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[®]

Determining Pattern Applicability Using Non-Functional Requirements

by
James Moody, B.C.S.

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

School of Computer Science
Carleton University
Ottawa, Ontario
May 2002

Copyright © 2002 James Moody

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

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

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, 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.

0-612-72081-0

Canada

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

**Determining Pattern Applicability Using
Non-Functional Requirements**

submitted by James Moody, B.C.S.

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

Thesis Supervisor

Director, School of Computer Science

Carleton University
May 8, 2002

Abstract

Design patterns are a useful aid to software engineers. They describe known solutions to recurring problems in software design, and form a vocabulary for communicating these solutions to others. Selecting design patterns that are compatible with the stated non-functional goals of a software project is a difficult problem. In particular, while design patterns describe their direct impact on a small set of non-functional requirements, the indirect effects of the pattern in a particular context is unknown.

We describe a method for formally specifying design patterns, non-functional requirements and their interactions. We introduce an evaluation algorithm that uses a pattern as the basis for a traversal over a graph of non-functional requirements to determine how each is affected. Finally, we present a software tool that operates on catalogues of patterns, non-functional requirements and forces to determine the impact of each of the patterns on a specified set of non-functional requirements.

Acknowledgments

I would like to thank Dwight Deugo for acting as my supervisor. His interest, feedback and constant accessibility have provided me with an extremely valuable learning environment at Carleton.

The family that I have at OTI has been crucial to my academic experience. Greg Adams and Kevin Haaland have not only given me the encouragement, the time and the flexibility to pursue this worthy goal, but also have been the source of wise advice. Kevin McGuire, in addition to being my team leader, has also been a great friend, a diligent sounding board for ideas, and a tireless reviewer of my work. I thank him and the rest of the Eclipse team for their patience and support.

Finally, I would like to thank my wife, Carletta Shannon, for being the greatest supporter of all. The extent of her enthusiasm, commitment and love cannot be expressed.

Contents

1. Introduction.....	1
1.1 Problems with Choosing a Pattern.....	2
1.2 Requirements	3
1.3 Our Approach.....	4
1.4 Our Contribution.....	6
1.5 Organization of This Thesis.....	7
2. Related Work	8
2.1 Patterns.....	8
2.1.1 Patterns in Computer Software	10
2.1.2 Pattern Forms	10
2.1.3 Forces are Important	11
2.1.4 Formalizing Patterns	14
2.2 Non-Functional Requirements	15
2.2.1 Softgoal Interactions	17
2.2.2 Evaluating Softgoal Interdependency Graphs	19
2.2.3 Cataloguing NFRs.....	22
2.3 From Patterns to Non-Functional Requirements	23
2.4 Extensible Markup Language (XML).....	24
2.5 Eclipse.....	25
2.6 Summary	26
3. Design and Methodology	28
3.1 Outline.....	29
3.2 The Structure of Pattern Forces	29
3.3 Identifying Forces in Patterns	33
3.4 Cataloguing Non-Functional Requirements	34
3.5 Specifying Machine-Readable Catalogues of NFRs and Patterns.....	35
3.5.1 Representing Requirements	35
3.5.2 Representing Forces.....	36
3.5.3 Representing Patterns.....	41
3.6 The Evaluation Algorithm	42
3.7 Conflicts During Evaluation	43
3.8 Extensions to the Evaluation Procedure	44
3.9 Implications of Impreciseness.....	51
3.10 Automated Tool Design Goals.....	54
3.10.1 Design Decisions	55
3.10.2 Designing the Domain Objects	56

3.10.3 Design of Requirements and Forces	56
3.10.4 Design of Force Graphs	60
3.10.5 Design of the Evaluation Algorithm	62
3.11 User Interface	68
3.12 Summary	72
4. Results	73
4.1 A Sample Non-Functional Requirement Catalogue	73
4.2 Object Creation Patterns	76
4.2.1 Abstract Factory	77
4.2.2 Factory Method	78
4.2.3 Singleton	78
4.2.4 Summary of Patterns	79
4.3 Considering the Indirect Impact of Patterns	80
4.3.1 Indirect Impact of the Abstract Factory Pattern	80
4.3.2 Indirect Impact of the Factory Method Pattern	81
4.3.3 Indirect Impact of the Singleton Pattern	82
4.3.4 Results of Manual Evaluation	83
4.4 Running the Software Tool	85
4.4.1 Creating Machine-Readable Input	85
4.4.2 Obtaining Results from the Software Tool	89
4.5 Resolving a Conflict	91
4.6 Comparing Results	93
4.7 Scalability Issues	94
4.8 Observations	95
5. Conclusions and Future Work	97
5.1 Summary	97
5.2 Comparison to Previous Work	99
5.3 Limitations	100
5.4 Extensions and Future Research	101
5.5 Conclusions	103

List of Tables

Table 1: Mapping Contribution Types to Labels	20
Table 2: Transition Table	45
Table 3: Initial Extensions	46
Table 4: Some+ and Some- Contributions for W-	46
Table 5: Defining Helps and Hurts Transitions	47
Table 6: Defining Makes Transitions	47
Table 7: The Complete Transition Table	48
Table 8: Summary of Pattern Impacts on NFRs	79
Table 9: Impact of Patterns on Target Requirements	83

List of Figures

Figure 1: Conflict Resolution.....	68
Figure 2: Specifying Input Catalogues	69
Figure 3: Selecting Target NFRs	70
Figure 4: Selecting Resolution Strategy	71
Figure 5: Defining Target Requirements	73
Figure 6: Reusability Helps Development Cost.....	74
Figure 7: Extensibility Helps Reusability	74
Figure 8: Verifiability Helps Traceability.....	75
Figure 9: Simplicity Helps Verifiability	75
Figure 10: Impact of Abstract Factory	81
Figure 11: Impact of Factory Method	82
Figure 12: Impact of Singleton	83
Figure 13: Locating the Sample Input	89
Figure 14: Selecting Development Cost and Traceability	90
Figure 15: Introducing a Conflict	91
Figure 16: Extensibility Conflicts with Coupling	92

1. Introduction

Design patterns describe known solutions to recurring problems in software design. A pattern not only contains a problem and a corresponding solution, but also describes the context that gives rise to the problem. Most importantly, a pattern contains a set of often-contradictory forces that are balanced by the solution. Patterns are pervasive in software design because they provide a common vocabulary with which developers may communicate solutions.

The forces in a design pattern describe how the solution affects various attributes of the software system. Many of these forces describe the direct impact the pattern will have on attributes of the system. The forces are often contradictory; the solution achieves some of them in favour of others. An understanding of these trade-offs is critical to making an informed decision to use a particular pattern.

Pattern forces are often expressed in terms of non-functional requirements (NFRs). NFRs are non-behavioural attributes of a software system, which describe not what the system does, but how it does it. Common NFRs include speed, usability, security and reusability. Many of these requirements are desirable in a system, but are often neglected due to a lack of understanding of how to achieve them. As a result, developers often attempt to satisfy NFRs after the fact when problems arise, rather than at design time.

1.1 Problems with Choosing a Pattern

Hundreds of books and papers document thousands of design patterns. They range from very general patterns, which are applicable to any system, to domain-specific patterns that solve very specialized problems. The sheer number of identified patterns and the rate at which newly discovered patterns are being published ensures that one developer's familiarity with them is necessarily incomplete.

The fact that there is more than one way to solve a problem is reflected by the fact that multiple patterns may purport to solve the same problem. In such a situation, deciding which pattern to apply may be difficult. These patterns likely balance forces in different ways. Therefore, deciding whether or not using a pattern is appropriate is a matter of determining whether the pattern balances the forces in a way that is most compatible with one's goals.

The forces in a pattern are not always apparent. A variety of pattern forms exist, and each of these forms specifies forces in a different way. Some advocate the explicit enumeration of forces, but others bury them in paragraphs of prose, making it difficult to extract. In order to analyse a pattern's forces they must be identifiable; this limitation must be overcome in order to make efficient use of forces.

Pattern forces often describe the direct impact of a pattern on a set of NFRs. However, it is clear that, independently of any pattern, an NFR may impact other NFRs. These requirement interactions may be general in nature, applying to any problem. A typical example of such an interaction is the relationship between size and speed; each negatively impacts the other. Relationships between NFRs may also be domain-specific. If a catalogue of NFRs and their relationships is constructed for a particular domain, it is

clear that through a series of interactions, a pattern may have an indirect impact on an NFR that is not mentioned in the pattern text.

If the choice of a pattern should be based heavily on ensuring that its impact is consistent with the developer's non-functional goals, then it is necessary to consider requirements that are indirectly affected by the pattern, as well as those the pattern directly affects. However, the indirect impact of a pattern on some NFRs is not specified in the pattern text; the pattern describes only the direct impact on a much smaller set of requirements.

There are often large numbers of identified NFRs at work in the context of a particular problem, independently of a pattern. The interactions between them are similarly numerous. Determining the impact of a pattern on even a single NFR in this set, given a complicated set of interactions, is time-consuming and error-prone if performed manually. Computing the impact of a large number of patterns on a large number of NFRs becomes even more difficult.

In this thesis, we focus on the process of identifying the effect of a pattern on a set of NFRs. We will demonstrate that it is possible to determine the impact of the pattern on each of the NFRs. Due to the inherent difficulties of doing this manually, we will introduce a method of automating this procedure so that it may be performed quickly with minimal user input.

1.2 Requirements

We will address the stated problems by adhering to the following requirements:

1. The ability to determine the indirect impact of a set of patterns on a specified set of NFRs is crucial to the decision-making process when choosing a pattern.
2. A catalogue of NFRs and their relationships describes many aspects of the problem domain. This catalogue should be assembled only once and reused when determining the impact of each pattern on the set of NFRs.
3. This process must be largely, if not completely, automated due to the potentially large number of patterns, NFRs and relationships, coupled with the complexity of determining the indirect impacts.
4. The automation of this process requires that patterns, requirements and forces be specified in a machine-readable format.

1.3 Our Approach

With the goal of introducing an automated tool to determine the indirect impact of a pattern on a set of NFRs, we begin by defining a formal specification. This specification, written in XML, describes how patterns, non-functional requirements, and forces may be stored in a machine-readable format. It allows us to extract the forces from a pattern and specify them in such a way that they may later be linked to the non-functional requirements in the catalogue. Thus, large reusable catalogues of patterns may be assembled and reused. Similarly, domain-independent or domain-specific NFRs and their interactions may be captured in a catalogue and reused.

Using previous research about non-functional requirements, we know that it is possible to trace the impact of one or more requirements on others. A graph of requirements is constructed, with each requirement represented by a node and each force

represented by a directed edge. A subset of the nodes is given initial values designating each node's initial "label". This label describes whether the corresponding requirement is satisfied or not, and to what degree. Using a graph evaluation algorithm, labels are propagated from the initial nodes to other nodes in the graph depending on the types of relationships between them. At the end of the algorithm, all the nodes in the graph are labelled.

A pattern's forces describe the direct impact of the pattern on a set of NFRs. We will use this set of NFRs to provide the initial node labelling in the graph described above. The evaluation algorithm is then applied. The resulting label on a node indicates the indirect impact of the pattern on the corresponding requirement.

By applying the algorithm successively to a large number of patterns, using the same catalogue of NFRs and forces, the developer may determine, in an automated way, the impact of each of the patterns on a set of NFRs. This information is not available solely by reading the pattern, since the pattern lacks much of the information encoded in the NFR catalogue. Even with such a catalogue in hand, achieving these results manually is cost-prohibitive due to the potentially large number of patterns, NFRs and forces.

We introduce a sample implementation of a software tool that processes our XML format and applies the evaluation algorithm. The input to the tool is as follows:

- A catalogue of patterns
- A catalogue of NFRs and their relationships
- A set of target NFRs, which is a subset of the NFRs containing those requirements for which labels should be generated

The output of the tool is the impact of each pattern on each NFR in the target set.

1.4 Our Contribution

Forces are often difficult to discern from the pattern text. Not only is prose representation of forces problematic from the point of view of an automated tool, but it is also an obstacle to a clear understanding of the pattern. To alleviate this problem, we introduce an XML specification for patterns, NFRs and forces. While formal specifications for patterns existed previously, most focus on formally specifying the pattern solution. None attempts to specify the forces in a way that is suitable for this type of computation; ours does so.

The forces in a pattern describe the direct impact of the pattern on a set of non-functional requirements. However, it does not describe the indirect effects of the pattern on other requirements; these effects may be far-reaching and even domain-dependent. To attempt to discover these effects, we first formalize extensions to the graph evaluation algorithm described in [Chung 99]. These extensions allow the evaluation algorithm to propagate more accurate labels to the nodes. We then describe how this algorithm may be incorporated into a larger algorithm that determines the indirect impact of a set of design patterns on a set of non-functional requirements.

The fact that there are simply so many patterns is problematic for someone searching for an appropriate pattern. The necessary unfamiliarity with most patterns makes it time-consuming to consider each pattern as a candidate. As an automated solution is one answer to this problem, we introduce an automated software tool. This tool takes as its input catalogues of patterns, non-functional requirements and forces.

