirements.

Non-functional requirements: *Non-functional requirements* tend to be stated in terms of constraints on the results of components satisfying functional requirements. In [41], non-functional requirements are defined as follows:

software requirements that describe not what the software will do.

but how the software will do it; for example, software performance requirements, software external interface requirements, software design constraints, and software quality attributes.

In analyzing the non-functional requirements of a given piece of software the following issues usually arise: how fast, how efficiently, how safely, and so on, a particular task is performed; how flexible an architecture is; how usable a software seems to be; and so forth.

System component: A *system component* is an encapsulated part of a software system. A component is formally described in terms of two parts [11]:

- An interface part that defines the functionality provided by the component and specifies how to use it.
- An implementation part that includes the actual code for the functionality provided by the component.

Components serve as the building blocks for the architecture of a system.

System architecture: The architecture of a system is defined as the way the parts work together to make the whole [7]. More specifically, a system architecture is a description of a system and the relationships among its components [11].

The architecture of a system may also be defined as the *structure* or structures of the system, which comprise components, the externally visible properties of those components, and the relationships among them [4].

Defining *system architecture* in this way has several implications:

- Architecture defines components. The architecture embodies information about how the components interact with each other.

- Every system has an architecture, because every system can be described in terms of several components and the relations among them.
- Systems can comprise more than one architecture.

Framework: A *framework* is a partially complete software that implement generic solutions by providing common facilities to different applications [27]. It defines the architecture for a family of sub-systems and provides the basic building blocks to create them. It also establishes the places where adaptations for specific functionality should be made [11].

Design decisions: Any design or development involves decision making. Decisions depend on the availability of certain resources and are forced by some constraints, such as market trends. Some decisions are taken after the completion of the system development, and they may involve identifying current and potential problems, requiring the modification of the system, such as in performance tuning [39] or in refactoring [18].

2.2 Non-Functional Requirements: Nature and Significance

A *non-functional requirement* denotes a feature of a system that is not covered by its functional description. Hence, non-functional requirements are derived as a complementary side of the functionality already defined. It is therefore useful to begin with the functional requirement statement to comprehensively describe the non-functional requirements attached to a project.

However, almost all the designer's attention is typically given to functional requirements. On the other hand, it is often seen that research communities give attention

merely to single non-functional requirements. The consequence of these “omissions” is that, only at a later design phase, a non-functional requirement may become apparent, with an undesirable effect upon the project. Let us illustrate this claim with two imaginary cases:

Software migration. Software migration from DOS to Windows came to be a critical issue for several software companies in the early nineties. A very well-known software company (let us call it XYZ) produced a fine word-processor application that enjoyed the dominance of the DOS market over several years in the late eighties (at that time, DOS seemed to be the only operating system for most computer users). When time to consider the migration to Windows arrived, XYZ decided to preserve in the migrated version all the GUI features that the DOS version had. The reason? Its designers had concluded that users preferred their word-processor because of its powerful features (by no means did they realize that many of those ‘powerful’ features were never used by most users). They also considered that keeping the GUI features would facilitate the migration, and in this way the *time-to-market* constraint would be easily satisfied.

Despite an initial market hit, progressively more and more users modified their preference and things quickly became different for this organization. What caused the change? Users found the new version of the word-processor too difficult to use in a Windows environment. A different word-processor was also available, with fewer features but able to fully exploit the Windows GUI (and also easier to use). The result: XYZ lost the race for the Windows word-processor market.

An e-mail system. John belongs to the technical staff of a large high-tech corporation with headquarters in Canada and several branches worldwide. Mainly by e-mail, he gives support to a group of important customers in Vancouver. On January 25, John, whose office is located

in Ottawa, is notified that effective February 5, he is to be temporarily transferred to the corporation branch in Buffalo, New York. As part of his duties, he is required to continue to support the group of customers in Vancouver.

Having completed all his personal arrangements, John moves to Buffalo and after settling into his new office he discovers that he cannot get access to his e-mail account from his new location. The Network Administrator in Buffalo only offers him a ‘solution’: creating for him a new e-mail account located on a different domain. Needless to say, this means an entirely new e-mail address for John. Thus, he will have to use a ‘temporary’ e-mail account as he stays in his ‘also’ temporary position. Moreover, he has to notify his customers in Vancouver immediately that he cannot be reached at his regular e-mail address (while he stays in Buffalo).

These two examples illustrate how non-functional requirements may have a determinant impact on software applications. On the one hand, *usability* determined the market predominance of a word-processor. On the other hand, the lack of *flexibility* of an e-mail system had a undesirable effect on the support service.

Naturally, the impact of ignoring the influence of non-functional requirements is variable and depends mainly on the nature of the application. Hence, it is important to carefully consider non-functional requirements, and analyze their impact on the software architecture.

2.3 Patterns: A Universe of Reusable Solutions

Defined as “something designed or used as a model for making things” by Webster’s Collegiate Dictionary [43], the term “pattern” takes a new dimension in Alexander’s work. He defines a *pattern* as a three-part rule, which expresses a relation among a certain *context*, a *problem*, and a *solution* [2]:

- *Context* or circumstances in which the design problem is being solved. This may be considered a set of preconditions which specify the applicability of a pattern.
- The system of often contradictory design considerations or forces that define the *problem* solved by a given pattern. This provides the reasons for using a pattern.
- *Solution* that resolves this system of forces. As patterns are linked in a pattern language, a solution usually involves the contributions of other patterns.

Thus, problems specify conflicting forces, and solutions resolve the conflict of forces while context indicates priorities or preferences of the designer. Figure 2.1 (based on [30]) represents the relationship among the elements implied in using a pattern.

Figure 2.1: Software patterns are reusable solutions to recurring design problems

In Software Engineering, the importance of applying patterns is based on the following [7]:

- Patterns are recurring decisions made by experts, expressed in such a way that those less skilled are able to understand them and use them easily.

- Patterns describe the *why* of design.
- Patterns provide a language for designers that makes it easier to plan, talk about, and document designs.

Moreover, patterns provide a common language that designers can use in interacting with users at any stage of design and development [17]. By using a common language, users can express more candidly their needs and expectations; and designers are able to improve their comprehension of system requirements.

There are two roles involved in applying patterns:

- *Pattern writer*: person (or team) who identifies a recurring solution. S/he documents the context of the problem, the forces involved; and provides a rationale for the solution. S/he may also identify the possible interrelationships between a new pattern and other existing patterns. Thus a new pattern may be integrated into an existing pattern language, or a new pattern language may be developed.
- *Pattern user*: person interested in selecting and applying patterns to design. S/he is usually interested in using pattern languages, so as to come up with a new solution resulting from the integration of several patterns.

It is important to state that a pattern does not provide a ready-made solution. It just describes a solution for a problem in very general terms. When a pattern is applied, it has to be adapted to the concrete circumstances of design.

2.4 Non-Functional Requirements and Patterns: Two Linked Approaches

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.

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 *NFR framework* [12]. This trend tends to be open to the specialization of a particular software project's statement of requirements. The approach proposed for making associated design decisions entails a pattern user initiating a search of design goals with patterns addressing these goals retrieved for possible selection. Based on the manner a selected pattern is incorporated in the resulting context, subsequent searches may be made for other sets of applicable patterns.

Treating functional and non-functional requirements as goals allows exploring and retaining a design history of alternative design choices, their rationales and the reasons why they were accepted or rejected during design. Non-functional requirements relate to each other and to the alternative solutions discussed in patterns. Non-functional requirements are the criteria for evaluating why to accept or reject a given solution.

In [20], Gross and Yu propose a systematic approach to organizing, analyzing, and refining non-functional requirements. This provides much support for the structuring, understanding, and application of patterns during design. They explore how the way non-functional requirements are represented in patterns aids in better understanding the rationales of design, and makes patterns more amenable to analysis and structuring.

Gross and Yu use their approach to represent, analyze and then apply design

patterns during design. They extract non-functional requirements soft-goals and their contributions on each other from the problem section of a pattern. Patterns are applied by relating operationalizations in the goal graph to the functional elaboration of the system under development. However, since a soft-goal hierarchy is developed for each pattern, presumably from scratch, rather than for a project or domain, the scalability of the pattern description is limited. In its present form, the approach does not take advantage either of the known morphological links among patterns in a pattern language.

2.5 Related Work

Besides the initiative of Gross and Yu [20], other efforts to treating patterns more formally are the following:

In [22], Gullichsen and Chang provide probably the earliest attempt at formalizing patterns. They designed a Prolog-based expert system that implements Alexander's original pattern language to produce three-dimensional views of architectural forms. The authors define patterns as morphological laws that define a permissible set of structural arrangement of the elements of architectural forms within a given context. For each pattern, the formal representation contains the pattern name, and a judgment of its validity. 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 [9]. 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, the 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 a tool is the *Pattern Editing Tool (PET)* developed by the author.

2.6 Chapter Summary

In this chapter we have reviewed some topics relevant to the design approach proposed in this work. This comprises the following concepts: *requirements*, *functional* and *non-functional requirements*, *system components*, *system architecture*, and *framework*.

Likewise, the importance of *non-functional requirements* was stressed. It was pointed out why the analysis of non-functional requirements should not be omitted during design. We will go in depth on this subject when considering the NFR framework in the following chapter.

Definition, importance, and roles involved in the use of the *pattern* approach were also covered here. This served as a preliminary conceptualization of this crucial issue. We will go back to this topic when covering patterns and pattern interrelationships in Chapter 4.

We also explored how patterns and non-functional requirements may be linked through the use of the NFR framework. This notion establishes the foundation for the new pattern representation introduced in this work. Thus, the following two chapters will cover the NFR framework and patterns in more detail to provide the conceptual framework necessary for conceiving the new pattern format and the accompanying design process.

Chapter 3

The NFR Framework

As mentioned earlier, the NFR framework [12] provides one of the conceptual basis for this work. In this chapter we will review several aspects from the NFR framework relevant to the formulation and application of the proposed design process presented in this thesis.

The NFR framework uses non-functional requirements like interoperability, security, performance and others to manage the overall design process. Non-functional requirements are represented as *soft-goals* [12]. Soft-goals are analyzed and decomposed so that implementation possibilities are initially devised. The entire representation of soft-goals plus their decomposition is known as a *soft-goal graph* [12]. The construction, elaboration, analysis, and revision of a soft-goal graph comprise the whole application of the NFR approach.

3.1 Using the NFR Framework

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*. Thin-line clouds are used to represent soft-goals and sub-goals, and lines connecting them denote interdependency links. At some level, when a soft-goal is sufficiently refined, *operationalizations* come up. Represented by thick-line clouds, *operationalizations* are possible development techniques for achieving the corresponding non-functional requirements. By enumerating all the possible operationalizations, the developer may choose specific solutions for the target system. Thus, a soft-goal graph registers all the developer's consideration of soft-goals, and shows the interdependencies among them.

Soft-goals are incrementally refined, and *trade-offs* are made as conflicts and synergies are discovered. Thus, the operation of the NFR framework is the result of the incremental and interactive construction, elaboration, analysis, and revision of a soft-goal graph. Figure 3.1 illustrates this notation through an example.

Figure 3.1: An example of an NFR framework representation

Soft-goals, which are “soft” in nature, are represented by solid-line clouds. Operationalizations that link soft-goals to a particular design are shown as clouds with a

thick solid border. The solid links between soft-goals indicate contributions between them. The notation defines different types of contributions: *Break*, *Hurt*, *Some-*, *Help*, *Make*, and *Some+* (see Figure 3.2).

Figure 3.2: Distinction among contribution types (adapted from [12])

- *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 as described in [12]. Other elements of the notation are *claims* and *correlation links*. *Claims* justify the choice of contribution type on a link, and are shown as dotted-line clouds. *Correlation links* indicate side effects that are not at the center of the analysis corresponding to this thesis. They are shown as dotted-line links.

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 [37]. Thus, we

may think of “satisficing” soft-goals as *being sufficiently satisfactory* [12]. This notion communicates the idea that quality requirements are often subjective in nature. They are often achieved not in an absolute sense, but to a sufficient or satisfactory extent. Accordingly, the achievement of soft-goals is subject to the sufficiency of contributions from other (sub-)soft-goals.

The whole analysis process can be summarized into the following steps [12]:

- acquire knowledge of the application domain;
- identify specific non-functional requirements for the domain;
- decompose non-functional requirements;
- deal with priorities and trade-offs;
- identify operationalizations; and
- select among alternatives and evaluate the impact of decisions.

The use of the NFR framework does not force the application of these steps in a strict top-down manner. One may need to iterate over them many times during the design process. A developer may choose refinements, having operationalizations in mind. Thus, the development process may move up and down.

A summary of the details involved in the application of these steps follows [12]:

Acquire domain knowledge. A crucial aspect of the NFR framework involves acquiring and using information about the domain and the system to be developed. This includes the following subjects:

- **Functional requirements.** Essentially the functionality of a system is determined by what the system does. This is a crucial issue in acquiring domain knowledge.

- **Expected organizational workload.** Designing and developing a system respond to an organizational need. In turn, organizational needs may be expressed in terms of some metrics that determine the volume of work that the system will have.
- **Organizational priorities.** Some functional requirements that the system achieves exert more influence on the organization. The developer may identify those requirements and tag them as priorities. In this way, the developer can focus on what is important. Thus, it will provide a guideline for design decisions when constraints, such as money, are considered.

Identify non-functional requirements. The construction of an initial soft-goal graph starts with the identification of the main non-functional requirements that the particular system under development should meet. These non-functional requirements are treated as soft-goals to be clarified, elaborated upon, and so forth.

The developer will identify specific possible development approaches. At that time, he will choose solutions in the target system that meet the source requirements specification. In this way, the developer starts by systematically decomposing the initial NFR soft-goals into more specific soft-goals (or sub-goals).

Decompose and refining soft-goals. The initial non-functional requirements are usually broad and abstract. To effectively deal with broad requirements, high-level non-functional requirements may need to be broken down into smaller elements, so that an improved representation is achieved and effective solutions can be found. By treating a requirement at a high level, it can be represented as a soft-goal and decomposed into more specific sub-goals, which together should meet the higher soft-goal. Hence, soft-goals are refined downwards into sub-goals. In turn, sub-goals contribute upwards to parent soft-goals.

Deal with priorities and trade-offs. A very important result of acquiring domain knowledge is that it allows the designer to devise different alternatives of implementation and to establish priorities. Thus, the design process can focus on the most important issues. Some soft-goals will be identified as crucial as they are vital for the success of the system.

Priorities may be used to make appropriate trade-offs among soft-goals. Furthermore, knowledge of different trade-offs can be captured in catalogues and be reused in later stages of design when some conflicts become apparent.

Identify possible operationalizations. At some point, when the non-functional requirements have been sufficiently identified and refined, it is possible to find possible development techniques for accomplishing these non-functional requirements and then come up with specific solutions for the target system. These are the “operationalizations of the soft-goals”. Filling the gap between non-functional requirements and operationalizations involves performing analysis and dealing with several factors, such as priorities, trade-offs, and so forth.

The transition from “non-functional requirements soft-goals” to “operationalizations of the soft-goals” is a crucial step in the process. There can be different ways for refining and/or elaborating the operationalizations of the soft-goals. In turn, every way of refining operationalizations may be comprised by a selection of different alternatives.

Select among alternatives and evaluating the impact of decisions. So far, the designer has considered different non-functional requirements (soft-goals), priorities, trade-offs, and possible operationalizations. The whole process may continue until the possible solutions for the system under design have been taken into account, and no other alternative needs be considered.

Having a soft-goal graph containing all this information, the designer is prepared to

select among different alternatives. This step implies to choose some operationalizing soft-goals and evaluating the impact of their implementation on the fulfillment of the system requirements.

Soft-goal graph representation allows designers to evaluate the impact of selecting certain operationalizing soft-goal. Thus, the evaluation process works towards the top of the graph, determining the impact on higher-level soft-goals.

The evaluation of soft-goals is initiated by assigning labels. A “ $\checkmark$ ” or an “ $\mathcal{X}$ ” indicates whether the operationalization is selected or not. In this way, a positive contribution of the soft-goal propagates the label to its parent. Hence, a selected soft-goal ($\checkmark$) results in a satisfied parent ($\checkmark$), and a denied soft-goal ($\mathcal{X}$) results in a denied parent. It is also possible to identify a weak positive or negative support for a soft-goal parent. In that case, the soft-goals are labeled “ $\mathcal{W}-$ ” or “ $\mathcal{W}+$ ” to represent inconclusive negative or positive support, respectively. When there is no possible to establish a positive or negative support, parent goal is assigned the label “ $\mathcal{U}$ ”.

Thus, the evaluation process depends entirely on the selection of operationalizations and the corresponding contribution to the soft-goal parents. The evaluation process continues until the top-level soft-goals are reached.

Let us illustrate how to use the NFR framework to consider several alternatives and to evaluate the impact of decisions via an example taken from [25].

A bank is considering the appropriate number of tellers it needs in order to keep an acceptable performance. 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 3.3 shows these two possibilities and the evaluation of the impact of selecting each one. In the first alternative customer satisfaction is maximized (performance is satisfied $\checkmark$) by paying a high price (profitability is denied $\mathcal{X}$). The second choice implies tolerating an acceptable reduction in performance ($W-$) to obtain the essential profitability ($W+$). It is important to highlight that the second choice uses the Multiplexing technique [25]. This technique implies sharing an expensive or scarce resource provided that users are guaranteed a certain degree of satisfaction.

Figure 3.3: Two alternatives in considering the number of tellers of a bank

The degree of satisficing of top-level soft-goals (main non-functional requirements) is the result of this evaluation process. Now the designer may draw conclusions on the selection of different operationalizations and their impact on non-functional requirements.

3.2 The NFR Framework: A Goal-Driven Process

Summarizing, the NFR framework process starts with an initial set of both functional requirements and non-functional requirements defined at the highest possible level. Then, the non-functional requirements are represented as high-level soft-goals and decomposed into more specific sub-goals. Interdependencies state the relationship between soft-goals and their corresponding sub-goals. Iteratively, soft-goals are refined until it is possible to identify possible mechanisms for achieving their satisficing. Refinements aid disambiguation and prioritization.

Prioritization may be needed due to resource limitation, domain constraints, and so forth [12]. For example, due to limited development time available, the developer may need to spend more effort and time on high-priority soft-goals than on less-important ones. Thus, the provision of operationalizing soft-goals makes it possible to use non-functional requirements as criteria for selecting among design alternatives. This process of refinement, selection among alternatives, analysis of trade-offs and prioritization is repeated for all the high-level soft-goals until no more refinements are needed.

These non-functional requirements are next satisfied by operationalizing soft-goals, either design or implementation components, which can be used to meet functional requirements. Operationalizing soft-goals can also be refined via decompositions into other operationalizing soft-goals. In the presence of multiple competing techniques, trade-offs are considered before some of them are finally chosen to be included in the target design or implementation. Throughout the refinements of non-functional requirements, their operationalization, prioritization and selection among competing techniques, arguments can be captured to support particular decisions.

Figure 3.4 shows how (source) functional requirements and (target) different alternatives of implementations are related to each other and to the soft-goal graph. Note that the different alternatives of implementation are targets of the operationalizations. In turn, non-functional requirement decomposition influences the development

Figure 3.4: Non-functional requirements influence and guide the development process (adapted from [12])

process of taking functional requirements to a target system. Thus, by using the components and development process, the NFR framework takes a goal-oriented approach to dealing with non-functional requirements. This approach makes it possible to systematically treat non-functional requirements and ultimately use them as a criterion for making selections among competing target alternatives. Hence, it offers a goal-driven, top-down approach to dealing with non-functional requirements [12].

3.3 Chapter Summary

In this chapter we have considered several concepts involved in the use of the NFR framework, such as *soft-goal graphs*, *forces*, *claims*, *operationalizations*, *contributions*,

soft-goal decompositions into soft-goals, and so on. We have also reviewed the notion of *satisficing* and described the steps encompassed by the application of the NFR framework.

We have also stressed the *goal-driven* nature of the NFR framework. Essentially, the NFR framework is a *top-down approach*. It may be used to support system analysis by modeling non-functional requirements and the elements involved in their fulfillment. The NFR framework also helps establish a clear link between non-functional requirements and functional aspects of design alternatives.

Certainly, as designers we may want to benefit from the features provided by the NFR framework for modeling several design alternatives and evaluating their impact on design. However, we are also interested in taking advantage of good results from previous efforts. We do not need to reinvent the wheel. We could consider proven successful design alternatives and choose the most appropriate for the design issue under analysis. This idea, the reuse of best practices, is behind the notion of patterns and pattern languages. We will cover this interesting topic in the following chapter.

Chapter 4

Patterns

In Chapter 2 we reviewed the notion of patterns. This time we will consider how patterns can be interrelated; and how they, as building blocks, fit in frameworks. Finally, we will present a brief exploration of current approaches for organizing and selecting patterns. Thus, foundations for the new approach that relates patterns and non-functional requirements are laid for the next chapter.

4.1 Patterns and Pattern Languages

Patterns are not used in isolation. Although individual patterns are useful at solving specific design problems, we can benefit further from positioning them among one another to form a *pattern language*.

Pattern languages are networks of patterns of varying scales embodied as concrete prototypes. Each pattern occupies a position in a network of related patterns, in which each pattern is connected to the “smaller” patterns it contains, as well as to the “larger” patterns in which it itself is contained. A pattern contributes to the completion of patterns “above” it in the network, and is completed by patterns

“below” it. Although the network of patterns constituting the pattern language can contain cycles, its general form is thus hierarchical. In [17], Erickson summarizes the process described by Alexander [2] for generating a pattern language as follows:

1. Choose the patterns which best describe the overall scope of the project.
2. Go to the end of the patterns, 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.

Given a single coherent pattern language, Alexander offers a straightforward procedure for choosing relevant patterns to be compressed into the decision makers particular design [3]. However, with the adaptation of the pattern approach to software design and given the yearly deliberations to introduce new patterns [14, 23, 28, 42] the number and diversity of patterns available to the designer continues to grow. Hence, it is no longer practical to scan and choose from among a list of pattern titles as originally suggested by Alexander.

The pattern user must scrutinize the contents of each pattern before make a decision to apply it. Pattern writers tend to discuss the benefits and liabilities of applying their pattern, which are identified as consequences in some pattern formats. By reading the consequences, the pattern user can determine, by subjectively trading off the consequences of those patterns under consideration, to select the pattern with more

perceived merit [11]. Again, with the number of possible candidate patterns contending for the attention of the pattern user, this approach may become impractical [29].

4.2 Patterns and Frameworks

Frameworks are like kits of unfinished parts. When developing an application by using a framework, less effort must be expended before devoting attention to the details of the problem [33].

From the perspective of application frameworks, patterns can be seen as their building blocks. From the perspective of patterns, an application can be seen as a pattern for complete software systems in a given application domain. An entire application may be based on a pattern language [11], or a combination of several patterns from different sources.