With a large degree of automation, this tool uses the algorithm described above to determine how each of the target requirements is affected by each of the patterns.

1.5 Organization of This Thesis

The remainder of this thesis is organized as follows. Chapter two presents a comprehensive review of previous work relative to this problem. Chapter three introduces our formal specification of patterns, NFRs and forces, describes an algorithm to determine the indirect impact of patterns on NFRs. Also included in chapter three are the details of our implementation of an automated software tool. Chapter four discusses the success of the software tool by analysing the results achieved when running the tool on a set of sample data. Chapter five summarizes the work and suggests directions for future research and extension.

2. Related Work

A wealth of information exists in the domains of patterns and non-functional requirements. This section discusses patterns, both in their general form and as they apply to software design, and describes the primary components of these patterns. Various pattern forms are presented, and commonalities and differences are discussed. Particular attention is given to the forces contained in patterns. An understanding of what patterns are and how they are structured, as well as an understanding of the forces contained therein, is critical to our investigation into how one can determine the indirect impact of a pattern on non-functional attributes of a system.

Non-functional requirements are introduced, and their variations and interactions are examined. As non-functional requirements can describe the attributes of a system that patterns may indirectly affect, it is important to understand their interdependencies. We explain the process of constructing a graph of non-functional requirements and the subsequent evaluation of the graph.

Commonalities between patterns and non-functional requirements are discussed. This discussion will show that pattern forces will provide initial labelling for some nodes in the graph of NFRs. Evaluation of this graph will determine the impact of a pattern on NFRs contained in the graph.

2.1 Patterns

In 1979, architect Christopher Alexander published the book “The Timeless Way of Building” [Alexander 79]. After having observed buildings and spaces around the

world, Alexander delves into a deep analysis of quality. He recognizes that some buildings possess an un-nameable goodness that he calls the “Quality Without A Name” [Alexander 79]. Moreover, he states that this quality is not a new discovery: buildings thousands of years old possess it. By examining building designs, he notices recurring themes that impart the Quality Without A Name to those spaces that use them. He calls these “good” solutions patterns. He defines a pattern as “a three-part rule, which expresses a relation between a certain context, a problem, and a solution” [Alexander 79]. He goes on to describe a pattern in terms of the resolution of forces, stating “each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves” [Alexander 79].

Alexander claims that by appropriately using patterns as templates to solve recurring problems, one can create buildings that possess the quality referred to above. An important point is that simply using a pattern is not a guarantee of goodness; pattern use must take into account not only the problem being solved but also the context and the specific set of forces being balanced. A document of frequently asked questions related to the discussion of patterns gives some more insight into the balancing of forces:

In architecture, or in mechanical / civil engineering, a designed entity exists in relationship to a system of interacting physical forces. Designs which do not resolve each force, and the system as a whole, fail. ... By extension, designs must satisfy other demands which interact in complex, and sometimes unforeseen ways; the term “force” was extended to cover these cases [Lea 00].

2.1.1 Patterns in Computer Software

In 1994, fifteen years after Alexander's book, "Design Patterns: Elements of Reusable Object-Oriented Software" [Gamma 94] was published. This book is popularly referred to as the "Gang of Four Book" for its four authors, Gamma, Helm, Johnson and Vlissides. This book is a direct application of the principles described by Alexander to the field of software development:

Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context [Gamma 94].

Gamma et al. present twenty-three patterns in software design that have been observed in practice. These patterns have formed the basis of a vocabulary with which software developers can more easily communicate design decisions and solutions. The number of books that have since been written to document general or domain-specific software design patterns is enormous. Many annual PLoP (Pattern Languages of Programs) conferences are held [PLoP 99, EuroPLoP 99, ChiliPLoP 02], and patterns figure prominently in workshops and paper submissions at such conferences as [OOPSLA 02] and [ECOOP 02].

2.1.2 Pattern Forms

The structure of a pattern varies according to the form used to express it. Many pattern forms exist which accommodate a variety of styles. The Alexandrian form [Alexander 77] is a relatively informal style that provides a clear separation between a problem and its solution. The "Gang of Four" form introduced by Gamma et al. [Gamma

94] provides more clearly identifiable sections, including optional sections describing implementation, sample code, known uses and related patterns. The Portland form [Cunningham] provides more flexibility to the author to express the pattern in prose style. The Canonical form [Appleton 00] contains much structure and a clear description of what is to be contained in each section. Regardless of which form is used, a pattern has a number of essential constituents, without which it may not be called a pattern. Depending on the form, these pieces of information may be contained in different places, or may not be explicit, but they are always present.

2.1.3 Forces are Important

A pattern contains a number of components, regardless of the form in which it is presented. To begin with, each pattern requires a name. Patterns form a vocabulary with which people may communicate about a particular domain; without a label for them, the essential requirement of communication is hampered. The choice of a name is beyond the scope of this paper, but it must effectively and concisely communicate the purpose of the pattern [Appleton 00, Gamma 94]. Examining Alexander's definition above, we see that, aside from the name, the four critical components of a pattern are the context, the problem, the forces, and the solution.

One would certainly not apply a solution without knowing the problem that the solution addresses. By examining the problem contained in the pattern and comparing it to one's situation, one can perform a simple heuristic to see if the pattern is relevant. If one is faced with solving the same problem that is addressed in the pattern, then the pattern might be worth further investigation. The Canonical form presents the problem

explicitly. The Gang of Four form does not do so; instead the problem is usually presented in the section entitled “Motivation”.

Simply knowing the problem that a pattern solves does not facilitate a truly informed decision. There may be assumptions or other particulars to the situation in which the problem arises; if one’s context is different than the one in which the pattern was observed, the pattern might not be applicable. The context of a pattern should describe these assumptions explicitly, and give any other details that are relevant to explain the context in which the problem-solution pair applies. The Canonical pattern form provides an explicit section labelled “Context”. The Gang of Four form usually includes this information in the “Motivation” and “Applicability” sections.

By definition, a pattern must include a solution. The format of the solution can vary according to the type of pattern. A solution in a software design pattern may require different presentation than a solution in an architectural pattern, for example. The Canonical form makes an explicit “Solution” section mandatory. In addition, this style provides optional sections such as “Resulting Context” to describe the state of one’s design after implementing the pattern and “Rationale” to describe why the solution is a good one; both of these sections include elements of the solution. In the Gang of Four style, the solution is described in the sections entitled “Structure”, which includes a diagram showing the class relationships, “Participants”, which describes the responsibility of each participating class, and “Collaborations”, which describes how the participants interact. The Gang of Four form also provides a “Consequences” section that is similar to the “Resulting Context” in the Canonical form but also contains elements of the forces section, described below. Many pattern forms include an optional “Example”

section, where a concrete example of applying the pattern may be given. Software design patterns also frequently contain a “Sample Code” section, which presents and discusses code that may be helpful in implementing the pattern solution.

The last mandatory component of a pattern is a discussion of the forces present therein. Alexander underlines the importance of forces in a pattern, defining a pattern as “a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves” [Alexander 79]. The solution cannot be considered independently of the forces that it resolves, and the way in which they are resolved. Design pattern researcher Jim Coplien notes that

if we understand the forces in a pattern, then we understand the problem (because we understand the trade-offs) and the solution (because we know how it balances the forces), and we can intuit much of the rationale. For this reason, forces are the focus of a pattern [Coplien 96].

Perhaps because of the importance of forces, the Canonical form provides a mandatory section entitled “Forces” which contains an enumerated list of forces present in the pattern. These forces are addressed or balanced in some way by the solution. To emphasize their importance, the author notes that “the most appropriate solution to a problem ... is the one that best resolves the highest priority forces” and advises pattern authors to ensure that “the reader understands why this solution was chosen” [Appleton 00]. To justify the choice of an enumerated list of forces, it is further suggested that “people like to have convenient handles for concepts such as the forces which affect the choice of solution”; a list of forces provides such handles. In addition, the author addresses a common pitfall of many pattern forms: “prose pattern descriptions can be

very pleasing to read but may be hard to use as a reference because the forces are buried in the prose" [Appleton 00].

The Gang of Four form often describes forces in the section labelled "Consequences", along with the resulting context. They, too, indicate the importance of forces in a pattern.

The consequences are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them [Gamma 94].

So although they do not provide an enumerated list of forces, they are advocates of an explicit description of forces in a pattern.

2.1.4 Formalizing Patterns

Chapter three will investigate in more detail the relationship between the forces in patterns and non-functional requirements. It will be shown that to solve the problem of using an automated tool to facilitate pattern selection, a formal specification for pattern representation is required. In particular, given the close relationship we will identify between pattern forces and non-functional requirements, such a specification must clearly and explicitly enumerate the forces present in a pattern. A number of independent efforts have attempted to create a formal specification for representing patterns. Some of these efforts will be examined, with particular attention paid to how each of these efforts addresses the issue of representing pattern forces.

A.H. Eden has put considerable effort into examining how patterns may be formally specified [Eden 98a, Eden 98b]. The primary goal of his system, LePUS, is to capture the structure of pattern solutions; it does little to address the forces present in a pattern. This research is typical of most existing research on formal pattern specification.

At least one formalization technique does exist which focuses on the structure of the pattern itself, rather than of the implementation of the solution. [Deugo 01] describes a method of specifying patterns using XML. This technique addresses pattern forces explicitly by providing XML tags corresponding to the sections present in a particular pattern form (for example, the “Consequences” section of a Gang of Four pattern, or the “Forces” section of a pattern written in the Canonical form). Thus it is possible to do detailed keyword searches by section, or to vary the pattern presentation. This effort, however, does not adequately address the extraction of pattern forces. As chapter three will discuss, individual forces must nearly always be manually extracted from the pattern in a consistent and machine-readable way to achieve our goal. The specification described in [Deugo 01] provides machine-readable structure, but only to the extent that the forces are already made explicit in the pattern’s original form.

2.2 Non-Functional Requirements

Characteristics of a software system can be divided into two groups. The first group describes aspects of functional behaviour of the system. These characteristics relate to “what the system does” [Chung 99, p. 1]. The second group describes “global requirements” such as “development or operational costs, performance, reliability, maintainability, portability, robustness and the like” [Chung 99, p. 1]. The components of

this second group have been referred to as “constraints”, “goals”, “extra-functional requirements”, “non-behavioural requirements”, and “non-functional requirements” by various authors [Chung 99, p. 1]. They will be referred to as non-functional requirements, or NFRs, herein to emphasize their distinction from functional requirements.

To understand this important difference, consider a specification for a software system that states the system must compute the square roots of numbers, and must do so quickly and accurately. The computation of square roots is a functional requirement because it describes what the system does. The requirements of speed and accuracy are non-functional because they are not related to the computational output of the software but rather to constraints under which the system must perform its functions.

Despite the importance of non-functional requirements, they are not well understood. Lawrence Chung et al. describe their neglect:

Surprisingly, non-functional requirements have received little attention in the literature and are definitely less well understood than other, less critical factors in software development. As far as software engineering practice is concerned, they are generally stated informally during requirements analysis, are often contradictory, difficult to enforce during software development and to validate, when the software system is ready for delivery [Chung 99, p. 2].

Initially, an NFR may be thought of as a “goal”. In its common usage, a goal indicates something to which a label of “satisfied” or “unsatisfied” can be applied. These labels tell us in an absolute sense whether or not the goal is accomplished in all circumstances. However, it must be understood that NFRs have a subjective component to their nature; there are no absolute measurements for the accomplishment of many of these goals. To aid in this distinction, Chung et al. introduce the term “softgoal” to indicate that these goals do not necessarily have “*a priori*, clear-cut criteria of satisfaction” [Chung 99]. Rather than “satisfied”, they use the term “satisficed”, which

refers to “satisfying at some level a variety of needs” [Chung 99]. Now, instead of a black-and-white indicator of whether a goal has been accomplished, there exist labels ranging from satisfied to denied, which also denote the degree to which a softgoal is satisfied or denied. We will adopt this terminology.

2.2.1 Softgoal Interactions

Non-functional requirements in the form of softgoals are a representation of desirable attributes of a software system. However, independently they are of little use; what is needed is information about how these softgoals affect one another, and how they are affected by design decisions during the development process. These interactions are termed “contributions” [Chung 99]. Chung et al. introduce terminology that indicates the type of contribution one softgoal has on another. Initially, these contributions may be thought of in terms of positive or negative effects; a softgoal can either help or hurt another softgoal. In addition to the direction (positive or negative), the contribution can indicate the degree as either partial or complete. Using all possible combinations of direction and degree, [Chung 99] introduces the following terms for contribution types:

- *Makes* provides sufficient positive support.
- *Breaks* provides sufficient negative support.
- *Helps* provides partial positive support.
- *Hurts* provides partial negative support.
- *Some+* provides an unknown amount of positive contribution (either equivalent to *Makes* or *Helps*).