Developing an application by first developing a framework, upon which details are to be added, takes longer to implement than simply developing code that solves a specific problem [33]. However, as the application evolves many new features may be added after the first release. Here is where developing a framework first pays off system development: addition of new features will be easier because behavior that is common to all features is factored out of application code and put into a framework.

4.3 Pattern Languages: A Lingua Franca for Design

System design must be a communicative process. This means that a common field for expressing and exchanging knowledge about system design is needed. It is also

required to engage the users in this process, no matter their lack of acquaintance in system analysis and design. This is a challenge, especially when considering the heterogeneity of individuals in a community of users. Thus, we need to involve to current users (if an existing system is to be modified), potential users, designers, developers, and other members of a community affecting (or affected by) the system.

The more entangled systems are with daily life, the more ‘users’ need to be involved in the design process. However, users are certainly no more homogeneous than ‘designers’. This issue raises a problem: how are the various stakeholders in the design process going to communicate with one another, when they share little or nothing in the way of a core discipline, practice, or theoretical basis?

In [17], Erickson defines *lingua francas* as conceptual frameworks that disciplines and professions bring to bear during the design process. The idea is that system design should include the development of a lingua franca for a particular design project, a common language that permits all stakeholders to participate more fully in the design process.

As it was originally devised, one of the goals pursued by pattern languages was to support the design of environments by helping non-architects participate in the design of their own environments. This goal is achieved by providing its users with a common language that enables them to reflect on their experiences and on the relationship between their experiences and their environment.

4.4 Representing a Pattern

From the very first moment that patterns were explored, they were conceived as textual based descriptions [1]. Graphical descriptions such as pictures, object diagrams and the like, which often accompany the text, are usually added to represent the desired solution structures suggested by the pattern.

Patterns are written using predefined templates or forms. Several pattern forms exist in the literature, differing by the elements they emphasize. For example, among others, there is the Alexandrian form [2], the GOF (Gang of Four) form [19], and the Coplien form [15]. In the following sections we will consider the Alexandrian form (the essential pattern representation), as well as a very comprehensive pattern representation developed by Buschmann et al. [11].

4.4.1 The Alexandrian Form

Although not explicitly stated, the Alexandrian form [2] is comprised by three elements:

- Context: This component includes information concerning important forces found in the situation addressed by the pattern.
- Problem: This part describes the pulling forces that demands a satisfactory solution. The persistence of the occurrence of the pattern is stated here.
- Solution: This element explains how the forces are resolved in a convenient way.

The Alexandrian form starts with a statement summarizing the idea behind a specific topic. For example, the Secret Place pattern [2] starts as follows:

Where can the need for concealment be expressed; the need to hide; the need for something precious to be lost, and then revealed?

That statement is followed by a discussion that demonstrates the existence of the pattern, emphasizing its simultaneous occurrence in different situations. In the case of the Secret Place pattern, this section partially reads:

*To live in a home where there is such a place alters your experience.
It invites you to put something precious there, to conceal, to let only some*

in on the secret and not others. It allows you to keep something that is precious in an entirely personal way, so that no one may ever find it. until the moment you say to your friend, "Now I am going to show you something special" – and tell the story behind it. There is a strong support for the reality of this need in Gaston Bachelard's The Poetics of Space (New York: The Omen Press. 1964).

Alexandrian format ends with some sort of recipe to help incorporate the pattern into an actual design. For the Secret Place case, it reads:

Make a place in the house. perhaps only a few feet square. which is kept locked and secret; a place which is virtually impossible to discover – until you have been shown where it is; a place where the archives of the house. or other more potent secrets, might be kept.

Thus, in Alexandrian representation for the Secret Place pattern, we can recognize the following elements:

- Context: Given by the interaction between a human being and his/her surroundings at home. as well as with his/her fellows. This also comprises human feelings and intimate or very valuable personal belongings.
- Problem: It is described in the opening claim: "Where can the need for concealment be expressed; the need to hide; the need for something precious to be lost. and then revealed?"
- Solution: It is given in the final statement: 'Make a place in the house. locked and secret; where the archives of the house, or other more potent secrets. might be kept.'

4.4.2 From Architecture to Software Engineering: Expanding the Alexandrian Form

When Software Engineering specialists devised the applicability of Alexander's approach, the generality of the Alexandrian representation was intentionally kept. In this way, the simplicity and generality of pattern representation was preserved. However, for the sake of implementation purposes, some designers and pattern reviewers break down the Alexandrian form by including additional sections to improve the understanding and the applicability of patterns (e.g., Gross and Yu [20], Smith and Williams [39], and others).

One of these 'expanded' versions for pattern representation is presented in [11] by Buschmann et al. This expanded format helps developers to recognize different elements involved in a solution described in a pattern. Likewise, this representation provides a helpful link for implementing a given solution. The referred format consists of the following sections:

- **Name:** The name and a short summary of the pattern.
- **Also known as:** Other names (aliases) for the pattern, if they are known.
- **Example:** A real-world example that demonstrates the occurrence of the problem and the need for the pattern.
- **Context:** The situations in which the pattern may apply.
- **Problem:** The problem the pattern addresses, including a discussion of its associated forces.
- **Solution:** The fundamental solution principle underlying the pattern.
- **Structure:** A detailed specification of the structural aspects of the pattern.
- **Dynamics:** Typical scenarios describing the run-time behavior of the pattern.

- **Implementation:** Guidelines for implementing the pattern.
- **Example resolved:** Discussion of any important additional aspects of the solution not covered in the previous sections.
- **Variants:** A brief description of variants or specializations of a pattern.
- **Known uses:** Examples of the use of the pattern, taken from existing systems.
- **Consequences:** The benefits the pattern provides.
- **See also:** References to patterns that solve similar problems, and to patterns that provide refinements to the pattern described.

4.5 Organizing and Selecting Patterns

Several efforts have been made for organizing patterns. In general, patterns are grouped under common aspects and areas of application. This leads to pattern catalogs, the most common scheme of organizing patterns.

4.5.1 Pattern Catalogs

In pattern literature several proposals regarding catalog of patterns have been presented. Some of these efforts are the following:

Pattern map. One of the first attempts to identify pattern interrelationships and to organize them took place during the conference PLoP'95 [14], it was called the pattern map [11]. Pattern writers linked their patterns to related patterns from other authors. They wrote the pattern names on paper, placed these sheets somewhere on the floor of the main conference room, and connected each pattern to related patterns

with string. Thus, a first picture of the pattern universe was obtained. Although this approach was very informal, about three hundred different patterns were connected.

Others followed this initiative. For example, in early 1996, the Hillside Group [24] took this pattern map and wrote more than one hundred and fifty pattern abstracts. This included the pattern name, a short problem description, the key ideas of its solution, and a reference to the full description of the pattern. These pattern abstracts were linked together using different relationship types.

Pattern system. Based on a broad classification for patterns (architectural patterns, design patterns, and idioms), Buschmann et al. [11] offer a methodology for selecting an appropriate pattern for a given problem. They state a pattern system for software architecture as “a collection of patterns for software architecture, together with guidelines for their implementation, combination and practical use in software development”. This concept, that resembles the pattern language notion, supports the development of high-quality software systems, by fulfilling both functional and non-functional requirements. Thus, pattern selection results from applying the following steps:

1. Specify the problem
2. Select the pattern category
3. Select the problem category
4. Compare the problem descriptions
5. Evaluate benefits and liabilities
6. Select the variant that best implements the solution to the design
7. Choose an alternative problem category

Pattern roadmap. In [33], Rogers presents a scheme for organizing patterns called Pattern Roadmap. It is intended to optimize the search time and make it less likely that applicable patterns be missed.

The Pattern Roadmap is a table containing three columns labeled: “If this applies:”, “Then use this pattern:”, and “And check the applicability of:”. Under the first column a description of pattern applicability is given. The second and third columns show several pattern names (initially Rogers included 12 patterns) corresponding to the description contained in the first column.

After considering the domain/facet description of the problem, a pattern user goes down the first column of the roadmap, writing down the names of those patterns that definitely apply. After going through all patterns, a pattern user goes back to the rows of each that apply and gives the patterns mentioned in the third column another look if they are not already deemed to be applicable.

4.5.2 Pattern Catalog Usefulness

Pattern catalogs are useful as long as the number of patterns is low and their interrelationships and applicability are well defined. However, since the universe of possibilities (patterns) grows immeasurably, using pattern catalogs may make very difficult to choose an adequate pattern. A pattern user would face a huge number of patterns as s/he chooses the more appropriate one for decision design, or when s/he incorporates patterns into the framework. It is also difficult to identify the interrelations between patterns. This factor hinders pattern language construction.

4.5.3 Pattern Decision Aid and the Need for a New Approach

During recent years, the pattern community has documented a large range of patterns for software design. Several books on patterns are available (e.g., [11], [19], [33], [36]), and many patterns are discussed in conferences such as PLoP [14, 23, 28, 42], as well as in mailing lists [24].

A pattern user must scrutinize the contents of each pattern before selecting the most appropriate. Pattern writers tend to discuss the benefits and liabilities of applying their pattern, which are identified as consequences in some pattern formats. The pattern user may determine, by subjectively trading off the consequences of the patterns under consideration, to select the pattern with more perceived merit. However, with the number of possible candidate patterns contending for the pattern user's attention, eventually this approach may become unachievable.

In [29], McPhail presents the *Pattern Decision Aid*, a decision analysis approach suitable for comparing candidate software patterns. The main idea is to organize them in such a way that more emphasis is given to those candidates that will enhance the decision maker's total satisfaction with recommended results.

To address the selection a *hierarchy of criteria* is established where many principles can be categorized or generalized. The criteria are analogous to the forces in the context of a pattern. Thus, pattern forces are organized into a hierarchy rooted on *total satisfaction*. Using an extension of ELECTRE II [38], the number of candidate patterns is refined to a practical number for the pattern user's final decision.

However, there is a gap between the notion of force hierarchy presented by McPhail and the way patterns are currently formally represented. We need a way of describing patterns in terms of forces and their decompositions, constraints and trade-offs, and existing links between forces and pattern contributions in a given context.

With the aid of the NFR framework we may formalize the representation of patterns by using the notion of force hierarchy. In this way, the existing gap between the 'classical' pattern representation and the notion of hierarchy of forces is filled. As a consequence, we may count on a means for supporting pattern selection.

4.6 Patterns: A Solution-Driven Approach

Let us review the notion of patterns. As mentioned in section 2.3, a pattern can be defined as a three-part rule, which expresses a relation among a certain *context*, a *problem*, and a *solution* [2]. In this way, problems specify conflicting forces, and solutions resolve the conflict of forces while context indicates priorities or preferences of the designer.

Thus, when considering the nature of the pattern approach in terms of forces, resources, and other possibly involved elements in a design issue, we observe that a pattern solution may have effect on certain resources (possibly constraints), and on some forces (related to the resources affected). Often, a solution provided by a pattern is the result of balancing some resources, and/or trading-off.

On the other hand, system goals may be linked to the forces discussed in patterns, and related to the decomposition of forces and corresponding resources when applying patterns during design. Using patterns provides a more convenient language when describing how the design contributes to high-level system goals, such as performance, reliability, and so on.

These considerations determine that the use of patterns in design constitutes a solution-driven method for the application of best practices. Any consideration on patterns establishes a solution whose application causes a re-adjustment of both resources and involved forces. This imposes a bottom-up way of addressing a design issue.

4.7 Chapter Summary

In this chapter we have established the importance of viewing patterns as integrated solutions. We need to interrelate patterns as we explore the most appropriate selection for a given design issue under analysis. In order to achieve this goal, we identified the need of counting on an effective way of *representing* and *organizing patterns*. Thus, we considered several approaches currently in use for describing patterns, from the format originally proposed in Architecture to templates used in Software Engineering. Although all those pattern representations help understand the solutions supplied by patterns, they do not help the *selection process* when the number of choices is high, neither they necessarily promote the *generative nature* of patterns.

This leads us to consider the need for a new approach for selecting patterns. The *Pattern Decision Aid* method constitutes an attempt to address this issue. Based on the notion of force hierarchy, this approach provides a systematic method for narrowing the number of patterns more suitable for use in a design. Nevertheless, a bridge is still required between current pattern templates and this approach. We still need a way of representing patterns that leverages their generative nature, promotes pattern understanding, and help their selection possibly with the use of the Pattern Decision Aid. This issue represents the core of this work.

With the discussion on patterns, pattern languages, and pattern organization and selection approaches presented in this chapter we have concluded the examination of the bases required to define the proposed design process. We have established the need for a new approach for organizing and selecting patterns, and we also count on a means that may help us address that need: the NFR framework.

Thus, in the following chapter we will define the new pattern representation and outline the proposed design process.

Chapter 5

A New Design Process

Patterns aid in documenting and communicating solutions to recurring problems. They describe not only how to solve design problems, but also why a solution is chosen over others and what trade-offs are made. On the other hand, non-functional requirements play a crucial role in understanding the problem being addressed, the trade-offs discussed, and the solution proposed.

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

By combining both approaches, we may derive useful results when conceiving a satisfactory architecture in terms of user needs. This is the key issue behind the design approach presented here. Therefore, in this chapter we will consider how both approaches are integrated in such a way that a design process may be initially devised.

5.1 Combining Patterns and Non-Functional Requirements

The notion of forces is the center of patterns [26]. Forces can be thought of pushing or pulling the problem towards different solutions. A good pattern resolves or balances these forces, and contributes toward a stable design.

Before establishing a new representation for patterns based on the NFR framework, we will present a discussion on forces and trade-offs based on [26]. This issue is essential in understanding the relationship between patterns and the NFR framework.

5.1.1 Forces and Trade-offs

In analyzing a given problem, we may find that not all forces are equally important. The relative importance of the forces is determined by the context. As Coplien and Schmidt state:

If we understand the forces in a pattern, then we understand the problem (because we understand the trade-offs) and the solution (because we know how to balance the forces) [14].

In design decisions trade-offs are ubiquitous. Both forces and resources are subject to scrutiny to figure out the most effective trade-off while requirements are still satisfied, or just to get a proper balance that will make the implementation efficient.

For instance, in a given system, components may be involved in many activities, as well as an activity may bridge several components. Putting together these two perspectives, an issue immediately arises: How to define the correctness of a given system? The answer is given by the following two assertions [26]:

- Safety: Nothing bad ever happens to a component.
- Liveness: Something eventually happens within an activity.

How can these two forces, safety and liveness, get sometimes into conflict? Safety failures lead to unintended behavior at runtime (things just start going wrong). Liveness failures lead to no behavior (things just stop running - nothing ever happens). Thus, an easy thing to do is to guarantee safety by destroying liveness (if nothing ever happens, nothing will certainly fail). Conversely, there are sometimes good reasons for selectively sacrificing some safety issues for liveness (after all, the system should come up with a result). Therefore, since a system with such an “easy” solution will never succeed, we have to consider balancing the relative effects of certain kind of failures (sacrificing safety), to promote liveness.

At the same time, system design promotes safety by nature. That is why some liveness might be sacrificed (at least, a minimum set of functions is left so that the user is satisfied). As we can see, both forces pull up. The system design is affected by this interaction, and the solution is a consequence of the resolution (balancing) of both forces. The solution must exhibit an acceptable degree of safety as it offers the users a satisfactory number of functions.

Regarding safety and liveness, two other important forces also demand attention [26]:

- Reusability: The utility of objects and classes across multiple contexts.
- Performance: The extent to which activities are executed soon and quickly.

Since performance goals require every function attached to a given component to execute soon and quickly, this force supports and extends the liveness force. On the other hand, a component is reusable if it can be readily employed across different contexts. It is usually possible to design components to be safe across all possible contexts. However, it is also possible that systems using safe components might encounter

liveness failures in it (e.g., deadlock). Conversely, the functionality of a component in an unsynchronized environment is always live (regarding the possibility of locking), but may encounter safety violations in a concurrent execution environment.

Thus, it is important to understand that some very useful and efficient components are not, and need not be, absolutely safe, and that useful services supported by some components are not absolutely live. Instead, they operate correctly only under certain usage contexts.

On the other hand, forces are not the only concern when considering trade-offs. System resources, such as time and space, are also subject to trade-offs. For instance, in discussing several implementation solutions for improving system performance, Smith and Williams [39] describe important rules based on trading-off space for time:

- Store pre-computed results rule: Compute the results of expensive functions once, store the results, and satisfy subsequent requests with a table lookup.
- Caching rule: Store data that are accessed most often to make them cheapest to have access to.
- Lazy evaluation rule: Defer evaluation until a result is needed. This rule is applied when instantiation and initialization of objects is postponed until the last minute (since the object may not be used).
- Data structure augmentation rule: Augment data structures with extra information, or change the structure so that it can be accessed more easily.
- Batching rule: Make fewer calls and process multiple requests with each call.

Conversely, although applying the previous rules allows improving the performance, some design decisions may lead to consider sacrificing some execution time for saving space. This might be appropriate when the actions are needed relatively seldom. Applying the following techniques leads to trade time for space [39]:

- Packing rule: Use dense storage representation to decrease storage cost by increasing the time required to store and retrieve data.
- Interpreter rule: Represent common sequences of operations compactly and interpret them as required.

Albeit the time-for-space trade-off does not appear to improve the performance, rules like packing can be applied to compress files or data to save time for network transfers. Hence, an indirect result is the improvement of the system performance. This illustrates how trading-off resources may constitute a crucial concern in system design.

5.1.2 Force Hierarchies: NFR-Based Representations for Patterns

As it was described earlier, forces, resources, and their trade-offs are intimately tied to design decisions. Moreover, 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 possible solutions are considered, and the more adequate, the one proposed by the pattern, is selected.

In the proposed design process, forces (non-functional requirements) 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 these resources.

A *force hierarchy* is a hierarchical graph of forces and their contributions on each other. At the top of the force hierarchy is total satisfaction. Force hierarchies are inspired by, and can be modeled as NFR soft-goal hierarchies (already discussed) [12].

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

- The *application domain* force hierarchy consists of a representation of the top-level non-functional requirements and the contributions of the resource constraints. This establishes the *base force hierarchy*, a starting point for representing associated patterns.
- By augmenting the base force hierarchy with the contributions of the pattern to constraints on these resources we obtain the *pattern solution* force hierarchy.

When documenting force hierarchies in practice we need not include a node for total satisfaction, as it is implied. Moreover, to avoid making the diagrams busy, we only show the top-level non-functional requirements and resources actually affected by the pattern in the augmented force hierarchy for a given pattern. Figure 5.1 shows the force hierarchy for the Centralized Design pattern in the distributed application domain (based on the design techniques discussed in [25]).

Using the force hierarchy for representing a pattern. Let us take a formal description of a pattern, the Deviation pattern [20, 31], and see how to represent it by using the NFR approach. A compacted version of this pattern is shown below:

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

Context: The purpose of a fire alarm system is to survey a plant, a building, or some smaller unit like an office or an apartment. 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

Figure 5.1: An example of a force hierarchy, showing the base force hierarchy and the pattern solution force hierarchy

normally operate as a single integrated system. A fire alarm system does not do very much most of the time. 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. When this happens the fire alarm system takes appropriate action such as alerting people in the building through alarm bells and text displays, invoking extinguisher systems, and calling the fire brigade automatically.

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. Such conditions may be faults, when for example some communication channel is broken, or when a backup battery is failing. There is another category of less critical abnormal conditions i.e. disturbances, such as temporarily disabled sensors or dirty sensors. When faults and disturbances are detected, they cause actions to be taken in a manner similar to that of fire alarms.

Problem: 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.

The logical behavior of the system is completely independent of its distributed nature; an actuator may depend on some particular input regardless of whether it is connected to the same control unit or some other control unit in the system. Furthermore, the number of inputs in the system can be considerable, and the number of control units can also be large. If every control unit were to store the current status of every input sensor, it would place heavy demands on the memory capacity of each control unit. 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.

The force hierarchy representation for the Deviation pattern is shown in Figure 5.2 (adapted from [20]).

The top part of the force hierarchy for the Deviation pattern indicates that the

Figure 5.2: The force hierarchy representation for the Deviation pattern

pattern writer faces two major non-functional requirements during design: Efficient memory utilization while the system performance achieves good performance. Thus, this represents the dilemma the designer has to resolve and for which the Deviation pattern provides a satisfactory solution.

Thus, the use of force hierarchies allows us to obtain a new representation for patterns. For viewing an extensive utilization of this new pattern representation, please refer to Appendix A. It contains a catalog of patterns for use in call center design. Under each pattern description, a complementary section called “NFR-based representation” shows the corresponding force hierarchy.

5.2 Defining a New Design Process

So far, we have defined a new pattern representation based on the relationship between patterns and the NFR framework. Grounding in the same principle, this section describes a new design process that makes use of the new pattern representation already presented.

5.2.1 Approaching the New Design Process

In accordance with the proposed design approach, let us redefine the roles described in section 2.3:

- **Pattern writer.** S/he defines the base force hierarchy for the domain and generates the new NFR-based representation for the set of patterns.
- **Pattern user.** Besides selecting and applying patterns to a design, s/he may also be able to refine the new NFR-based representation of a pattern, if new relationships, or forces, become apparent when applying selected patterns to the design.

Furthermore, the application of the design process demands the use of the following elements:

- **Application-domain force hierarchy.** Having known the domain of the application to be designed, as well as its high-level non-functional requirements, a force hierarchy for the application domain is established. To avoid ambiguities, it is often useful to state a conceptual definition of each non-functional requirement, and their associated resources. This application-domain force hierarchy constitutes the base force hierarchy for representing patterns.

- **Pattern catalog.** Depending on the domain of the system to be designed, a list of potentially applicable patterns is prepared. Using the base force hierarchy, the corresponding NFR-based representation for each pattern is derived. Thus, they are ready for considering their application during the design process.
- **Requirement analysis sheet.** This format allows the designer to separate each functional requirement, and to relate them to resources and high-level non-functional requirements involved in its achievement (see Figure 5.3).

Design document name		
Design Issues/ Functional Requirements	Resources	NFRs

Figure 5.3: Requirement analysis sheet

5.2.2 The Design Process Steps

The design process to accompany the new NFR-based pattern representation proceeds through the steps shown in Figure 5.4.

First step. The *first step* implies the definition of a force hierarchy for the application domain, its base force hierarchy, as described in section 5.1.2. Conceiving a force hierarchy may require ongoing refinement due to the following:

Figure 5.4: Steps of the new design process

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

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

Second step. In practice, the *second step* involves to construct not only an augmented force hierarchy for the accepted pattern solution, but also one for each design alternative. Although we could represent several design alternatives in the same force hierarchy (e.g., Figure 3.3 in sub-section 3.1 shows a force hierarchy with two design alternatives), this often quickly becomes unmanageable, and the impact of the various designs on the forces may not be clearly visible. We generally prefer to

represent the design alternatives in different hierarchies.