- *Some*- provides an unknown amount of negative contribution (either equivalent to *Breaks* or *Hurts*).
- *Unknown* provides some contribution in an unknown direction (either positive or negative) and degree (partial or sufficient).

These contribution types indicate the effect of one softgoal on another. However, there are many cases where more than one softgoal taken together form a relationship with another softgoal. To handle these cases, [Chung 99] introduces notation for *AND* and *OR* contributions. These contributions group multiple child softgoals and describe their contribution to a single parent softgoal. In the case of the *AND* contribution, the parent is satisfied only if each of the children is satisfied. In the case of the *OR* contribution, the parent is satisfied if one or more of the children is satisfied.

For each of the contributions, there follows naturally a corresponding softgoal label [Chung 99, p. 75]. This label indicates whether a softgoal is satisfied or denied, and the degree of satisfaction or denial. Possible labels are:

- *Satisfied* indicates the softgoal is sufficiently satisfied.
- *Denied* indicates the softgoal is considered unsatisfactory, or unsolvable.
- *Weakly Satisfied*, also denoted as "*W+*", indicates partial but not necessarily sufficient satisfaction of the softgoal.
- *Weakly Denied*, also denoted as "*W-*", indicates partial but not necessarily sufficient denial of the softgoal.
- *Unknown*, also denoted as "?", indicates that the state of the softgoal is not known.

- *Conflict* indicates that the state of the softgoal is not known because it has received conflicting contributions from children.

Chung recognizes that sometimes it is beneficial to describe supporting evidence or justification for a force. “An important premise of the Framework is that design decisions should be supported by well-justified arguments or *design rationale*.” [Chung 99, p. 33]. They introduce claim softgoals as a way of encapsulating this information. A claim softgoal refers to a statement. This statement may be used as an argument for or against something. In their literature, Chung et al. use claim softgoals to lend support to a force that one NFR exerts on another.

2.2.2 Evaluating Softgoal Interdependency Graphs

Chung et al. suggest a method of formalizing softgoals and their interdependencies. “To help a developer address non-functional requirements, the NFR Framework represents softgoals and their interdependencies in softgoal interdependency graphs (SIGs)” [Chung 99, p. 48]. These graphs consist of softgoals as the graph nodes, and contributions as the edges. They use this representation as part of an algorithm to deduce the effects of some softgoals on others: “To determine if top-level requirements are met, an evaluation procedure is offered” [Chung 99, p. 48].

To understand how a graph of softgoals may be evaluated, one must first consider the set of labels (*Satisfied*, *Denied*, *Weakly Satisfied*, *Weakly Denied*, *Unknown*, and *Conflict*), and the set of possible contribution types (*Makes*, *Breaks*, *Helps*, *Hurts*, *Some+*, *Some-*, and *Unknown*). [Chung 99] defines a functional mapping of an input label and an input contribution type to an output label as shown in Table 1.

Child	Breaks	Some-	Hurts	?	Helps	Some+	Makes	Equals
Denied	W^-	W^-	W^-	?	W^-	W^-	Denied	Denied
Conflict	Conflict	Conflict	Conflict	?	Conflict	Conflict	Conflict	Conflict
?	?	?	?	?	?	?	?	?
Satisfied	Denied	W^-	W^-	?	W^-	W^-	Satisfied	Satisfied

Table 1: Mapping Contribution Types to Labels

The first thing to notice about this graph is that it does not consider W^+ or W^- as input labels. The reasons for this decision will become apparent as the evaluation procedure is discussed. In addition, simple rules are defined for the *AND* and *OR* contributions. For these contribution types, a simple ordering of labels is defined (with denied being the smallest and satisfied being the greatest). The rules for *AND* and *OR* contributions defined in [Chung 99] are:

If offspring₁ AND ... AND offspring_n MAKES parent
Then label(parent) = min(label(offspring_i))

If offspring₁ OR ... OR offspring_n MAKES parent
Then label(parent) = max(label(offspring_i))

The evaluation procedure consists of two steps. In the first step, the contributions of each child softgoal toward a parent softgoal are collected. These contributions are referred to as “individual contributions”. They are calculated easily according to the above table and the simple rules for conjunction and disjunction.

At this point there is a known set of individual contributions for a softgoal. “The second step of the evaluation procedure combines them into a single label” [Chung 99, p. 76]. After collecting the individual contributions into a “bag” (a set that retains the number of instances of each entry) some rules are applied to discover the output label that should be applied to the softgoal. Some of these rules are fairly simple, and “decisions can be made automatically, by following a procedure. This can be done without developer intervention since no uncertainty is introduced” [Chung 99, p. 77]. An example

of such a rule is that if the bag consists entirely of *Satisfied* labels, the output label will be *Satisfied*.

In the case where a softgoal receives both positive and negative contribution, the evaluation procedure does not automatically compute a label. Instead, “the developer can use expertise about NFRs, the domain and development, as well as other knowledge, in order to resolve conflicts” [Chung 99, p. 77]. This conflict resolution involves combining the multiple individual contributions into a single output label.

In this second step, [Chung 99] states that $W+$ and $W-$ contributions are first combined into one or more *Satisfied*, *Denied*, *Conflict*, or *Unknown* contributions. In this way, the transition graph does not need to include entries for $W+$ and $W-$. However, they describe “a possible extension to retain such values as outputs of the second step, and inputs to the first step” [Chung 99, p. 75]. These extensions are stated informally and are incomplete. Such extensions would aid in the accuracy of the softgoal interdependency graph, as it would increase the granularity of the results and allow for non-absolute labels. This would be consistent with their earlier justification of the use of the term “softgoal” rather than “goal”.

The evaluation procedure may begin when at least one node in the interdependency graph has been given an initial label. Using the two steps above, this label may be iteratively propagated to other nodes in the graph. The evaluation procedure can be considered complete when all nodes in the graph have labels, or when the nodes that are of interest have labels. The output of the procedure is the labels that the nodes possess.

2.2.3 Cataloguing NFRs

Throughout [Chung 99, Gross 01], the authors suggest that catalogues of non-functional requirements may be created and reused.

Refinement methods and correlations can be defined, collected and organized into *catalogues*. Catalogues are then available for sharing, reuse and adaptation, within the same application contexts and across different ones. This helps alleviate time-consuming and often difficult searches for know-how. [Chung 99, p. 89]

They suggest that a great deal of information about non-functional requirements and their relationships is applicable across a wide range of domains. This information needs to be compiled only once, and can be reused in a variety of contexts.

Certain non-functional requirements seem to be pervasive in the field of software development. Chung et al. list efficiency, usability, correctness, flexibility, reusability, and eight other attributes, and claim that they “apply to all software systems ... however, additional requirements may apply for special classes of software” [Chung 99, p. 3]. They go on to present a catalogue of 158 requirements; this lists includes many of the attributes previously described by [Boehm 76, Keller 90, Sommerville 92].

Chung notes that “different people use different terminologies; this can make it harder to use given categorizations without customizing them” [Chung 99]. Indeed, even with similar terminologies, the definition of each non-functional attribute is likely to differ. However, this does not mean that the creation of a reusable catalogue should be abandoned; it simply means that care must be taken in ensuring consistency of terms and definitions within such a catalogue. The IEEE standard for non-functional requirements [IEEE] has English definitions of several of the important attributes, and is worthy of investigation as a starting point to such a reusable catalogue.

2.3 From Patterns to Non-Functional Requirements

The forces in a design pattern have much in common with the study of non-functional requirements. In general, a force describes the impact of one thing on another. In the case of patterns, forces describe the impact of one possible course of action on some relevant attribute of the resulting system. In this way, forces may be thought of as an “if A then B ” relationship, where A is the course of action, and B is the affected attribute. The attribute described must be either functional or non-functional in nature, and as Daniel Gross and Eric Yu describe, “non-functional requirements are pervasive in descriptions of design patterns” [Gross 01].

It is the non-functional variety that is of interest here, as it represents a direct link from the field of patterns to the field of non-functional requirements: the non-functional attribute in the force may be represented as a node in a softgoal interdependency graph.

In a pattern, one or more forces are presented, and the solution balances or addresses these forces in some way. Some attributes are traded off as being more important than others, and the solutions justifies the design choices it offers in relation to the balanced forces. As such, some forces will be satisfied and some will not, and the solution should indicate whether a force is satisfied. The non-functional attribute in the force may be thought of as a node in the softgoal interdependency graph. Likewise, its satisfaction state may be thought of as its initial label (recall that the evaluation procedure requires that at least one graph node has an initial label).

In [Gross 01], they propose “a systematic treatment of NFRs in descriptions of patterns and when applying patterns during design” [Gross 01]. This approach to

“organizing, analyzing, and refining non-functional requirements can provide much support for the structuring, understanding and applying of design patterns during design” [Gross 01]. Their suggestions include clarifying the role that NFRs play in patterns, and showing the relationships among the goals in a graph.

They see the potential applications of combining pattern knowledge with non-functional requirement catalogues:

Our approach recognizes the need for “applicability conditions” to make the application of design knowledge (such as those captured in design pattern) more selective, and context dependent, when applied during system design... Further work needs to be done to explore how applicability conditions are best specified and utilized when applying patterns. [Gross 01, p. 21].

In addition, they hint at the benefits of creating reusable catalogues of patterns and non-functional requirements in this context:

With more explicit structure, patterns can be retrieved more easily from a richer catalogue. There can be more dimensions for indexing, accessing and navigating the catalogues. Future work will focus on how to index, access and navigate such catalogues and how such catalogues could serve as a basis for a knowledge base, for storing and guiding retrieval of design know-how that addresses NFRs and related tradeoffs during design. [Gross 01, p. 22].

Both of these suggestions are interesting, as the authors recognize the practical implications of the relationship between patterns and non-functional requirements. These suggestions will be addressed in chapter three.

2.4 Extensible Markup Language (XML)

Any attempt to formalize patterns, non-functional requirements, and forces in a machine-readable manner must begin by choosing a format for representing this information in a structured way. The Extensible Markup Language (XML) is “the

universal format for structured documents and data on the Web” [W3Ca]. The structure provided by XML “makes it easy for a computer to generate data, read data, and ensure that the data structure is unambiguous” [W3Cb]. XML is an open standard that operates on any platform, and is well supported [W3Cb]. Development and deployment are helped by the wide availability of tools for creating, reading, writing and processing XML documents, coupled with the flexibility XML [Apache]. XML has also been used by [Deugo 01] for formally specifying patterns. For the reasons cited above, XML is a good choice for formally representing patterns, non-functional requirements and forces in preparation for their processing by an automated tool.

2.5 Eclipse

The choice of a development language and platform for a software tool is influenced by many factors. Time of development, cost, extensibility, reusability and portability are high on the list of such important goals. With these in mind, we investigate the Eclipse Project [Eclipse]. Eclipse is a “universal tool platform” and an “open extensible IDE” written in Java. It provides easy integration of third-party plug-ins and is particularly well suited to the creation and deployment of software development tools. Its extensible XML-based plug-in architecture provides an easy way to contribute actions to the workbench, the main Eclipse user interface window. In addition, it is expected that many future software development tools will be built using Eclipse technology; as such, if a tool was written as an Eclipse plug-in, it will be available to the users of such development tools as an integrated part of their environment.

Eclipse provides a rich set of classes that eliminate many of the issues related to user interface and resource management, as well as a development environment in which the tool may be written.

2.6 Summary

Patterns describe a solution to a problem in a particular context. In addition, a pattern describes how a set of conflicting forces is balanced by the solution. The forces present in a pattern may be represented in many forms, ranging from prose to enumerated lists, and may appear in various sections of the pattern depending on the pattern form. Forces are critical to an understanding of the pattern, as they describe the implications of the solution, yet they are often misunderstood or ignored. Although pattern forces describe the direct impact of design decisions on some attributes, other attributes are affected in other ways that cannot be deduced simply by reading the pattern.

Various attempts at formally specifying patterns have been mostly geared towards the formalization of pattern solutions, from which functional applications of the pattern may be automatically or semi-automatically generated. These approaches do not address the applicability of patterns, or the impact of a pattern solution on attributes of a system.

Non-functional requirements are attributes of software systems that describe not what the system does but how it does it. Rather than a strict black and white view of whether a requirement is satisfied, NFRs may be thought of in terms of various degrees of sufficient to insufficient satisfaction. The requirements and their interactions can be assembled into a graph. If this graph is given an initial value indicating whether one or

more of the requirements is satisfied, the graph may be iteratively evaluated to determine how other requirements in the graph are affected.

NFRs and their interactions can be collected into domain-independent or domain-specific catalogues. These may be reused for a variety of problem-solving methods during design and development. It should be noted, however, that there is no complete catalogue of NFRs, nor do there exist universal definitions for even the most common NFRs.

XML was identified as a applicable technology for formally representing structured data, such as patterns, non-functional requirements and forces. An automated tool could operate on catalogues specified in this machine-readable format. The Eclipse Project was determined to be a useful platform on which a software tool could be built quickly and efficiently.