Third step. In the *third step*, we associate the impacted top-level non-functional requirements and common resources with each requirement. This is the most creative step in the process, and there are no hard and fast guidelines on how to go about extracting the forces from the requirements. It will be often easier to identify the resource constraints (e.g. reduce effort, or save money) first, and then the top-level non-functional requirements that will be affected. The resulting forces define the context for selecting patterns.

Fourth step. Based upon the impacted forces and resources identified in the third step, in the *fourth step* we select applicable patterns based on how closely they match the forces. Following a “best-fit” approach, this step is currently performed by scanning the forces resolved by each pattern. With the assistance of an automated tool for the Pattern Decision Aid [29] this step would not require this “best-fit” method and would allow us to count on a much larger and richer number of patterns.

Fifth step. Once all patterns have been selected, the selected pattern solutions can be incorporated into the current design in the *fifth step*. Except for small designs that designers can still manage in their heads, we should defer incorporating patterns until all requirements have been covered by the fourth step. 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 pattern language.

The first two steps represent the reusable portion of the proposed design approach. As the patterns are applied to concrete projects, omissions may be identified in the force hierarchy and in the pattern descriptions, which will then be iteratively refined.

From the third step on, specifying the application requirements in terms of the impacted forces and resources, the process involves the actual requirements for a given

system.

Since ongoing refinement is a crucial aspect of this design approach, a pattern writer may iterate as much as needed to refine the force hierarchies (both of the application domain and of the pattern solution). Likewise, a pattern user may iterate as much as needed to incorporate new solutions to design, as well as to provide new findings that could also contribute to refine the force hierarchies.

5.3 Chapter Summary

We started this chapter with a discussion on *forces* and *trade-offs*. This led us to the notion of *force hierarchy*, the basis for representing patterns. Force hierarchies comprise two scopes: *application domain* and *pattern solution*. Force hierarchies follow the same guidelines stated by the NFR framework for soft-goal graphs, with a variant introduced for incorporating aspects typically found in pattern descriptions: forces, resources, and other elements. The use of the new pattern representation is illustrated with its application to the Deviation pattern, as well as to patterns contained in the catalog used in the case study.

The new *design process* was defined. This comprised a redefinition of the roles involved in pattern application, as well as a description of the elements required in the new approach application. Moreover, we outlined the steps corresponding to the new design process. It starts with the definition of a force hierarchy for the application domain corresponding to the system under design. The final architecture results from the selection and instantiation of several patterns as several design issues (involving functional and non-functional requirements) are considered.

The *iterative nature* inherent to the proposed design process provides for ongoing refinement as design issues are considered, patterns are selected, and design matures.

Thus, we have obtained the two most significant contributions of this work:

- A new representation for patterns based on the NFR framework.
- A design process centered at the notion of force hierarchy.

It remains to consider how the design approach may be applied. We will illustrate the application of the new design process through a case study: devising the architecture of a call center.

Chapter 6

Case Study: A Call Center Architecture

As part of this research, a case study was conducted at Mitel Networks to re-examine the design of a call center project.

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 interoperability are emphasized. This serves as an adequate opportunity to explore recurring solutions, apply proven ideas, and derive conclusions for future developments.

6.1 Analyzing and Designing a Call Center

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 (e.g., specialists within a team) [5].

Figure 6.1: A basic structure for a call center application (adapted from [35])

For example, a travel agency may maintain a database of all its clients and each client may be assigned to a particular travel agent. When a particular client calls the travel agency, the call is delivered to a hunt group device that ordinarily would just route it to the next available travel agent. By using intelligent routing services, a call center application is able to override this default routing. It uses the identity of the calling device associated with a given call to identify the client in its database; the call may then be routed to that customer's assigned travel agent.

In order to get a better idea of its structure, Figure 6.1 shows a basic representation of a call center. To avoid suggesting a specific design, elements commonly found in call centers, such as ACD and PBX switches [5], were not included in this depiction.

Since the core of a call center system is software that controls a telephony switch (usually a PBX), call center applications may also be viewed as embedded real-time systems [39]. Embedded real-time systems are embedded systems that have deadlines

on their execution. These timing constraints arise because of the nature of the external events to which they must respond. Figure 6.2 shows the view of a call center application as an embedded real-time system, after adapting a model presented in [39].

Figure 6.2: Call centers viewed as embedded real-time systems (adapted from [39])

6.2 Applying the Proposed Design Approach

Mitel's call center project was divided into three phases (1.0, 2.0, and 3.0). Functional components addressed in Phase 1.0 (the phase considered in this case study) are the following:

- **Computer Telephony Integration.** It provides links from the PBX to customer applications, via API support, as well as through DLL and DDE connectivity.

- **Agent Desktop (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 manager and other agents.
- **Intelligent Routing (Interactive Voice Response).** This provides for customizable routing of calls based on caller inputs and system values such as time of day, day of year.

As it was mentioned earlier, the application domain that corresponds to call center design is distributed systems. Initially, 14 patterns documented in [11], [25], [33], and [39] were identified as potentially applicable to distributed systems (please refer to Appendix A for a description of these patterns). Informally, this resulted in a catalog from which the patterns were available for selection.

6.2.1 Outlining the Application of the Design Process

Let us establish the steps we will follow according to the proposed design approach.

Force hierarchies. This comprises the application of steps 1 and 2 of the proposed design process. Accordingly, it involves:

- **Step 1:** Definition of the force hierarchy for the application domain. Section B.3 describes the corresponding force hierarchy for call center design (distributed systems).
- **Step 2:** Definition of the force hierarchies for pattern solutions. This implies obtaining the NFR-based representation corresponding to the 13 patterns that comprise our informal catalog (in Appendix A, please refer to sub-section “NFR-based representation”, under each pattern description, to see these force hierarchies). Thus, patterns are ready for selection and application in step 4.

Iterations. To iterate on steps 3 and 4 of the proposed design process we will proceed as follows:

- Step 3: From the system documentation we extract a design issue concerning the fulfillment of functional requirements. Both the involved resources and the associated top-level non-functional requirements are identified and collected in the requirement analysis sheet.
- Step 4: The issue considered in step 3 is addressed by selecting applicable patterns. By now, we base the pattern selection on “best-fit” with the aid of the new NFR-based representation.

Three patterns will be selected, two for the architecture of the call center under the perspective of the Computer Telephony Integration functionality, and one for the structure of Agent Desktop applications. The Intelligent Routing functional issue will not be considered as system documentation does not provide information about this subject.

Pattern instantiations. Selected patterns in step 4 are instantiated in step 5. For the purposes of the case study, in step 5 we introduce a variant. After a pattern instantiation we insert a validation step comprised of a comparison between what pattern instantiation predicts for the resulting structure and what system documentation contains regarding the designer’s decision. This step is illustrated in Figure 6.3.

Thus, after selecting these patterns, a view of the structure of these components as well as their potential integration may be obtained once selected patterns are instantiated in terms of the call center design.

Figure 6.3: Validating the results of pattern instantiations

6.2.2 Iterating on Steps 3 and 4 of the Design Process

First iteration. From the system documentation 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 6.4 shows the aspects concerning this issue, as well as involved resources and non-functional requirements.

Call Center – System definition		
Design issues/ Functional Requirements	Resources	NFRs
Overall system architecture Adjunct vs. All in one Adjunct It provides support for existing and future PBXs. The switch ACD and agent log on would take advantage of All in one All the functionality is directly programmed in the switch	<ul style="list-style-type: none"> • Time (response time) some communication overhead and coordination tasks might be added (-) • Interoperability (+) Support for future incorporations of PBXs is required 	<ul style="list-style-type: none"> • Performance (-) • Scalability (+)

Figure 6.4: Considering the overall system architecture

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). This 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. The microkernel also serves as a platform for plugging in these extensions and coordinating their collaboration. The Microkernel pattern force hierarchy is shown in Figure 6.5. Effort and time are traded-off against scalability (here in the sense of allowing us to add components to the system). As a consequence of this, performance and efficiency are traded-off against scalability.

Figure 6.5: Force hierarchy for the Microkernel pattern

Second iteration. The second design issue discusses the *trade-offs between Computer Telephony Integration (CTI) and “Flash and Dial” integration* (see Figure 6.6 for details of this issue according to the system design documentation).

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 affect call center applications. Likewise, changes to the applications should not require changes to the switch.

Call Center – System Definition		
Design issues/ Functional requirements	Resources	NFRs
Call Control API for the application to communicate with the switch (PBX). CTI integration vs. “Flash and Dial” integration. <i>CTI integration:</i> <ul style="list-style-type: none"> • Changes in the PBX platforms are possible without affecting the applications • Changes in applications are possible without affecting the PBX platforms • Some degree of penalty in performance (indirection) <i>Flash and Dial integration:</i> <ul style="list-style-type: none"> • Changes in the PBX platforms will likely have impact on the way the applications interact with the PBX (and vice versa) 	<ul style="list-style-type: none"> • Scaling (applications and platforms) (+) • Time (response) (-) 	<ul style="list-style-type: none"> • Scalability (+) • Performance (-)

Figure 6.6: Considering the Call Control API for switch – application interaction

After matching the non-functional requirements related to this issue with the forces addressed by the Indirection/Binding pattern [25] (see its force hierarchy in Figure 6.7), 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.

Third iteration. This time, we need to devise the *agent desktop application structure*. Though it is stated that it has to follow the client-server model, it is also important to consider the non-functional requirements associated with this design issue (see Figure 6.8).

Figure 6.7: Force hierarchy for the Indirection/Binding pattern

Here the priority is the transparency (in terms of interoperability) that an agent desktop application requires in interacting with other software components (e.g., client-server interaction, Data Collector - agent desktop 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 [11] addresses the non-functional requirements related to this design issue, following the client-server model, it results the “best fit” from our pattern catalog (Figure 6.9 shows its corresponding force hierarchy).

6.2.3 Instantiating Patterns

After iterating on steps 3 and 4, we will instantiate the patterns and devise the resulting architecture for the call center.

Call Center – System definition		
Design Issues/ Functional Requirements	Resources	NFRs
Agent Station architecture • 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. <i>Notes:</i> Agent Station applications must follow the Client-Server model.	• Time (response time) As a server, response time of the server-side agent station application may be affected by the number of requests. • Scaling (+) Support for additional agents must be provided. Platform independence must be provided.	• Performance (-) • Scalability (+) • Interoperability (+)

Figure 6.8: Considering the architecture of agent desktop applications

Pattern 1: Microkernel.

- **Pattern instantiation:** Instantiation of the Microkernel pattern for the call center is shown in Figure 6.10. 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 structure of this pattern. This component is not filled yet by any object (this is fairly typical for architectural patterns such as the Microkernel pattern).
- **Compare and amend:** The designer's decision also implies leaving a minimal expression of functionality in the switch and to build on separate functions 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

Figure 6.9: Force hierarchy for the Broker pattern

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.

Pattern 2: Indirection/Binding.

- Pattern instantiation:** The Indirection/Binding pattern instantiation for the call center is shown in Figure 6.11. 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

Figure 6.10: Microkernel pattern instantiation for the call center

provide the core (switches) with a certain degree of independence with respect to call center applications.

- **Compare and amend:** The designer's decision entails the use of a CTI component. The main responsibility of this component, as is usual for applications based on CTI [5], 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 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.

Pattern 3: Broker.

Figure 6.11: Indirection/Binding pattern instantiation for the call center

- Pattern instantiation:** The instantiation of the Broker pattern is shown in Figure 6.12. This pattern establishes a client-server structure for the agent desktop 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.
- Compare and amend:** The system documentation states that the structure of the agent desktop 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) [40]. It makes possible that an agent desktop client application calls a remote component in a server, and that an agent desktop 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 desktop applications. It also discharges agent desktop applications from including

complex mechanisms for enabling interconnection. In this way, the Broker pattern matches the use of RMI.

Figure 6.12: Broker pattern instantiation for the call center

Consolidating a view of the call center architecture. So far, we have obtained a view of the pattern instantiation in each iteration. Figure 6.13 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 desktop applications. It also remains to define the structure of the Data Collector and its possible interaction with the Agent Desktop server.

A general view of the high-level architecture description provided by the system design documentation is shown in Figure 6.14.

We observe that the core and the functions has been separated (switch representing the core, intelligent routing as an adjunct function). In our design process this issue resulted from applying the Microkernel pattern. A CTI component handles the

Figure 6.13: Structure of the call center application after incorporating patterns into design (three iterations)

interaction with the PBX, supplying the indirection required. The structure resulting from applying the Binding/Indirection pattern satisfies this requirement.

The architecture of agent desktop applications is not completely clear yet. However, since they have to interact with third-party applications, as well as the data collector, we can expect that the Broker pattern provide what is required to satisfy the desired scalability.

It is important to highlight that as we iterate on the proposed design process, the generative nature of patterns has helped us identify and incorporate appropriate solutions into design.

Figure 6.14: An overview of the call center structure (actual design). Components partially addressed in the case study are shaded

6.3 Devising a Pattern Language

Let us emphasize the claim that closes the previous section:

...the generative nature of patterns has helped us identify and incorporate appropriate solutions into design.

As Schmidt et al. point out in [34], when reading a given pattern description in isolation it is not obvious how to apply a second pattern in the presence of a first one. Given a catalog of patterns, one might consider all their possible combinations. However, this approach is not practical in any way.

We could consider how the design 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, 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 initial pattern language (see Figure 6.15).

Figure 6.15: An initial attempt to devise a pattern language

Although the relationships among these three patterns are established as a consequence of the case study, and this constitutes a backbone for a design scheme based on these patterns, we could also take two possible paths to integrate more patterns to our initial language:

- Seek out loose ends in the new pattern representation and address their solution.
- Incorporate other patterns for which relationships have been demonstrated before.

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 application*. 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 system documentation when stating that entire control of call center configuration and management may be delayed until Phase 2.0 in favor of the time to market constraint. This issue may be addressed by applying the Deviation pattern [31] (to consider how to keep a compact representation of the state of the system) and the Blackboard pattern [11, 16] (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 a similar situation is stated in [10].

Now let us consider the second path. In [11], it is stated that the implementation of the functional components derived from the Microkernel pattern may 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, Layers will be placed after Binding/Indirection in the 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 6.16.

Figure 6.16: Completing the pattern language with known relationships (patterns added are shaded)

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), or by finding out new relationships through

the application of the proposed design approach (as shown in the case study).

6.4 Chapter Summary

The case study contained in this chapter serves to illustrate how the design approach proposed in this work may be applied to a specific design, in our case, a call center project at Mitel Networks.

It started with a brief description of the call center project. The application of the new design process was outlined, which involved a description of every step to be applied, as well as the number of iterations to be performed. A validation step was included to support pattern incorporation as design issues are considered. This step also allows us to compare partial results from pattern instantiations with actual components described in the system documentation.

In a nutshell, the case study results are the following:

- A preliminary view of the architecture for the call center. This comprised three design issues involving functional and non-functional requirements. The ensuing architecture resulted from selecting three patterns (one for each design issue), their corresponding instantiation, and integration of the resulting structures.
- A pattern language for distributed system design. Firstly, we devised some interrelationships among selected patterns through the application of the design process. Later on, we integrated those patterns to others whose interrelationships were already established.

Moreover, we validated the resulting call center architecture by comparing it to the actual design.

It remains to complete this work with some concluding remarks. They are presented in the following chapter.

Chapter 7

Conclusion

The intent of this work has been to present a design process based on addressing functional and non-functional requirements, as well as patterns through the definition of a new representation based upon the NFR framework.

On the one hand, the use of the NFR framework leads to a structured representation of forces and other elements involved in patterns, as well as to obtain a better understanding of pattern interrelationships. On the other hand, as patterns are essentially transformation rules that solve conflicting forces, using patterns to address soft-goal hierarchies may lead to a more effective and consistent way of addressing non-functional issues related to system design.

In this chapter, we will summarize how the proposed goals and contributions are met. Likewise, we will present a brief description of potential applicability of the results of this work, as well as possible further development.

7.1 Review of Objectives

Non-Functional Requirements. By elaborating on the NFR framework [12], particularly through the use of soft-goal graphs, modeling non-functional requirements and their contributions to particular goals is addressed to consider the achievement of non-functional aspects during design. The notion of force hierarchy, the basis for the new pattern representation introduced in this thesis, is based on this important aspect of the NFR framework.

Thus, based on the NFR framework, the new pattern representation developed in this work provides a basis for a design approach intended to consider the non-functional requirements inherent to a given problem, and address their fulfillment.

Patterns. In this work, the NFR framework has been used to represent both application domains force hierarchies and pattern solutions. As shown in the case study, this implies that several interrelationships may become apparent as patterns are selected and applied. Hence, this work has made possible both to provide a new representation for patterns and to develop pattern languages (by devising pattern interrelationships).

The description of this new representation based on the notion of the force hierarchy and the outlining of the steps that encompass the new design approach confirm the accomplishment of this goal. Moreover, the way the design process is applied to the design of a call center (case study) establishes the feasibility of the proposed approach.

7.2 Summary of Contributions

A review of the contributions this thesis has made is presented below:

- This thesis has provided a means to improve the understanding and applicability of patterns. This work has elaborated on this important issue through the development of a new and complementary representation for patterns. Pattern users may now capture the idea that a given pattern solution provides by using the new NFR-based representation. The development of this new pattern representation confirms this contribution. Over the case study the new pattern representation was extensively used: both for representing patterns from the catalog, as shown in a complementary section under each pattern description, and for supporting the selection of patterns in the case study.
- As seen, patterns are at the center of the proposed design process. For the call center, pattern selection and application through the use of the proposed design approach determined a preliminary architecture. The result of the structuring provided by the NFR framework is that the understanding of pattern contributions is easier and clearer for pattern readers. This encourages the use of patterns as proven and reusable solutions for new problems in system design.
- The iterative nature of the design process presented in this thesis 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.
- This work supports 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.
- A significant result of this work has been the development of a design process entrenched in functional and non-functional requirements, and patterns. The outline of the involved steps, as well as the definition of the design process interaction, confirm this contribution. Obtaining a preliminary architecture for a call center, the subject of the case study, served to illustrate how the design approach may be applied. Comparing the results from the case study with the

actual design aided in confirming the feasibility of the proposed design process.

- It offers support for devising pattern languages. With the support of the new design process, the way several patterns are interrelated was devised in the case study. Together with previous knowledge of other interrelationships in distributed systems, a pattern language was conceived based on the result of the pattern instantiations resulting from the case study. This serves to confirm that the new design process may be used to devise pattern languages.
- This study bridges the use of the Pattern Decision Aid. With a new pattern representation built on the notion of force hierarchy, a basis for the automation of the Pattern Decision Aid has been established. On the one hand, by providing a pattern representation entrenched on the notion of force hierarchy. On the other hand, by making available to pattern users a means to facilitate pattern selection once the Pattern Decision Aid has narrowed the number of candidates.

7.3 Potential Benefits and Applicability

Some benefits of the proposed approach are:

- 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 help designers make informed decisions and anticipate design issues for future application releases.
- To encourage users get actively involved in the analysis and design phases, by drawing on patterns and pattern languages.

Moreover, there are several areas to which the proposed design approach is potentially applicable:

- In early stages of design, when the structure of the application is progressively derived by applying architectural and design patterns [11].
- In evolutionary design approaches, such as *Extreme Programming* [6, 32], where design is continuously scrutinized by testing and confronting user requirements. In preparing the tests, validation of non-functional requirements may prove crucial for immediate decisions in design. Evolutionary approaches also exploit the use of best practices (patterns). Thus, the approach proposed here may also be useful in helping the selection of patterns, as well as in confirming, at a reasonable degree, the satisficing of non-functional requirements.
- In planning future releases. Often, when trade-offs are made, some decisions have to be postponed. The design approach described here allows designers to predict yet-to-come decisions by anticipating the impact of a given non-functional requirement and the need of satisfying it in an upcoming release.

7.4 Future Directions

The suitability of the proposed approach is clear for systems whose requirements and consequent design decisions are well established, as the conducted case study showed. Our next steps may be oriented towards three directions:

- Applying the design approach to an ongoing project. It would allow the complete application of the proposed design process. It also would allow us to confirm its suitability in a different context and provide the opportunity of feedback and refinement.
- Exploring the possibility of devising pattern languages through the analysis of the pattern representation for a given catalog of patterns. This would involve

integrating all those patterns under their application domain force hierarchy and analyzing their contributions and interrelationships. On the other hand, this would also imply considering the analysis of each pattern representation to devise possible consequences of pattern application, e.g., finding a subsequent pattern in a chain of a pattern language.

- Using the new pattern representation to complete the bridge between the Pattern Decision Aid and pattern descriptions through the development of an automated tool for supporting pattern selection.

Appendix A

A Catalog of Patterns for Call Center Design

This section contains a set of patterns applicable to distributed systems. In this work they are used in the application of the proposed design process to the design of a call center. Accordingly, a new section was added to each pattern description. It is called “NFR-based representation” and contains the new pattern format introduced in this thesis.

Thus, this appendix starts with a description of some principles taken from [39] that support several pattern solutions contained in this catalog. Later, patterns for distributed systems, documented in [11, 16, 20, 25, 31, 39], are described.

A.1 Pattern Principles

In [39], Smith and Williams present a set of principles for creating responsive systems. These principles help identify design alternatives that lead to meet performance objectives. Some of those principles are next summarized because they provide support

for patterns centered on performance described in this appendix.

Centering. Centering leverages performance by focusing attention on the parts of the software that have the greatest impact on it. Centering is concerned with identifying the subset of the system functions that will be used most of the time. These frequently used functions are the dominant workload functions. They also cause a subset of the operations in a software system to be executed most, as well as the code within the operations. Hence, improvements made in these dominant workload functions will have a significant impact on the overall performance of the system. The Centering principle states:

Identify the dominant workload functions and minimize their processing.

Locality. Locality refers to the “closeness” of desired actions, functions, and results to the physical resources used to produce them. The types of locality are: Spatial, Temporal, Effectual, and Degree. For instance, if we need to sort a list of names, and those data are in local memory instead of being on a disk on a remote node in the network, the locality is good. Here we are referring to spatial locality. The Locality principle states:

Create actions, functions, and results that are close to physical computer resources.

Processing Versus Frequency. This principle deals with the amount of work done when processing a request and the number of requests received. It pursues to make a trade-off between both. It may be possible to reduce the number of requests by doing more work per requests, or vice versa. The Processing Versus Frequency principle states:

Minimize the product of processing times frequency.

Spread-the-Load. When multiple processes require exclusive use of one or more resources, and must take turns having access to it/them, they will experience resource contention delays. These delays may be reduced if one of the following actions is taken:

- schedule the processes so that they do not use the resource(s) at the same time.
or
- divide the resource(s) so that the processes use distinct parts of the resource(s) and thus do not need the same resource(s) at the same time.

The Spread-the-Load principle states:

Spread the load when possible by processing conflicting loads at different times or in different places.

A.2 Catalog of Patterns

In this section a compacted version of the patterns comprising our catalog is presented. Table A.1 shows the pattern names and their corresponding source of documentation.