The forces present in a design pattern are characterized by the inclusion of non-functional requirements in their descriptions. An overlap exists between the fields of design patterns and non-functional requirement. The forces in a pattern describe direct impacts of design decisions on non-functional requirements, but are missing important information about resulting indirect effects on other requirements. Non-functional requirement research introduces a method for cataloguing and assembling related non-functional requirements into graphs to determine the impact of one on another. Researchers have noted the implications of this overlap but little work has been done to explore the implications. In this thesis, we will address this overlap by demonstrating that by combining patterns with NFRs, a developer can gain valuable information about the indirect implications of a pattern.

3. Design and Methodology

We will show that it is possible to determine the impact of using a pattern on non-functional requirements in a largely automated way. A pattern contains within it a set of forces that are balanced by the pattern's solution. These forces describe the direct impact of the pattern's solution on certain NFRs. This information may be combined with a graph of NFRs and their relationships. From this, it is possible to determine the impact of the pattern on NFRs contained in the graph.

In order to automate the processing of pattern forces, these forces must first be clearly identified and expressed in a machine-readable format. The identification of forces in a pattern cannot be automated, due to the wide variety of expression afforded by human language. The manual identification of forces in a pattern will make these forces accessible to automated tools, and also contribute to a better understanding of the consequences of applying the pattern. Such a manual identification needs to be made only once for a particular pattern.

With the goal of automated processing of patterns and non-functional requirements, a machine-readable format for expressing pattern forces is necessary. Similarly, a format for expressing non-functional requirements and their relationships is also needed. The evaluation procedure in [Chung 99] can be used as a basis for a larger algorithm that incorporates a graph of non-functional requirements as well as pattern forces.

The evaluation procedure in [Chung 99] requires human intervention in many cases. Using extensions described in [Chung 99] and other improvements introduced in this section, it is possible to automatically resolve many of these cases without input from

the user. For the remainder, we introduce a flexible strategy of heuristics and user interaction that minimizes the user's interaction with the algorithm.

3.1 Outline

We begin by investigating the structure of pattern forces. Particular attention is given to the relationship between pattern forces and non-functional requirements. The issues surrounding the identification and extraction of forces from a pattern are investigated. A machine-readable storage format for non-functional requirements, patterns and forces is presented. The feasibility of creating a domain-independent catalogue of non-functional requirements and their relationships is examined.

An algorithm is introduced which incorporates the evaluation procedure in [Chung 99] and operates on a combined graph of non-functional requirements and patterns. An investigation of the impacts of conflicts during evaluation is presented. Extensions to the evaluation procedure are introduced which deal with these conflicts efficiently. The implications of these extensions are thoroughly examined.

A proof of concept automated tool is developed, incorporating the presented storage format and evaluation algorithm. The design goals and decisions are discussed, and various technical aspects of the resulting code are examined.

3.2 The Structure of Pattern Forces

In order to automate the processing of pattern forces, it is first necessary to understand the structure of these forces. In the pattern form suggested by [Appleton 00],

an explicit “Forces” section precedes the pattern solution. In this form, the forces are typically listed in a bulleted list, which makes it easy to separate one from another. There is often a lack of precision when representing patterns. This is presumably due to the inherent difficulty of recognizing the forces in a pattern. This issue is exacerbated by the fact that forces are often represented using non-structured prose. In fact, many pattern forms represent forces in this way [Gamma 94, Cunningham]. To properly perform any sort of analysis of pattern forces, it is necessary to consider a pattern form that distinguishes individual forces from one another.

Forces can be represented in different ways, but generally share one of two main underlying structures. The first structure is as follows:

If <proposed decision> then <[non-]functional consequence of decision>.

For example,

If you use linked lists, then the solution will be slower.

There are three important things to note about this structure.

- The consequence may be either functional or non-functional in nature. In the above example, the consequence, slower software, is non-functional.
- It describes a positive or negative impact of the proposed design or implementation decision on the functional or non-functional attribute mentioned in the consequence. Above, speed is negatively impacted by the decision to use linked lists.
- The proposed decision described by the force is not necessarily the solution proposed by the pattern; it may simply be offered to illustrate why the actual

solution was chosen instead of this proposed decision. In this case, the proposed decision is using linked lists.

The second common structure of forces in design patterns is as follows:

If <non-functional attribute> then <non-functional consequence>.

For example,

If the solution is slow, then people will not use it.

It is important to distinguish this type of force from the first type described. In the first, a particular design decision was made. In the second, no design decision was made, but a causal link is established between one attribute, the non-functional attribute of speed, and another, the non-functional attribute of usability.

Forces conforming to the first of these forms will be referred to as *primary forces*. This is because they represent forces that flow directly from functional design decisions. Forces of the latter form will be called *secondary forces*. They are brought about indirectly.

Both force structures share the more general structure:

If <condition> then <consequence>.

The first variable, the condition, will be referred to as the *antecedent*, and the second variable as the *consequent*.

To trace the effects of primary forces on functional or non-functional attributes, we will build a graph of a pattern's forces. The key to doing this is to recognize when consequent of one force is equal to the antecedent of another. This usually cannot be automated because different people write them in different ways. Such identification must be made by a careful analysis of the pattern. However, if this analysis is performed

once for a pattern, the results can be stored in a way that is accessible to automated tools.

Consider the following two forces in a fictitious pattern:

Using linked lists will make the solution slower.

The slower the solution is, the less usable it will be.

Using the manual analysis procedure, one would first attempt to identify each force as matching either the primary or secondary form. Next, one would rewrite them to more easily identify the antecedent and consequent of each. So,

If you use linked lists, then the solution will be slower.

If the system is slow, then it will be less usable.

The first force is identified as being primary in nature, and the latter as secondary. In this simple example, the reader has undoubtedly also recognized that the consequent of the first force and the antecedent of the second are in fact identical (i.e. "Speed"). Having recognized this fact, the forces may be again rewritten in a way that will make the relationship between them clearer:

If you use linked lists, then speed is negatively affected.

If speed is negatively affected, then usability is negatively affected.

Now the first and second forces can be chained together in a graph. This logical resolution allows transitive statements to be deduced. In this case, traversing this graph will demonstrate the following:

If you use linked lists, then the solution will be less usable.

This may seem obvious in this example. However, in a typical pattern containing many forces, the interactions between the forces may not be so obvious. Extracting this

information will not only make the forces accessible to an automated tool, but could contribute to a better understanding of the consequences of applying the pattern.

It is noted at this point that the graph of pattern forces and their effects on each other bears a striking resemblance to the force hierarchies described in [Chung 99]. This is important because the similarities between pattern forces and the force hierarchies of non-functional requirements will allow us to integrate pattern forces into existing force hierarchies and gain a deeper understanding of the impact of the pattern forces.

3.3 Identifying Forces in Patterns

With an understanding of the structure of pattern forces, we can address the problem of how they can be identified. An immediate problem in need of a solution is that patterns are most typically written partially or completely in prose-style English. The requirements of an automated tool mandate that forces be represented in a uniform machine-readable way. For example, if “Usability” is present in a force, it must be identifiable as the same “Usability” that is present in a catalogue of non-functional requirements. Due to this requirement for precision, care must be taken when identifying a pattern’s forces.

The forces present in a pattern written in the Canonical form are relatively easily identified, as this form explicitly provides a section for a list of forces balanced by the pattern’s solution [Appleton 00]. Even in this case, it is still a fairly manual process to extract the forces in such a way that they may be used in conjunction with a non-functional requirements catalogue. However, as patterns are by their nature reusable templates, a catalogue of pattern forces may be built up in the same way as a catalogue of

NFRs, and reused in many different domains and problem sets. As Daniel Gross and Eric Yu note, “NFRs that are explicitly represented in design patterns aid in better understanding the rationales of design, and make patterns more amenable to structuring and analysis” [Gross 01].

Some patterns forms [Gamma 94, Cunningham] do not require that forces be explicitly identified or enumerated within the pattern text. This does not mean that they do not exist; indeed, forces are critical in any pattern [Alexander 79]. The forces are often intermingled with the solution, the resulting context, or even in the motivation section of the pattern.

3.4 Cataloguing Non-Functional Requirements

It would not be feasible to construct a domain-dependent catalogue of non-functional requirements every time one wished to analyse the impact of a particular pattern. It is clearly necessary to collect a reusable catalogue of these requirements and their relationships which will apply to any query.

One problem with the existing catalogues of NFRs in [Chung 99, Keller 90, Boehm 76, Sommerville 92] is that they omit details on how the requirements affect each other. Many requirements impact each other in positive or negative ways that are constant regardless of the domain. A classic example in software development is the relationship pair:

Time negatively impacts Space.

Space negatively impacts Time.

Since the NFRs and their relationships are often domain-independent, such a catalogue of NFRs should capture these well-known interactions.

Another problem is that the existing catalogues of NFRs are not accessible to an automated tool. It is clear that a machine-readable catalogue is needed.

3.5 Specifying Machine-Readable Catalogues of NFRs and Patterns

In order to construct an automated software tool that utilizes a catalogue of requirements, forces, and patterns, an important pre-requisite must be satisfied. A method of formally specifying requirements, the relationships between them, and design patterns in a machine-readable and machine-parsable format is critical to the success of such a tool. Although many storage formats could have produced a working design, XML [W3Ca] was chosen for a number of reasons.

- It combines structure with clarity [W3Cb].
- It is open, royalty-free and standards-based [W3Cb].
- There is wide availability of tools for reading and writing this file format.

3.5.1 Representing Requirements

The most basic element of our catalogue is a functional or non-functional requirement. This simple element can be described by a unique identifier and a corresponding description. The requirement element can be described abstractly in XML with the following Document Type Description (DTD):

```

<!ELEMENT requirement (
    description
)>
<!ATTLIST requirement
    id CDATA #REQUIRED
>

```

As this element will be used to describe both functional and non-functional requirements, an example of each follows. The first of these is the XML markup for the non-functional requirement of security, including the simple definition given in [Merriam-Webster]:

```

<requirement id="Security">
    <description>The quality or state of being
        secure</description>
</requirement>

```

In the case of functional requirements found in patterns, the value of the identifier attribute is less important, as it need not match other entries in the catalogue; its uniqueness, however, is just as important:

```

<requirement id="1">
    <description>Use linked lists</description>
</requirement>

```

Notice that the identifier in this case might simply number the requirements found in a pattern, or provide a more useful identification depending on the context. The choice is left to the creator of the catalogue.

3.5.2 Representing Forces

As previously discussed, a force generally describes the impact or effect of one or more requirements on another requirement. It has already been discovered that the forces present in design patterns generally conform to one of two types. Primary forces describe

the impact of a functional requirement on a non-functional requirement, and secondary forces describe the impact of a non-functional requirement on another. Our specification should include this information. A third type of force is defined by [Chung 99]. This force, called a “claim force”, provides supporting evidence or justification for another force. In the case of a claim force, rather than having an antecedent and consequent, as primary and secondary forces do, it consists of a description of the claim and a list of one or more forces that are justified by the claim force.

Also important in pattern forces is whether or not a primary force is satisfied by the pattern; this must also be captured in our definition. As we will demonstrate, this information is crucial because it will determine the initial state of the functional nodes in the first stage of the evaluation procedure. In addition, it is important to be able to describe the type of impact or contribution the antecedent of a primary or secondary force makes upon the consequent. Thus, it is convenient to adopt the terminology defined in [Chung 99] and allow the following contribution types:

- *Makes* provides sufficient positive support; if the antecedent is satisfied, the consequent can be satisfied.
- *Breaks* provides sufficient negative support; if the antecedent is satisfied, the consequent can be denied.
- *Helps* provides partial positive support; if the antecedent is satisfied, partial positive support is given to the consequent.
- *Hurts* provides partial negative support; if the antecedent is satisfied, partial negative support is given to the consequent.

- *Some+* provides an unknown amount of positive contribution (either equivalent to *Makes* or *Helps*).
- *Some-* provides an unknown amount of negative contribution (either equivalent to *Breaks* or *Hurts*).
- *Unknown* provides some contribution of an unknown “sign” (either positive or negative) and “extent” (partial or sufficient).
- *Equals* indicates that the antecedent and the consequent are equivalent.

Finally, the definition must allow for conjunctive and disjunctive combinations for a force antecedent in primary and secondary forces, as described in [Chung 99]. This can be achieved by providing a definition that allows for one or more forces in the antecedent, and an optional tag describing the antecedent type as either a conjunction or a disjunction.