Pattern name	Source of documentation
Alternate Routes	[39]
Balancing	[25, 39]
Batching	[25, 39]
Binding/Indirection	[25]
Blackboard	[11, 16]
Broker	[11]
Coupling	[39]
Deviation	[20, 31]
Fast Path	[39]
First Things First	[39]
Flex Time	[39]
Layers	[11]
Microkernel	[11]
Slender Cyclic Functions	[39]

Table A.1: 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. For example, when there is only one downstream process for many incoming processes. Under high loading conditions, the downstream process could not keep up with the load, causing traffic jam.

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 (e.g., hashing algorithms, a different key assignment strategy, accessing different tables as the updates are executed, and so forth).

For the process coordination problem, we need to find a way to route requests to different processes. An alternate route solution would be to have multiple downstream processes, each one updating its own region of the database.

These solutions all are instances of the Spread-the-Load principle. They spread the load spatially by routing database accesses to different areas of the database and routing processing tasks to different processes.

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 A.1 shows the force hierarchy for the Alternate Route pattern.

Figure A.1: Force hierarchy for the Alternate Route pattern

Balancing. System performance improves if and only if we devote additional resources to a bottlenecked resource, the most constrained one. When we relieve a bottleneck, however, it is possible for another resource to become a bottleneck. Thus, we must remove the bottlenecks one by one until all resources are equally constrained.

Problem. Assume that we want to increase the speed with which we can 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. How to remove the bottleneck (tape drive) and get a balanced system (all the resources equally constrained)?

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 a second approach.

Solution. For the example, we must improve the I/O bus to match the drive speed, perhaps by using a different I/O bus technology (however, for the sake of the new technology, we could probably require a new type of tape compatible with the bus). 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.

The bottleneck device is the one with the highest utilization. This is the device that limits the scalability. Several approaches can be followed: changing the software, using a faster device, or adding more devices.

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

Problem. The problem occurs when requested tasks require considerable overhead processing for initialization, termination, and, in distributed systems, 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 uses the *Processing Versus Frequency* principle to minimize 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 A.3 shows the corresponding force hierarchy for the Batching pattern.

Figure A.3: Force hierarchy for the Batching pattern

Binding/Indirection. This pattern handles situations in which several levels of abstraction are established and 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 A.4 shows the corresponding force hierarchy for the Binding/Indirection pattern.

the notion that there will be some agents available, unknown to the agent at design time.

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 to the problem 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 A.5 shows the force hierarchy for the Blackboard pattern.

Figure A.5: Force hierarchy for the Blackboard pattern

Broker. Structuring 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 is required. If components handle communication themselves, the resulting system faces several dependencies and limitations. For example, the system becomes dependent on the communication mechanism used, clients need to know the location of servers, and in many cases the solution is limited to only one programming language.

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.

There should essentially be no difference between developing software for centralized systems and developing for distributed ones. An application that uses an object should only see the interface offered by the object. It should not need to know anything about the implementation details of an object, or about its physical location.

The Broker architecture pursues to balance the following forces:

- Components should be able to have access to services provided by others through remote, location-transparent service invocation.
- You need to exchange, add, or remove components at run-time.
- The architecture should hide system- and implementation-specific details from the users of components and services.

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. It achieves this goal by introducing an object model in which distributed services are encapsulated within objects. Broker systems therefore offer a path to integration of two core technologies: distribution and object technology. They also extend object models from single applications to distributed applications consisting of decoupled components that can run on heterogeneous machines and that can be written in different programming languages.

NFR-based representation. Figure A.6 shows the force hierarchy corresponding to the Broker pattern.

Figure A.6: Force hierarchy for the Broker pattern

Coupling. The Coupling pattern addresses the situation in which the best tool available is required for accomplishing a given task. In terms of object-oriented design, it means to match the interface of an object with its most frequent uses.

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 is an example of using the Centering principle to identify interfaces that should be streamlined. It uses the Locality principle to combine information likely to be needed together. It also uses the Processing-Versus-Frequency principle to minimize the total processing required for the interface.

Figure A.7: Force hierarchy for the Coupling pattern

The Coupling pattern is particularly important in distributed systems because the cost of remote calls is high. It is also significant for database systems because such a cost is composed of the cost of the data access and the network transfer of possibly large result sets.

NFR-based representation. Figure A.7 shows the 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 like an office or an apartment. 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. A fire alarm system does not do very much most of the time. 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. When this happens the fire alarm system takes appropriate action such as alerting people in the building through alarm bells and text displays, invoking extinguisher systems, and calling the fire brigade automatically.

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. Such conditions may be faults, when for example some communication channel is broken, or when a backup battery is failing. There is another category of less critical abnormal conditions, i.e. disturbances, such as temporarily disabled sensors or dirty sensors. 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.

The logical behavior of the system is completely independent of its distributed nature; an actuator may depend on some particular input regardless of whether it is connected to the same control unit or to some other control unit in the system. Furthermore, the number of inputs in the system can be considerable, and the number

of control units can also be large. If every control unit were to store the current status of every input sensor, it would place heavy demands on the memory capacity of each control unit. 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 A.8.

Figure A.8: 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 typically 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 (centering principle). 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.

A different situation in which the Fast Path pattern may be applied arises when some data are needed far more frequently than other data. In that way, Fast Path pattern is implemented by recognizing the more required data and minimizing their processing. For example, prices for the most frequently traded stocks should be cached to minimize the time required to retrieve them.

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 Deviation pattern is shown in Figure A.9.

Figure A.9: 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 more important tasks will be accomplished.

Problem. In a radar tracking system [13], tracks are processed in 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 (centering principle). For the radar tracking system, applying the First Things First pattern involves to compute 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 A.10 for the force hierarchy corresponding to the First Things First pattern.

Figure A.10: 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 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

Figure A.11: Force hierarchy for the Flex Time pattern

delays for computing resources. All these results improve the overall responsiveness of the system.

NFR-based representation. Figure A.11 shows the corresponding 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. Some parts of the system deal with low-level issues such as hardware traps, sensor input, reading bits from a file or electrical signals from a wire. There may also be high-level policies such as telephone billing tariffs. A typical pattern of communication flow consists of requests moving from high to low level, and answers to requests, incoming data or notification about events traveling in the opposite direction.

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.

Figure A.12: Force hierarchy for the Layers pattern

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 A.12 shows the corresponding 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 A.13 for the force hierarchy corresponding to the Microkernel pattern.

Figure A.13: 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 it was 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.

*APPENDIX A. A CATALOG OF PATTERNS FOR CALL CENTER DESIGN*¹¹⁷

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. This solution uses both the Centering principle and the Shared Resources principle.

NFR-based representation. Figure A.11 shows the corresponding force hierarchy for the Slender Cyclic Functions pattern.

APPENDIX A. A CATALOG OF PATTERNS FOR CALL CENTER DESIGN¹¹⁸

Figure A.14: Force hierarchy for the Slender Cyclic Functions pattern

Appendix B

Case Study: Conceptual Definition

In this appendix we present the most relevant aspects concerning the conceptual definition of the Mitel's call center project. This comprises both the functional and non-functional issues related to this project.

For the non-functional aspects, the application of the design process proposed in this project establishes the definition of a force hierarchy. Consequently, a definition of the non-functional requirements involved in the designer's view of the call center is included. Later, the force hierarchy corresponding to the application domain, distributed systems, is described.

B.1 Call Center Functionality

Mitel's call center project was divided into three phases:

- Phase 1.0 addresses three basic areas of functionality for the call center: Computer Telephony Integration, Interactive Routing, and Agent Desktop.
- Phase 2.0 is intended to improve the basic functionality in the areas addressed

by Phase 1.0 and provide functionality in ACD, Call Reporting, Management Desktop. Moreover, phase 2.0 adds FAX, e-mail, and Web capabilities to the call center.

- Phase 3.0 would enhance all areas providing enterprise-level capability to the call center.

Regarding Phase 1.0 (the subject of the case study), system level requirements, as well as requirements for the agent desktop, the agent desktop GUI, computer telephony integration, and system administration are condensed in Tables B.1, B.2, B.3, and B.4 respectively.

Requirement	Description
Operating System Support	MS Windows 95/98/NT Client Support MS Windows NT Server Support
Operating System Support (for future clients)	Macintosh Client Support UNIX, X-Windows Client Support
PBX Support	Support for Mitel SX-2000 LITE (including ACD-2)
SQL Database	The database published to the user shall be a SQL database.
Remote Support	<p>Agent: Support for remote agents at home from a central call center.</p> <p>Administration: Call center managers should be able to perform certain administrative functions through RAS (Remote Access Service) access.</p> <p>Web Administration: Call center managers should be able to manage the call center through Web access.</p> <p>Telephone Administration: Call center managers should be able to manage the call center prompts through phone access.</p>
Integration	<p>The system must support integration with the following components:</p> <ul style="list-style-type: none"> - Mitel SX-2000 (PBX) (via AFC card) - Mitel Telecom Servers (TS800S, TS1500SR) - Mitel DNIC phones - Dynamic allocation of DSP resources - NMS AG-X (support for the NMS voice cards for DSP resources). - "Turnkey" format: software loaded on the server, cards and software installed
Licensing	Tracking of concurrent desktop and "port" licenses

Table B.1: System-level requirements

Requirement	Description
GUI	The GUI must be non-intrusive.
Real-time statistics	The agent desktop will provide real time statistical information every five seconds.
Live call information	Live call information will be provided within five seconds of the event.
Agent sign-on	Agents can sign on from any supported station (local or remote).
Desktop configuration privileges	The administrator can assign privileges to agents to configure their desktops.

Table B.2: Agent Desktop requirements

Requirement	Description
Application integration	Ability to pop customer application invoking an .EXE or .DLL and pass DDE or keystroke strings with IR data to the application.
TAPI server	TAPI service available on Server.

Table B.3: Computer-Telephony integration requirements

Requirement	Description
System management	Single entrance entry: the manager does not have to double enter the same information. System backup: the manager will be able to back up and restore the entire database, voice prompts and system configuration and pass DDE or keystroke strings with IR data to the application.
Agent Desktop configuration	Configuration of the agent desktops applications will be done from a central location. Two levels of privileges will be provided: Administration and Agent.
Computer Telephony Integration (CTI)	CTI installation and upgrades will be integrated with the Agent Desktop installation and upgrades CTI configuration will be administered from a central location.

Table B.4: System administration requirements

B.2 Non-Functional Requirements

In considering non-functional requirements for a given project, we face the problem that several definitions can be found for every requirement. Hence, it is needed to define the non-functional requirements in accordance with the application domain. Consequently, in this section we present a number of definitions for different non-functional requirements applicable to distributed systems. These definitions have been adapted from [11], [12], [21], [39], and [44].

Efficiency: Efficiency deals with the degree at which all the resources are employed when the system goals are achieved. How this resource employment impacts response time is also covered by this non-functional requirement, but it is treated as an issue concerning performance (see the definition of performance).

- **Effort.** This concept comprises human involvement (labor) in all system lifecycle.
- **Physical resources.** It comprises:
 - **Memory.** Any media used to store data. Memory may be sub-classified into main and secondary memory.
 - **Money.** Any monetary source involved in designing, developing, and executing a system.
 - **Space.** Physical space required to locate the remaining resources associated to the system.

Maintainability: This non-functional requirement is concerned with problem fixing after a software system error occurs. Software architecture adequate for maintainability facilitates localizing changes and minimizing *side effects on other components*. It has also a direct impact on code *modifiability* and *testability*.