With all of these requirements in mind, we define the force element in XML using the following DTD:

```
<!ELEMENT force (
    description,
    ((antecedent, consequent) |
    ((justifies)+)
)>
<!ATTLIST force
    id CDATA #REQUIRED
    type (
        primary | secondary | claim
    ) #IMPLIED
    satisfied (
        true | false
    ) #IMPLIED
    contribution (
        makes | breaks | helps | hurts |
        some-plus | some-minus | unknown | equals
    ) #IMPLIED
```

Some elements used above must be further defined. Both the antecedent and the consequent must refer to existing requirement definitions. The “requirement-reference” element is thus defined as a general-purpose element to refer to an existing requirement by identifier:

```
<!ELEMENT requirement-reference (
)>
<!ATTLIST requirement-reference
      id CDATA #REQUIRED
>
```

The “antecedent” element must be able to encapsulate one or more requirements, so its DTD is defined as follows:

```
<!ELEMENT antecedent ((
      requirement-reference
)+)>
<!ATTLIST antecedent
      type CDATA #REQUIRED
>
```

The “consequent” element is more straightforward:

```
<!ELEMENT consequent (
      requirement-reference
)>
<!ATTLIST consequent
      type CDATA #REQUIRED
>
```

For claim forces, the “justifies” element indicates which force is justified by this force. Its DTD follows:

```
<!ELEMENT justifies (
)>
<!ATTLIST justifies
      id CDATA #REQUIRED
      force CDATA #REQUIRED
>
```

With all the elements defined, some examples follow. They illustrate usages of the force DTD and its sub-elements to describe a primary force, a secondary force, a secondary force with a disjunction, and a claim force.

The first example illustrates our earlier force describing the negative impact of linked lists on speed.

```
<force
  id="force1"
  type="primary"
  satisfied="true"
  contribution="hurts">
  <description>Using linked lists will make the
    solution slower</description>
  <antecedent>
    <requirement-reference id="LinkedLists"/>
  </antecedent>
  <consequent>
    <requirement-reference id="Speed"/>
  </consequent>
</force>
```

The second example illustrates our earlier force that claims that if the system is slow, it will be less usable.

```
<force id="force2" type="secondary" contribution="helps">
  <description>If the solution is slow,
    then people will not use it.</description>
  <antecedent>
    <requirement-reference id="Speed"/>
  </antecedent>
  <consequent>
    <requirement-reference id="Usability"/>
  </consequent>
</force>
```

Our third example shows how a conjunction can be used to represent the claim that integrity, confidentiality and availability together satisfy security.

```
<force id="force3" type="secondary" contribution="makes">
  <description>Security is satisfied by its
    three constituent parts.
  </description>
  <antecedent type="conjunction">
    <requirement-reference id="Integrity"/>
    <requirement-reference id="Confidentiality"/>
    <requirement-reference id="Availability"/>
  </antecedent>
  <consequent>
    <requirement-reference id="Security"/>
  </consequent>
</force>
```

The final example illustrates a claim force that justifies the above force relating to security.

```
<force id="force4" type="claim">
  <description>Security can only occur if the system has
    integrity, confidentiality and availability.
  </description>
  <justifies id="1" force="force3"/>
</force>
```

3.5.3 Representing Patterns

The last element that needs to be formally specified is the design pattern. Creating a formal specification for design patterns is an interesting problem due to the fact that there are so many different pattern forms, many of which contain large amounts of prose that is unsuited for immediate markup. As well, copyright issues may prevent parts of many patterns from being copied into such a machine-readable format. However, for the purposes of the automated tool the only interesting portions of a pattern are the title, used for identification purposes, and the forces, for which a previously described formal specification already exists. As will be described later, the forces provide initial node labelling for the softgoal interdependency graph, and the title must be used so that the user can associate the results with a particular pattern. The problem, context and solution are not important in this situation, as they do not determine how the pattern will affect the softgoal interdependency graph, so they do not need to be included in the DTD. Obviously these portions of a pattern are of great importance when one is actually applying the pattern. Therefore, the relatively simplistic definition specified by the following DTD will be used:

```

<!ELEMENT pattern (
    forces
)>
<!--ATTLIST pattern
    name CDATA #REQUIRED
-->
<!ELEMENT forces (
    (force)+
)>

```

3.6 The Evaluation Algorithm

With machine-readable representations of the domain data, all the prerequisites for constructing an automated tool that operates on this data are satisfied. The general algorithm for deducing the impact of a pattern on a set of non-functional requirements is the next step. Such an algorithm will use as its basis the evaluation procedure described in [Chung 99]. It will now be explained how this evaluation procedure will be used as a part of the larger algorithm which incorporates patterns and pattern forces. This larger algorithm will help to attain the stated goal of determining the fitness of a pattern in relation to a set of target NFRs. The input to the algorithm is as follows:

- A set of patterns, P
- A set of NFRs, NFR
- A set of relationships, R
- A list of target NFRs, T

The output to the algorithm is as follows:

- For each pattern P , a set of labels for each NFR in T

The algorithm follows these general steps:

1. For each pattern P_n , create a graph G_n by taking the union of P_n , NFR and R .

2. Evaluate graph G_n using the evaluation procedure described in [Chung 99], with extensions described below.
3. Create a list L_n , containing the label for each requirement in T , after the evaluation procedure just performed.
4. Return as output all lists L . These lists contain the labels of each requirement in T .

3.7 Conflicts During Evaluation

The evaluation procedure described in [Chung 99] and used as part of the larger algorithm described earlier has some issues that need to be addressed. In particular, [Chung 99] describes that when a node receives both positive and negative satisfaction from its children as individual contributions, that node is said to be in conflict. The solution they present is that the developer must manually resolve conflicts when they occur, and such resolution is mandatory to the completion of the evaluation algorithm. Such a manual step is contrary to the stated goal of producing a tool that will evaluate force graphs automatically. Before an automated tool may be written, it must be determined how these conflicts should be resolved.

The conflict resolution itself is very necessary. Different children affect their parents in different ways, and equating two contributions of the same type, or saying that two contributions of similar size but opposite direction cancel each other, is obviously not a precise solution. Only a human with problem knowledge can realistically step in and resolve the conflict. There are really two solutions to this problem.

- The first solution involves manual resolution of conflicts by interaction with the user, as Chung suggests. This is the most accurate solution, but is less than ideal

for at least two reasons: such resolution would be time- and effort-consuming, especially given a large catalogue of NFRs, and it is contrary to the goal of automatic processing.

- The second solution involves some (possibly configurable) algorithms to automatically resolve conflicts when they occur. This is the more ideal solution from the point of view of an automated tool, as it minimizes user interaction. However, in this case, it must be apparent that the results will not be as exact or precise as results obtained when the user with knowledge of the problem manually resolves the conflicts.

After a careful examination of the evaluation rules, a flexible evaluation strategy will be introduced. This strategy automatically and accurately resolves many conflicts that are not automatically resolved by [Chung 99]. The conflicts that are not automatically resolved may be resolved either by utilizing user input during evaluation, or by relying on user-configurable heuristics. These heuristics quickly, and without user interaction, deduce a likely resolution to the conflict. The consequences of the automatic resolution of conflicts are discussed in detail, and the envisioned usage of these two solutions is presented.

3.8 Extensions to the Evaluation Procedure

The evaluation procedure as presented in [Chung 99] is not immediately suitable for automated processing. In its presented form, it precludes the propagation of weak labels from children to parents. Such propagation would not only provide more accurate results, it would also reduce considerably the number of cases where user input is

required. Even if these labels were propagated, however, there still exist a number of situations where a conflict must be resolved in order to propagate a label. These situations must be addressed in order for an automated tool to successfully evaluate such a graph.

For simplicity, the evaluation procedure in [Chung 99] combines one or more weak labels (such as *Weakly Satisficed*) into one stronger label (such as *Satisficed*). The decision on how to do so is left to the user, implying a significant amount of user interaction. They describe also a modification to their procedure that would allow for the propagation of weak labels. This modification should be used, as it would not only cut down on the number of necessary user interactions, but would also propagate more accurate results.

[Chung 99] informally describes new rules for the evaluation procedure that incorporate this extension. These rules are formally presented below, and others are introduced to form a complete transition graph. The original transition graph is repeated in Table 2 for convenience.

Child	Breaks	Some-	Hurts	?	Helps	Some+	Makes	Equals
Denied	W+	W+	W+	U	W-	W-	Denied	Denied
Conflict	Conflict	Conflict	Conflict	U	Conflict	Conflict	Conflict	Conflict
U	U	U	U	U	U	U	U	U
Satisficed	Denied	W-	W-	U	W+	W+	Satisficed	Satisficed

Table 2: Transition Table

The following rules are also defined for conjunction and disjunction:

*If antecedent₁ AND ... AND antecedent_n MAKES consequent,
Then label(consequent) = min(label(antecedent_i))*

*If antecedent₁ OR ... OR antecedent_n MAKES consequent,
Then label(consequent) = max(label(antecedent_i))*

In the above rules, no mention is made of the case where an antecedent is weakly satisfied or weakly denied. In fact, it is stated in [Chung 99] that such rules are unnecessary, since the values are eliminated during the second phase of the evaluation procedure. As an afterthought, extensions to the algorithm are mentioned which would allow these values to be propagated. Since this would allow for more accurate and flexible propagation of information through the evaluation procedure, these extensions will be adopted as described, and add other extensions as necessary, justifying each.

First of all, [Chung 99, p. 79] states “a $W+$ offspring and a $SOME+$ contribution could result in $W+$ for the parent”. Likewise, “a $W+$ offspring and a $SOME-$ contribution could result in $W-$ for the parent”. In addition, “UNKNOWN always contributes U ” [p. 73]. The equality rules also hold [p. 79]. So the table may be populated as shown in Table 3.

Child	Breaks	Some-	Hurts	?	Helps	Some+	Makes	Equals
$W+$		$W-$		U		$W+$		$W+$
$W-$				U				$W-$

Table 3: Initial Extensions

There is also an implied symmetry involving the $Some+$ and $Some-$ contributions, from which it may be inferred that a $W-$ offspring and a $Some+$ contribution could result in a $W-$ for the parent, and a $W-$ offspring and a $Some-$ contribution could result in $W+$ for the parent. This is shown in Table 4.

Child	Breaks	Some-	Hurts	?	Helps	Some+	Makes	Equals
$W+$		$W-$		U		$W+$		$W+$
$W-$		$W+$		U		$W-$		$W-$

Table 4: $Some+$ and $Some-$ Contributions for $W-$

In the previous rules defined, “it is interesting to note that the entries for the $SOME-$ and $HURT$ contributions are the same. Likewise, $SOME+$ and $HELP$ have

identical entries” [Chung 99, p. 74]. The inferred additions to the chart are shown in Table 5.

Child	Breaks	Some-	Hurts	?	Helps	Some+	Makes	Equals
W+		W-	W-	U	W+	W+		W+
W-		W+	W+	U	W-	W-		W-

Table 5: Defining Helps and Hurts Transitions

If *Makes* propagates satisfaction of a child to the parent, it would logically follow that weak satisfaction of a child *Makes* weak satisfaction of the parent. This is not specifically stated in [Chung 99] since they did not formally consider the propagation of weak labels, but given the rule that *Makes* propagates a satisfied label from a child to a parent, this seems logical. A similar reasoning exists for the propagation of weak denial from the child to the parent by means of a *Makes* rule, as shown in Table 6.

Child	Breaks	Some-	Hurts	?	Helps	Some+	Makes	Equals
W+		W-	W-	U	W+	W+	W+	W+
W-		W+	W+	U	W-	W-	W-	W-

Table 6: Defining Makes Transitions

In the same way that a denied offspring with a *Breaks* contribution type cannot necessarily contribute full satisfaction to the parent, only weak satisfaction, a weakly denied offspring must also contribute weak satisfaction to the parent [Chung 99, p. 75]. In fact, consider the case where a denied offspring contributes a weak satisfaction to the parent by means of a *Breaks* contribution. In that case that although it contributes weak satisfaction, “the developer may change the [weak satisfaction] to [satisfaction], depending on the circumstances” [Chung 99, p. 75]. Thus it appears there is an even stronger case that in this scenario, weak satisfaction is a valid label propagation.

Last, consider the case of weak satisfaction and a *Breaks* contribution type. If satisfaction of the offspring under the *Breaks* contribution causes the denial of the parent,

it seems obvious that weak satisfaction of the offspring will cause only the weak denial of the parent, rather than complete denial. The complete chart is presented in Table 7.

Child	Breaks	Some-	Hurts	?	Helps	Some+	Makes	Equals
W+	W-	W-	W-	U	W+	W+	W+	W+
W-	W+	W+	W+	U	W-	W-	W-	W-

Table 7: The Complete Transition Table

Now we have a complete set of rules for the determination of the individual contribution of an antecedent to a consequent, with which one is able to complete step one of the algorithm. The second step requires a more thorough examination, as it involves the combination of all the individual contributions for a given consequent into a single label. This step often involves the manual intervention of the user under the basic rules described in chapter two, depending on the type and number of individual contributions for a particular consequent. The basic rules also neglect to consider the extensions mentioned above, namely that weak satisfaction and weak denial are valid labels for a consequent. Therefore each rule will be examined individually to see how it may be modified, if necessary, to apply to the situation where the extensions are used. The basic rules were defined quite informally; presumably this corresponds to the uncertainty with which they were applied, and the manual intervention possible in each step. Attempts will be made to define them more formally, which will more easily correspond to the automated method of evaluation desired.

“Typically, W+ values alone in a bag would result in Satisfied or Undecided” [Chung 99]. Given our extension that allows *W+* as a label, it is now also conceivable that *W+* values alone might result in *W+*. Given this (more accurate) possibility, we discard the *Undecided* state, which existed only because it was the only realistic option other than *Satisfied* in the absence of *W+*. This means that *W+* values alone in the bag will result in

either *Satisfied* or W^+ . The situations in which each case might occur will be examined more closely.

“ W^- values alone would result in *Denied* or *Undecided*” [Chung 99]. Using similar reasoning to the previous rule, one should allow for the possibility of W^- values alone resulting in W^- , and discard the possibility of using *Undecided*. Therefore, W^- values alone will result in either *Denied* or W^- . The precise situations will be examined later.

“A mixture of W^+ and W^- values would typically result in *Satisfied*, *Denied*, or *Undecided*” [Chung 99]. This is a very vague rule that is quite subjective; their implementation requires the developer to resolve this conflict. Methods of balancing W^+ and W^- in some way to minimize the developer input will be discussed.

We add the following rules, which require little justification:

“One or more *Satisfied* contributions, in combination with zero or more W^+ contributions, and no negative contributions (neither W^- nor *Denied*), will result in *Satisfied*”. Given that there is at least one *Satisfied* contribution, and no negative contribution to detract from its satisfaction, *Satisfied* will be propagated to the parent in this case.

“One or more *Denied* contributions, in combination with zero or more W^- contributions, and no positive contributions (neither W^+ nor *Satisfied*), will result in *Denied*”. The reasoning is similar to the previous rule.

“One or more *Satisfied* contributions, in combination with zero or more W^+ contributions, and zero or more W^- contributions, and no *Denied* contributions, will result in *Satisfied*”. Even though there are weak negative contributions, the fact that there is a

single contribution that is strong enough on its own to cause undisputed satisfaction of the parent should override the weak negativity.

Likewise, “one or more *Denied* contributions, in combination with zero or more *W-* contributions, and zero or more *W+* contributions, and no *Satisfied* contributions, will result in *Denied*”. The reasoning is similar to the previous rule.

The next rules are less obvious, and as such more flexible solutions should be considered to accommodate differing subjective viewpoints on their interpretation.

If one or more *Satisfied* contributions are present, along with one or more *Denied* contributions, there are several possible strategies. The first is that if the *Satisfied* contributions outnumber the *Denied* contributions, the *Satisfied* label will be propagated; conversely, if the number of *Denied* contributions is greater, then *Denied* will be propagated. The second possibility considers the case of coincidental satisfaction and denial to be contradictory, and must consult the user. In another scenario, the user might be prompted if the number of *Satisfied* contributions equals the number of *Denied* contributions. Still another possibility is that the user has configurable preferences, set before the evaluation procedure begins, which might determine the following things:

- Which label shall prevail in the case of a tie between the number of *Satisfied* and *Denied* contributions?
- How many more *Satisfied* contributions must there be than *Denied* contributions in order for the parent to be *Satisfied*? (A similar potential preference asks how many more *Denied* contributions must there be than *Satisfied* in order for the parent to be *Denied*).
- Under which circumstances should the user be consulted to resolve a conflict?

- How many $W+$ contributions, if any, may be combined into a single *Satisfied* contribution? Likewise, how many $W-$ contributions may be combined into a *Denied*?

In the case where only $W+$ and $W-$ contributions are present, many of the same possibilities arise. One may choose to propagate whichever of $W+$ and $W-$ has the most contributions; in the case of a tie, one might ask the user. Similar preferences to those mentioned above may exist. Namely,

- How many $W+$ contributions may be combined into a single *Satisfied*; likewise, how many $W-$ contributions will combine into a *Denied*?
- What is the threshold difference between the number of positive and negative contributions, below which the user is prompted to resolve the conflict?
- Which label will prevail in the case of a tie?

Notice that if the first rule applies and $W+$ contributions are combined into a *Satisfied* (or $W-$ into a *Denied*), then the situation matches the previous case, where a mix of *Satisfied*, $W+$, $W-$ and *Denied* labels appear, and the corresponding rules apply.

3.9 Implications of Impreciseness

The extensions introduced previously involve the possible use of heuristics to resolve conflicts. Since the conflict resolution mechanism described in [Chung 99] is presented as an inherently subjective procedure, it is obvious that the labels assigned by the automated algorithm may not coincide exactly with those that a given developer might assign, if presented with the same force hierarchy. Even more, changing the values of the user preferences, as described above, may result in different labels for some nodes,

even when run on the same force hierarchy. The implications of this impreciseness must be examined before any claim of its usefulness can be asserted.

Regardless of the level of automation assigned to the evaluation of the graph, it must be possible for the user to arbitrarily resolve some conflicts manually if he or she chooses. This may be done after the fact, forcing the evaluation to be partially or completely re-run with the user-specified resolutions in place. Any automated tool must allow this functionality, even if the user's manual resolution contradicts one of the built-in rules. The reason for this is that contributions may differ slightly in different situations or domains, and the exact value may be known only by the user. Additionally, weak satisfaction and weak denial contributions may have drastically different impacts on their parents, since the exact magnitude is subjective and not captured by their label.

To understand how to reconcile this possible impreciseness, one must understand the general use cases for this algorithm. There are two envisioned usages for the automated algorithm, in the context of this thesis. The first of these is to determine, for a large input set of design patterns, a relative ranking of the patterns according to how well they support a specified set of NFRs. The second is to determine, more specifically, how well a single pattern, or a very select group of patterns, supports the given NFRs; that is, how the given NFRs are affected by the pattern.

In the first scenario, the developer is looking only for a rough ranking, not preciseness. Having a large set of patterns ranked imprecisely is very useful; it is obvious that the patterns closer to the top of the list probably conform more closely to the given NFRs than do those near the bottom. For patterns close together in the list, it is understood that they should have relatively similar characteristics, and that there may be

small margins of error, as such rankings are in actuality subjective. In this situation, running the evaluation algorithm over a large set of patterns, it will be advantageous to use a strategy that involves more automatic resolutions of conflicts, and fewer manual cases which must be posed to the user. The result is that the user can arrive quickly at a large set of coarse results. It is not envisioned in any way that the user should automatically choose the top pattern in the list, as the patterns may vary wildly in ways other than their conformance to a set of NFRs. These variances may make one pattern suitable and another unsuitable, regardless of their rankings. In addition, two patterns may solve completely different problems, making either or both useless to the task at hand. The results of this query should simply be viewed as a coarse-grained set of results, from which a finer set of results may be derived.

The second scenario involves the user looking for more detailed information about a particular pattern, or a small set of patterns. Presumably, the user has in this case either already run a more coarse search, as described above, or has started out with a particular pattern in mind (perhaps he or she has read about this particular pattern, or knows it to be a good solution to the problem at hand). The small set of patterns might be formed from a small subset of the patterns input into the coarse query, for example, or the user may want to determine how well the Visitor pattern [Gamma 94] conforms to the requirement of reusability. It is envisioned that in this case the user will choose a set of preferences that affords him or her greater control over the resolving of conflicts; this procedure will not be too onerous, since there is a much smaller search set for which to resolve conflicts. In addition, changes may be made even to those conflict resolutions that are automatically determined by the rules of the algorithm; these changes would reflect

domain-specific knowledge. After these changes have been made, and the evaluation algorithm is re-run with the manual resolutions in place, the results are to be considered much more accurate.

3.10 Automated Tool Design Goals

The automated tool is designed to include a flexible set of preferences that will allow the user to have significant control over the level of interaction required to resolve conflicts during the evaluation. When running the tool, the user should select a set of preferences, some of which are described previously, which govern the behaviour of the tool under certain circumstances. A sufficient set of preferences is critical to the usability of the tool, so the chosen set should afford the user significant flexibility. To think in terms of the NFRs described earlier, flexibility and usability are of high importance for this tool.

As the design of the tool will involve the creation of classes representing requirements, forces, patterns, and graph nodes, it is noted that these domain classes could be used again in the context of other tools that process or manipulate requirements, forces, patterns or graphs of such objects. In addition, the XML parsers could be reused by any tool that chooses to conform to the DTDs described here. Again, thinking in terms of the NFRs, reusability is also important.

The described conflict resolution mechanism relies on a defined set of preferences to determine its behaviour. It may be the case that one might like to replace this conflict resolution mechanism with another, or change the set of preferences and their meanings. This should be easily accomplished, and the design of the tool should allow for a simple

method of replacing the conflict resolution mechanism with another. Thus, the requirement of extensibility is desired.

3.10.1 Design Decisions

The programming language chosen for the software tool is Java [Gosling 96].

There are a number of reasons for choosing Java:

- Its object-oriented features facilitate encapsulated, reusable classes of domain objects
- Using its included collection classes reduces work.
- Its interface mechanism allows for multiple interface inheritance [Gosling 96].
- It is widely used, which increases the reusability of the reference tool.
- Tools for its development are readily available.
- Most importantly, familiarity with this language will allow for a more rapid prototyping phase.

The Eclipse Project is a “universal tool platform” and an “open extensible IDE” [Eclipse]. It provides easy integration of third-party plug-ins and is particularly well suited to the creation and deployment of software development tools. Eclipse was chosen as the platform on which the automated tool runs. Its extensible XML-based plug-in architecture provides an easy way to contribute actions to the Eclipse user interface. In addition, it is envisioned that many future software development tools will be built using Eclipse technology; as such, the tool being developed here will be available to the users of such development tools as an integrated part of their development environment.

Xerces [Apache] is a freely available standards-based XML parser framework.

The Java implementation of the Xerces parser framework is included in the Eclipse platform as a plug-in, and as such is available to all Eclipse plug-ins to use. It implements both a SAX and a DOM parser, and conforms to popular XML standards and proposals. It is actively developed and well documented. Xerces was chosen as the parser framework the tool uses to read patterns, requirements and forces stored in the DTDs described previously.

3.10.2 Designing the Domain Objects

The first step involves the creation of a small set of domain classes and interfaces to represent requirements and forces. These classes and interfaces are briefly described in the following sections, and an important subset of their Application Programming Interface (API) is explained. The first group of classes and interfaces represents requirements and forces. The second provides a way of constructing a graph of these requirements and forces by storing related objects in a catalogue and creating graph nodes.

3.10.3 Design of Requirements and Forces

The first group of classes and interfaces is sufficient to represent the various types of requirements (functional and non-functional) and forces (primary, secondary, and claim).

The interface `IRequirement` represents a functional or non-functional requirement. It consists of a unique identifier, a type that designates it as either functional or non-functional in nature, and a description. The Java code declaring this interface is as follows:

```
public interface IRequirement {
    /**
     * Returns the requirement's identifier.
     */
    public String getId();
    /**
     * Returns the requirement's description.
     */
    public String getDescription();
    /**
     * Returns the type of this requirement.
     * Valid values are:
     * IRequirementConstants.FUNCTIONAL
     * IRequirementConstants.NON_FUNCTIONAL
     */
    public int getType();
}
```

The interface `IForce` is the abstract super-interface of all interfaces that represent forces. This interface defines a unique identifier and textual description for the force. The Java code follows:

```
public interface IForce {
    /**
     * Returns the force's description.
     */
    public String getDescription();
    /**
     * Returns the force's identifier.
     */
    public String getId();
}
```

The interface `ISecondaryForce` is a sub-interface of `IForce`, and represents a secondary force. As such, it provides methods for retrieving the antecedent and

consequent of the secondary force, as well as retrieving the contribution type. The Java code is as follows:

```
public interface ISecondaryForce extends IForce {
    /**
     * Returns the force's antecedent.
     */
    public Antecedent getAntecedent();
    /**
     * Returns the force's consequent.
     */
    public RequirementReference getConsequent();
    /**
     * Returns the force's contribution type.
     * Valid values are:
     * IRequirementConstants.MAKES
     * IRequirementConstants.BREAKS
     * IRequirementConstants.HELPS
     * IRequirementConstants.HURTS
     * IRequirementConstants.SOME_PLUS
     * IRequirementConstants.SOME_MINUS
     * IRequirementConstants.UNKNOWN
     * IRequirementConstants.EQUALS
     */
    public int getContribution();
}
```

The possible values of the contribution mirror those values described as contribution types in [Chung 99], and those present in the XML declaration for forces.