Performance: The extent to which system activities are accomplished promptly (soon and quickly). It comprises the following attributes:

- Latency: Time elapsed between issuing a request and servicing it.
- Throughput: Number of requests that can be processed in some specified time interval or, for database interaction, the amount of data that can be transferred in a given amount of time.
- Response time: Amount of time required to respond to an event.

Profitability: A measure of the total income of a project compared to the total monies expended at any period of time.

Reliability: This factor quantifies the necessary reliability of the product. This is usually expressed as the allowable *time between failures*, or the total allowable failure rate. It also comprises any negative effect on the system performance that a decision in design-time may have. Moreover, the way a system behaves under *overloading conditions* is also comprised under reliability.

Scalability: The ability of an information system to provide satisfactory performance as greater demands are placed upon it, through the addition of extra computing power. It involves the following issues:

- Scaling: This is the main measure of scalability. It expresses at which degree more *users, system features, components, or requests* may be added to actual design without imposing the need of additional resources.
- Interoperability: Software architecture is often required to offer well-defined access to externally-visible *functionality and data structures*. On the other hand, being abode by some *standards* also provides ways to support interaction with

other systems. An additional aspect of interoperability is that of the operational commonalties that allow a set of programs to interact on a given *platform*. We identify the following issues as concerning to interoperability:

Transparency: Transparency removes the need for a given component to explicitly know or define the status of another component when communicating. This involves *time transparency*, two or more components interact without being all in active state at the same time, and *location transparency*, two or more components interact without being aware of the location of each other.

Security: Information security means protecting information. Addressing security issues leads to consider the following scopes:

- Availability: Guarding against interruption of service.
- Confidentiality: Guarding against unauthorized disclosure.
- Data integrity: Guarding against unauthorized update or tampering.

B.3 Force Hierarchy for the Application Domain

Figures B.1 shows the force hierarchy corresponding to the application domain for call center design. Figures B.2, B.3, B.5, B.4, B.6, B.7, and B.8 show the corresponding decomposition for each of the forces that contributes to the total satisfaction in this domain.

Figure B.1: Force hierarchy for Total Satisfaction

Figure B.2: Force hierarchy for Performance

Figure B.3: Force hierarchy for Scalability

Figure B.4: Force hierarchy for Efficiency

Figure B.5: Force hierarchy for Security

Figure B.6: Force hierarchy for Maintainability

Figure B.7: Force hierarchy for Reliability

Figure B.8: Force hierarchy for Transparency

Appendix C

CTI Terminology

In the case study we borrowed some concepts from Computer Telephony Integration (CTI). This appendix contains a compendium of some CTI terms. These definitions were taken from [5].

Computer Telephony Integration (CTI). The use of CTI involves three aspects:

- **Call control:** Call control is the ability to observe and control telephone calls, switching features and status, ACDs and ACD agents, and to use switching resources that include tone generators and detectors.
- **Telephone control:** Telephone control is the ability to observe and control physical telephone devices as computer peripherals.
- **Media access:** Media access involves binding telephone calls to other media services such as voice processing, fax processing, videoconferencing, and telecommunications.

Client-Server CTI System Configuration. A client-server configuration involves indirect control of telephony resources using a fan-out component in the form of a CTI server. The CTI server sits between client computers and the telephony resources being accessed. Logical integration takes place through the indirect path of CTI messages.

This configuration involves a personal computer running a CTI application connected to a CTI server hardware component over a local area network. The CTI server is, in turn, connected to a switch.

There are three service boundaries in this configuration: A protocol boundary between the switch and the CTI server, a protocol boundary between the CTI server and the personal computer, and a programmatic boundary used by the application software.

CTI Software Framework. CTI software framework follows a layered approach and forms the basis for integrating CTI software components on a given hardware component.

TAPI. TAPI stands for Telephony Application Programming Interface. This API, provided by Microsoft as part of the Windows operating system and the Windows Open Services Architecture, allows telephony-specific applications to access CTI functionality at a low level.

In TAPI, function calls and data structures are designed for use with switching domains that support only first-party call control. This simplification allows for application implementations to be streamlined somewhat. Thus, calls and connections are all effectively interchangeable when observing and controlling a first-party switching domain. With this insight, TAPI defines an address as the logical device in the switching domain that can be observed and controlled. Multiple logical devices are represented by multiple addresses.

Telephony-specific applications. Telephony-specific applications are software components with observation and control features of telephony resources.

Telephony-aware applications. They are comprised by any applications taking advantage of one of the high-level CTI interfaces. They represent the primary vehicle through which system integrators, customers, and individuals take advantage of the benefits of telephony solutions.

Routing services. This CTI feature permits the computing domain to override the default call routing as determined by the switching domain's call processing resources.

Switch. A switch is an implementation of telephony resources that includes call processing and switching resources. The switch makes connections to other telephony products through lines that represent the cabling, wireless transceivers, or other means of connection. Lines fall into two categories: extensions and trunks. The point where a line is connected to a switch is called a port. Additionally, a switch may connect with other special peripherals through interface ports.

The size of the switch is fundamentally measured in terms of its capacity for connecting lines (trunks and extensions) and its capacity for handling calls.

Among the most common switches used in CTI systems we find the following:

- **ACD:** An ACD (ACD stands for Automatic Caller Distributor) is the first device an incoming call is presented to. An ACD device has built-in rules for distributing calls and may make decisions based on call control information (e.g., the time of the day, the last device the call was directed from, information captured directly from the caller, and so on).
- **PBX:** A private branch exchange (PBX) is a general-purpose switch. It typically implements all telephony features and services internally, and may connect

any device to any other. A personal PBX is designed to be a peripheral in an individual's home, office or small business. It is conceived with emphasis on requiring little, if any, administration or maintenance while delivering the complete range of telephony functionality found on a normal PBX.

A distributed PBX is an implementation of series of distinct PBX cabinets that may be located at some distance from one another. Thus, the implementation of call processing functionality and switching functionality is distributed among the pieces of the PBX; these pieces are constantly in communication in order to coordinate their activities.

Bibliography

- [1] Alexander C.: *Notes on the Synthesis of Form*. Harvard University Press. Cambridge, MA. (1964).
- [2] Alexander C.: *A Pattern Language*. Oxford University Press. New York. NY (1977).
- [3] Alexander C.: *The Timeless Way of Building*. Oxford University Press. New York, NY (1979).
- [4] Bass L., Clements P., Kazman R.: *Software Architecture in Practice*. Addison-Wesley (1998).
- [5] Bayer M.: *CTI Solutions and Systems: How to Put Computer Telephony Integration to Work*. McGraw-Hill Companies Inc. (1997).
- [6] Beck K.: *Extreme Programming Explained*. Addison-Wesley (2001).
- [7] Beck K., Johnson R.: *Patterns Generate Architectures*. European Conference on Object-Oriented Programming (ECOOP), pp. 139–149, (1994).
- [8] Booch G., Rumbaugh J., Jacobson I.: *The Unified Modeling Language User Guide*. Addison-Wesley (1999).
- [9] Borchers J.: *A Pattern Approach to Interaction Design*. , John Wiley & Sons. Ltd., (2001).

- [10] Bosch J., Molin P.: *Software Architecture Design Evaluation Transformation*. In Proceedings of the IEEE Engineering of Computer Based Systems Symposium (ECBS99), (1999).
- [11] 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).
- [12] Chung L., Nixon B., Yu E., Mylopoulos J.: *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers. (2000).
- [13] 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).
- [14] Coplien J., Schmidt D. (editors): *Pattern Languages of Program Design*. Addison-Wesley, (1995).
- [15] Coplien J.: *A Generative Development Process Pattern Language*. In: Coplien J., Schmidt D. (editors). *Pattern Languages of Program Design*. Addison-Wesley. (1995).
- [16] Deugo D., Weiss M., Kendall E.: *Reusable Patterns for Agent Coordination*. In: Omicini A. (editor). *Coordination of Internet Agents*, Springer. (2001).
- [17] Erickson T.: *Lingua Francas for Design: Sacred Places and Pattern Languages*. In Proceedings of Designing Interactive Systems (DIS 2000), ACM, Brooklyn, NY, August. (2000).
- [18] Fowler M.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, (1999).
- [19] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley. (1994).

- [20] Gross D., Yu E.: *From Non-Functional Requirements to Design through Patterns*. Faculty of Information Studies, University of Toronto, Toronto, Ontario, Canada, (2001).
- [21] Graubmann P., Hwang W., Kudela M., MacKinnon K., Mercouroff N., Watanabe N.: *Engineering Modelling Concepts (DPE Architecture)*. TINA Consortium, (1994).
- [22] Gullichsen E., Chang E.: *Generative Design in Architecture Using an Expert System*. In Proceedings of the Graphics Interface Conference. 425–433. (1985).
- [23] Harrison N., Foote B., Rohnert H. (editors): *Pattern Languages of Program Design 4*. Addison-Wesley, (2000).
- [24] Frye J., Yoder J. (editors): *The Hillside Group: The Patterns Home Page*. URL: <http://hillside.net/patterns>
- [25] Keshav S.: *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley, Reading, Massachusetts, USA (1998).
- [26] Lea D.: *Concurrent Programming in Java - Design Principles and Patterns*. 2nd. Edition. Addison Wesley Longman, Inc., (2000).
- [27] Lethbridge T., Laganieri R.: *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*. McGraw-Hill, London. (2001).
- [28] Martin R., Riehle D., Buschmann F. (editors): *Pattern Languages of Program Design 3*. Addison-Wesley, (1998).
- [29] 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).
- [30] Meszaros G., Doble J.: *A Pattern Language for Pattern Writing*. Conference on Pattern Languages of Program Design, 529-574, (1998).

- [31] Molin P., Ohlsson L.: *Points & Deviations - A Pattern Language for Fire Alarm Systems*. published in Pattern Languages of Program Design 3. Addison-Wesley, (1998).
- [32] Newkirk J., Martin, R.: *Extreme Programming In Practice*. Addison-Wesley (2001).
- [33] Rogers G. F.: *Framework-Based Software Development in C++* Prentice-Hall Inc., Middletown, New Jersey (1997).
- [34] Schmidt D., Stal M., Rohnert H., Buschmann F.: *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Ltd.. (2000).
- [35] Sevcik P., Forbath T.: *The Call Center Revolution. Converged Networks Enable Integrated Customer Service Solutions*. Northeast Consulting. Boston, MA. (1998).
- [36] Shalloway A., Trott J.: *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, (2002).
- [37] Simon H.: *Models of Bounded Rationality*. Vol. 2. Cambridge, MIT Press. (1982).
- [38] Simpson L.: *Do Decision Makers Know What They Prefer?: MAVT and ELECTRE II*. Journal of the Operational Research Society. 47, 919-929. (1996).
- [39] Smith C., Williams L.: *Performance Solutions. A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley (2002).
- [40] Sun Microsystems, Inc.: *Java Remote Method Invocation - Distributed Computing for Java*. URL: <http://java.sun.com/marketing/collateral/javarmi.html>
- [41] Thayer R., Dorfman M.: *System and Software Requirements Engineering*. IEEE Computer Society Press, (1990).

- [42] Vlissides J., Coplien J., Kerth N. (editors): *Pattern Languages of Program Design 2*. Addison-Wesley, (1996).
- [43] *Webster's Collegiate Dictionary*. Tenth Edition. International Thomson Publishing. (1998).
- [44] *Wideman Comparative Glossary of Project Management Terms V. 2.0*. URL: http://www.pmforum.org/library/glossary/PMG_P06.htm. (1999).
- [45] Yu E., Liu L. *OME (Organization Modeling Environment)*. Project Home Page. URL: <http://www.cs.toronto.edu/km/ome>. (2001)