The interface `IPrimaryForce` is a sub-interface of `ISecondaryForce`, and as such inherits an antecedent, consequent, and contribution type. The only change in API is the addition of a method to answer whether the force is satisfied:

```
public interface IPrimaryForce extends ISecondaryForce {
    /**
     * Returns whether the force is satisfied.
     */
    public boolean isSatisfied();
}
```

The remaining force type is a claim. Rather than having an antecedent and consequent, a claim force uses one requirement to lend justification to one or more other forces, rather than to another requirement. The declaration of `IClaimForce` follows:

```
public interface IClaimForce extends IForce {
    /**
     * Returns the requirement which describes the claim.
     */
    public RequirementReference getClaim();
    /**
     * Returns the forces which this claim justifies.
     */
    public ForceReference[] getJustifies();
}
```

The classes `RequirementReference` and `ForceReference`, as used in interface declarations above, are simply lightweight locators for requirements and forces, respectively. They contain sufficient information to unambiguously locate the target, such as the identifier of the requirement or force. For the purposes of this design, the use of the identifier is sufficient, but in a more widely distributed scheme, additional information might encode the server, path, Uniform Resource Locator (URL), or other location-based information.

While a consequent of a primary or secondary force is a single requirement, the antecedent may be more complex. In addition to the case of a single requirement, conjunctions and disjunctions of multiple requirements must also be permitted. To this end, the interface `IAntecedent` declares the API for an antecedent:

```

public interface IAntecedent {
    /**
     * Returns the type indicating the relationship
     * between the requirements.
     * Valid values are:
     * IRequirementConstants.CONJUNCTION
     * IRequirementConstants.DISJUNCTION
     */
    public int getType();
    /**
     * Returns the requirements composing this
     * relationship.
     */
    public RequirementReference[] getRequirements();
}

```

3.10.4 Design of Force Graphs

The following classes and interfaces allow the creation of complex graphs of requirements and forces. Java provides multiple inheritance in interfaces, as well as implementation of multiple interfaces by a single class; as such, it is possible that design can be simplified by making a requirement node and the requirement itself the same object. Similarly, the same object may represent a force node and the force itself.

The `IRequirementNode` interface encapsulates a single requirement. It provides sufficient API to retrieve all forces for which the requirement is an antecedent, all forces for which the requirement is a consequent, and all claim forces that use this requirement as a claim. The declaration of `IRequirementNode` follows:

```

public interface IRequirementNode {
    /**
     * Returns the requirement that this node
     * encapsulates.
     */
    public IRequirement getRequirement();
}

```

```

    /**
     * Returns all forces for which this
     * requirement is an antecedent.
     */
    public IForceNode[] getAntecedentForces();
    /**
     * Returns all forces for which this
     * requirement is a consequent.
     */
    public IForceNode[] getConsequentForces();
    /**
     * Returns all forces for which this
     * requirement is a claim.
     */
    public IForceNode[] getClaimForces();
}

```

The **IForceNode** interface encapsulates a single force. It provides sufficient API to retrieve the underlying force, as well as any claim forces that justify this force. In the case of claim forces, it also provides access to forces that this force justifies. The declaration of **IRequirementNode** follows:

```

public interface IForceNode {
    /**
     * Returns the force that this node encapsulates.
     */
    public IForce getForce();
    /**
     * Returns all forces that this force justifies,
     * or null if this force is not a claim force.
     */
    public IForceNode[] getJustifiesForces();
    /**
     * Returns all forces that justify this force.
     */
    public IForceNode[] getJustifiedByForces();
}

```

The **ICatalogue** interface provides a single point of access to a group of requirements and forces. One may add forces and requirements to it, and in doing so,

force nodes and requirement nodes are created, which may later be retrieved and queried for their inter-relationships.

```
public interface ICatalogue {
    /**
     * Adds the given requirement to the catalogue.
     */
    public void add(IRequirement requirement);
    /**
     * Adds the given force to the catalogue.
     */
    public void add(IForce force);

    /**
     * Returns all force nodes in the catalogue.
     */
    public IForceNode[] getForceNodes();
    /**
     * Returns all requirement nodes in the catalogue.
     */
    public IRequirementNode[] getRequirementNodes();
    /**
     * Returns the requirement node matching the given
     * identifier, or null if no such node exists.
     */
    public IRequirementNode getRequirementNode(String id);
}
```

3.10.5 Design of the Evaluation Algorithm

The class `Evaluator` is the entry point for the evaluation of a graph of requirements and forces rooted at a pattern. The evaluator may be called repeatedly in a loop; this is useful in the case where the evaluation should consider each of a number of patterns as the roots of a graph. The declaration of its API method is as follows:

```
public class Evaluator {
    /**
     * Computes the labels for the given target
     * requirements, while considering a graph defined
     * by the given catalogue and rooted at the given
```

```

        * pattern.
        *
        * Returns an array of labels corresponding to the
        * targets.
        */
        public int[] computeLabels(Pattern pattern, Catalogue
catalogue, Requirement[] targets) {
            ...
        }
    }
}

```

The evaluator serves three main functions. First, it contains generic logic that will traverse the catalogue and ensure that each relevant requirement has a label. It determines which requirements are relevant by considering the graph created by the catalogue, with the given pattern as the roots, and selecting only those nodes that are reachable from the forces in the given pattern. Second, it implements the previously discussed set of rules by which individual contribution types are determined, and by which labels are automatically propagated in some circumstances. Third, in the case where a label cannot be determined automatically, it delegates to a conflict resolver.

It has been decided previously that any evaluation design must provide the flexibility to use a varying strategy in the cases where the label cannot be determined by the known rules. To such an end, the evaluator references a conflict resolver – an object to which the decision of label computation is delegated. Prior to computing the labels, one may provide the evaluator with such a conflict resolver, which is an instance of the interface `IConflictResolver`. The relevant API method of `Evaluator` is as follows:

```

/**
 * Sets the object which is responsible for the
 * resolution of labels which cannot be determined
 * automatically.
 */
public void setConflictResolver(IConflictResolver resolver)
{
    ...
}

```

The sole responsibility of the conflict resolver is to provide a label for a requirement node that cannot automatically be determined. Its interface definition contains a single method:

```

public interface IConflictResolver {
    /**
     * The label for the given node could not be
     * automatically resolved. Perform necessary actions
     * to determine its label.
     */
    public int computeLabel(IRequirementNode node,
        IForceNode[] forces, int[] contributions);
}

```

This delegation of an algorithm to a plug-able delegate is an example of the Strategy pattern [Gamma 94].

To determine individual contribution types, the evaluator maintains a static table that maps a child node label to a new node label, given a contribution. This table contains exactly the same data as the one described earlier in this section.

The design of the computation of labels is described by the following pseudo-code:

```

// Compute the labels for each requirement in targets
computeLabels(pattern, catalogue, targets) {
    // Find which nodes are reachable from the pattern
    for each force in pattern,
        // Use a visitor to collect reachable nodes
        visitor.visit(force)
    // Keep a table of reachable nodes and their labels
    for each reachable node found by the visitor,
        labels.put(node, initial label of UNKNOWN)
    // Find a label for each of the target nodes
    for each requirement in targets,
        computeLabel(target, labels)
    return labels for each requirement in target
}

```

The above method implements the API method of the evaluator. After first determining which requirement nodes are reachable from the given pattern, it creates a table of all relevant nodes and their labels. A node's label is initially unknown, and will be assigned in the computeLabel method. The label is then computed for each of the target requirements, and the labels are returned to the caller.

```

// Compute the label for the given target
computeLabel(target, labels) {
    // Assign labels to all relevant antecedent
    // requirements, recursively
    for each force for which the target is a consequent,
        for each requirement in the force's antecedent,
            if the requirement is relevant,
                computeLabel(requirement, labels)
    // Determine the individual contributions for forces
    for each force for which the target is a consequent,
        compute the individual contribution type from:
            1. labels of the antecedent's requirements
            2. the contribution of the force
    // Compute the label given the contribution types
    computeLabel(target, individual contributions)
}

```

This recursive method will determine the label for a given requirement node by first determining the label for all nodes which affect the given node. After all relevant

labels are assigned, the individual contribution types are assessed for each force. This corresponds to step one of the evaluation procedure described in [Chung 99]. Then, given a collection of individual contribution types, the method forwards to a third method to actually compute the label (step two of the evaluation procedure from [Chung 99]).

```
// Compute a label given a collection of contribution types
computeLabel(target, contributions) {
    if there is one or more satisfied and no denied
    contributions,
        return satisfied
    if there is one or more denied and no satisfied
    contributions,
        return denied
    if there are only unknown contributions,
        return unknown
    return resolver.computeLabel(target, contributions)
}
```

This method computes the label for a node given a collection of individual contributions. It first runs through a small set of trivial cases, for which labels may be automatically determined. It then delegates responsibility for label computation to the conflict resolver.

The automated tool described here includes two conflict resolvers, corresponding to the two general cases described earlier. The first resolver is largely automated and requires no user input. It operates based on a set of built-in heuristics to very quickly determine a large number of labels. The results from this resolver are very coarse, and should be refined when the number of candidate patterns has been reduced. The second resolver is very fine-grained; aside from the cases described above in which the evaluator may automatically determine the label, this resolved prompts the user to resolve each conflict that is encountered. This will result in a great deal of user interaction but will lead to very satisfactory results. Each of these resolvers is now described briefly.

The class `CoarseResolver` implements the first resolver. Its only user-definable value is the number of weakly satisfied contributions that will be combined into a single satisfied contribution. Likewise, the same number determines how many weakly denied contributions will be combined into a single denied contribution. It begins by combining weak satisfaction and weak denial contributions into satisfied and denied contributions, respectively. After this step, the only remaining contribution types should be satisfied, denied, and unknown. `CoarseResolver` then determines the label by choosing whichever of satisfied and denied contributions are more plentiful, choosing satisfied in the case of a tie. These steps require no user input during the evaluation. Prior to the evaluation, the only input required is setting of an initial value for the number of weak contributions to combine into a stronger one; a default value is present.

The class `FineResolver` implements the second resolver. Rather than rely on automated heuristics, this resolver prompts the user to determine the label for any questionable nodes. Figure 1 shows the resolver prompting the user to resolve a conflict using a dialog window.

Figure 1: Conflict Resolution

Although the time required to obtain results is higher, and the number of user steps is much greater, the results will be more accurate, as previously described.

3.11 User Interface

The user interface for the evaluation tool is quite simple. A wizard dialog prompts for any required information prior to the evaluation, and the results are displayed on the standard output stream. The user interface will now be briefly described, and screen shots of the elements will be included.

The wizard consists of three pages. The first page prompts for the location of the catalogues. The second page asks the user to select which requirements are to be considered target requirements. The final page selects which resolver should be used to resolve conflicts, as described earlier.

The input to the evaluation tool consists of three catalogues. The three catalogues contain the requirements, the forces, and the patterns, respectively. The catalogues are in the XML format previously described. The first page of the wizard is pictured in Figure 2.

Input Catalogs
Please specify the location of the input catalogues.

NFR Catalog:

Pattern Catalog:

Force Catalog:

Figure 2: Specifying Input Catalogues

After reading the input catalogues, the wizard prompts the user to select one or more of the requirements as targets. These are the requirements to which labels will be assigned and reported. Page two of the wizard provides a checkbox table to obtain this information from the user, and is pictured in Figure 3.

NFR Selection
Please select one or more target requirements.

- ☐ Extensibility
- ☐ Reusability
- ☒ Development Cost
- ☐ Simplicity
- ☐ Verifiability
- ☒ Traceability

< Back Next > **Finish** Cancel

Figure 3: Selecting Target NFRs

Finally, the user is prompted to select which resolver should be used during the evaluation procedure to deal with conflicts. Page three of the wizard provides a “combo box” in which the user selects either the coarse resolver or the fine resolver depending on his or her needs. This page is depicted in Figure 4.

The image shows a software dialog box titled "Resolution Strategy Selection". Below the title is the instruction "Please select a conflict resolution strategy." There is a list box containing two items: "Coarse-grained Resolver" and "Fine-grained Resolver". The "Coarse-grained Resolver" item is currently selected. At the bottom of the dialog box, there are four buttons: "< Back", "Next >", "Finish", and "Cancel". The "Finish" button is highlighted with a thicker border.

Figure 4: Selecting Resolution Strategy

After the user selects the "Finish" button from the wizard, the evaluation procedure is begun. If the user has selected the fine-grained resolver to handle conflict resolution, the conflict resolution dialog will appear each time user input is required during evaluation. When evaluation is complete, the results are printed to standard output.

The user interface described here is quite simple; it provides a minimum of functionality required to run the verification and validation tests described in chapter four, and to obtain results which may be used to lend support to the claims made in this chapter. A number of improvements could be made to the user interface to transform it into a more user-friendly and powerful evaluation tool. These improvements are discussed in chapter five.

3.12 Summary

We have investigated and described the structure of pattern forces, and recognized that the identification and extraction of forces requires great care. Such work needs to be performed only once; the results may be reused. With this understanding, a storage format for patterns, non-functional requirements and forces was introduced, which facilitates machine-readable storage for these elements.

Once all the domain elements are in a machine-readable catalogue, they can be processed by an automated tool. The evaluation procedure in [Chung 99] serves as a basis for an algorithm that uses pattern forces as an initial node labelling for the softgoal interdependency graph. This algorithm provides a label to each graph node, which indicates whether each softgoal was satisfied. This information indicates indirect, non-explicit effects of applying a pattern.

A sample design of an automated tool was described; this tool uses the above methodology and algorithm to automate graph processing. This tool will provide results that will be examined in depth in chapter four.

4. Results

This chapter will demonstrate the evaluation procedure described in chapter three, and obtain results from the software tool. A sample catalogue will be constructed, and three patterns will be introduced which impact requirements in this catalogue. Results will be obtained manually through a deductive reasoning process, and a second set of results will come from the software tool. The two sets of results will be compared and analyzed. The sample data will be modified in order to introduce a conflict, and new results will be obtained and analyzed. The chapter concludes with a summary of the results and a number of observations.

4.1 A Sample Non-Functional Requirement Catalogue

To demonstrate the evaluation algorithm and achieve results, we must first construct a sample catalogue of non-functional requirements. The contents of this catalogue will be chosen in such a way that they demonstrate the full potential of the evaluation procedure in a manageable amount of space. To begin to construct the catalogue, we will identify as the two target requirements “Development Cost” and “Traceability”. Formal definitions of these terms are not necessary to the argument. These requirements are shown in Figure 5.

Figure 5: Defining Target Requirements

If a component is designed in such a way that it is reusable, this may reduce development costs by reusing the component in more than one place in the system, alleviating the need for redundant implementations. In this way, we consider “Reusability” to help “Development Cost”, shown in Figure 6.

Figure 6: Reusability Helps Development Cost

It is also true that the extensibility of a system or of a component of the system can lead to reusability. Thus, “Extensibility” helps “Reusability”, as shown in Figure 7.

Figure 7: Extensibility Helps Reusability

Traceability is often considered to refer to the ability to track incremental changes to a system or to parts of a system. This attribute might help to ensure that the impact of certain changes is well understood during the development process. If the functionality of

a component is easily verifiable, for example, by use of an automated testing suite, then traceability is helped. We therefore state that “Verifiability” helps “Traceability” and add the interaction to the graph. The new graph is shown in Figure 8.

Figure 8: Verifiability Helps Traceability

It should be fairly obvious that the simpler a component is, the easier it should be to verify. Thus, to complete our sample catalogue of non-functional requirements, we note that “Simplicity” helps “Verifiability”, as shown in Figure 9.

Figure 9: Simplicity Helps Verifiability

The resulting catalogue documents some well-understood relationships between certain non-functional requirements. For the purposes of this example, we will imagine

the development of a software system for which development cost and traceability are the two crucial non-functional requirements. As such, it is desirable that one knows beforehand the effects that changes to the system will have on these attributes.

4.2 Object Creation Patterns

Consider that during the development of the fictitious system, the developer encounters the problem of object creation. That is to say that the developer needs to provide the client with the ability to instantiate a number of classes in the system. There are a number of ways that object creation can be achieved, and this is certainly not a new problem; thus the developer decides to consult relevant patterns for a solution to the problem that coincides with the constraints of the system.

Gamma et al. devote a chapter of their book to creational patterns. "Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed and represented" [Gamma 94, p. 81]. The five patterns they describe help to solve the problems associated with object creation and class creation in two ways. "First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together" [Gamma 94, p.81].

Although creational patterns can be used in complementary ways, Gamma et al. note that some of them may be competitors. In this example, three of these patterns will be evaluated as solutions for the problem of object creation. After a brief description of each pattern, the forces will be extracted from them in a way suitable for automated evaluation.

4.2.1 Abstract Factory

The Abstract Factory pattern is intended to “provide an interface for creating families of related or dependent objects without specifying their concrete classes” [Gamma 94, p. 87]. Gamma et al. make a number of claims in the “Consequences” section of the pattern, which should be mined for possible impacts on non-functional requirements.

“It makes exchanging product families easy” [Gamma 94, p. 89]. The example given is one of swapping the implementation of a widget family without changing any of the calling code. This implies that Abstract Factory provides some positive support for adaptability.

“It isolates concrete classes” [Gamma 94, p. 89]. By isolating the clients from the implementation classes and providing access only through the abstract interface, the client code is made more generic, as class names do not appear in the client code but only in the implementation of the factory. Thus Abstract Factory provides positive support for generality.

“It promotes consistency among products” [Gamma 94, p. 90]. This is a fairly simple and concise statement of positive support for consistency.

Supporting new kinds of products is difficult. ... Extending abstract factories to produce new kinds of Products isn't easy... because the AbstractFactory interface fixes the set of products that can be created [Gamma 94, p. 90].

These statements make it clear that while Abstract Factory has good points, these must be balanced with some possibly negative points. In particular, both modifiability and extensibility are negatively impacted by applying this pattern.

4.2.2 Factory Method

The Factory Method pattern defines “an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses” [Gamma 94, p. 107]. The consequences of applying the pattern follow.

It “provides hooks for subclasses” [Gamma 94, p. 109]. When using the Factory Method pattern, the client creates new subclasses that specify the instantiation behaviour. By design, this means that this pattern positively impacts both extensibility and modifiability.

Gamma et al. note, however, “a potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object”. This has a negative impact on simplicity, since the client must deal with subclassing the creator even when a subclass is not otherwise required.

4.2.3 Singleton

The Singleton pattern will “ensure a class only has one instance, and provide a global point of access to it” [Gamma 94, p. 127]. While this pattern has definite functional limitations that must be considered before deciding to use it, it also addresses a number of non-functional requirements.

Singleton provides “controlled access to sole instance” [Gamma 94, p. 128]. This ensures that a client may not arbitrarily create new instances of the class. This has a definite positive impact on controllability. In addition, applying the Singleton pattern

leads to a “reduced name space” [Gamma 94, p. 128]. This is an indication of the positive impact the pattern has on simplicity.

Gamma et al. note that the pattern “permits refinement of operations and representation” [Gamma 94, p. 128], implying a positive impact on extensibility. They also claim that it “permits a variable number of instances” and is “more flexible than class operations”, which indicates positive contribution to flexibility.

4.2.4 Summary of Patterns

Three creational patterns (Abstract Factory, Factory Method and Singleton) were chosen for this example. Their impacts on various non-functional requirements were noted, and these effects are summarized in Table 8.

	Abstract Factory	Factory Method	Singleton
Adaptability	HELPS	UNKNOWN	UNKNOWN
Consistency	HELPS	UNKNOWN	UNKNOWN
Controllability	UNKNOWN	UNKNOWN	HELPS
Extensibility	HURTS	HELPS	HELPS
Flexibility	UNKNOWN	UNKNOWN	HELPS
Generality	HELPS	UNKNOWN	UNKNOWN
Modifiability	HURTS	HELPS	UNKNOWN
Simplicity	UNKNOWN	HURTS	HELPS

Table 8: Summary of Pattern Impacts on NFRs

By extracting the forces from these patterns, we can more clearly make a comparison of the patterns’ direct effects on certain non-functional requirements. But we are limited to a discussion of the requirements that are specified in the patterns. It should be noted that, even in terms of their direct effects, these three patterns offer contradictory support to some non-functional requirements. For example, Abstract Factory hurts modifiability and

extensibility, but Factory Method helps these same two attributes. Also notice that Factory Method hurts simplicity, but Singleton helps it.

4.3 Considering the Indirect Impact of Patterns

The three patterns provide much information about their direct effects on certain non-functional requirements. However, we have previously constructed a catalogue of NFRs and some relationships between them, which shows the impact of one NFR on another. It should be obvious that a pattern that directly affects one NFR in this graph could have an indirect effect on another. It should also be noted that this relationship cannot be deduced by examining the text of the pattern. In the face of two important requirements (development cost and traceability), one should carefully consider the impact of each pattern on these requirements before deciding on a solution.

To accomplish this, each pattern will in turn be attached to the catalogue defined earlier. The indirect impacts of each of the patterns on certain NFRs should then be made obvious.

4.3.1 Indirect Impact of the Abstract Factory Pattern

Consider the impact of applying the Abstract Factory pattern on the target requirements of development cost and traceability, as shown in Figure 10.

Figure 10: Impact of Abstract Factory

Abstract Factory hurts extensibility. Since a positive impact on extensibility has a positive impact on reusability, a negative impact on extensibility negatively affects reusability. Likewise, this negative impact on reusability creates a negative impact on development cost. Therefore, Abstract Factory has an indirect negative impact on development cost.

Examining Figure 10, it can be seen that Abstract Factory has no direct or indirect impact on traceability.

4.3.2 Indirect Impact of the Factory Method Pattern

Referring to Figure 11, the impact of the Factory Method pattern on the target requirements can be deduced.

Figure 11: Impact of Factory Method

Applying the Factory Method pattern helps the attribute of extensibility. Since extensibility helps reusability and reusability helps development cost, it can be seen that Factory Method has an indirect positive impact on development cost.

Factory Method has a negative impact on simplicity. Since simplicity has a positive impact on verifiability, a negative impact on simplicity creates a negative impact on verifiability. Likewise, since verifiability has a positive impact on traceability, a negative impact on verifiability creates a negative impact on traceability. Therefore, Factory Method has an indirect negative impact on traceability.

4.3.3 Indirect Impact of the Singleton Pattern

Considering the third candidate pattern, Singleton, its impact on traceability and development cost can be discovered. These interactions are captured in Figure 12.

Figure 12: Impact of Singleton

Applying the Singleton pattern helps simplicity. Since simplicity helps verifiability and verifiability helps traceability, this means that Singleton indirectly has a positive impact on traceability.

Similarly, Singleton helps extensibility. Since extensibility helps reusability and reusability helps development cost, Singleton has an indirect positive impact on development cost.

4.3.4 Results of Manual Evaluation

Using a chart of the non-functional requirement catalogue and manually tracing the impact of the three candidate patterns, results in Table 9 are achieved.

	Development Cost	Traceability
Abstract Factory	Hurts	Unknown
Factory Method	Helps	Hurts
Singleton	Helps	Helps

Table 9: Impact of Patterns on Target Requirements

It is evident that the Singleton pattern is the only one of the three patterns that has a positive impact on both of the target requirements, and indeed is also the only pattern that does not have a discernable negative impact on either requirement. Singleton is therefore the pattern that is most consistent with the stated non-functional requirements.

The results above are fairly obvious due to the simplicity of the example. This simplicity is not typical of real-world examples, which may have dozens or hundreds of patterns, requirements and forces. In this case, simplicity facilitates a comparison of results achieved manually with results achieved by using the evaluation algorithm. Such a comparison would not be possible in a much larger example, as achieving manual results in such a case is not feasible. The example is suitable for the calculation of manual results for a number of reasons:

- The number of interactions is small. Resolving labels manually is fairly easy if only one or two interactions are present.
- The number of requirements is small. As such, a diagram can be constructed that a human can easily comprehend. If the number of requirements was much larger, as is typically the case, such a diagram would be much more difficult to understand by examining it visually and reasoning deductively.
- The number of candidate patterns is small. Each pattern required a manual traversal of the graph; if the number of candidate patterns was larger, this traversal would have been much more time-consuming.

4.4 Running the Software Tool

With reasonable results obtained manually, these results can be compared with those obtained from the software tool. In order to do this, we first need a machine-readable representation of the three patterns and of the non-functional requirement catalogue. Next, the software tool may be run, and development cost and traceability can be identified as interesting target requirements. The software tool will generate results using the evaluation algorithm, and finally the results will be compared to the results achieved manually.

4.4.1 Creating Machine-Readable Input

In chapter three, a method of specifying patterns, requirements and forces in a machine-readable format is presented. This specification uses XML to formally represent the various attributes of these items such that an automated tool may process them easily. Without such a specification, automated processing is not feasible, as the items are more regularly expressed in prose, which does not lend itself to such analysis. Using this method, we can first generate XML representing the non-functional requirements as follows:

```
<catalogue>
  <requirement id="Extensibility">
    <description/>
  </requirement>
  <requirement id="Reusability">
    <description/>
  </requirement>
  <requirement id="Development Cost">
    <description/>
  </requirement>
  <requirement id="Simplicity">
    <description/>
  </requirement>
```



```

    <requirement id="Verifiability">
      <description/>
    </requirement>
    <requirement id="Traceability">
      <description/>
    </requirement>
  </catalogue>

```

Next, the relationships making up the graph shown in Figure 12 can be represented similarly:

```

<catalogue>
  <force id="F1" contribution="helps">
    <antecedent>
      <requirement-reference id="Extensibility"/>
    </antecedent>
    <consequent>
      <requirement-reference id="Reusability"/>
    </consequent>
  </force>
  <force id="F2" contribution="helps">
    <antecedent>
      <requirement-reference id="Reusability"/>
    </antecedent>
    <consequent>
      <requirement-reference id="Development Cost"/>
    </consequent>
  </force>
  <force id="F3" contribution="helps">
    <antecedent>
      <requirement-reference id="Simplicity"/>
    </antecedent>
    <consequent>
      <requirement-reference id="Verifiability"/>
    </consequent>
  </force>
  <force id="F4" contribution="helps">
    <antecedent>
      <requirement-reference id="Verifiability"/>
    </antecedent>
    <consequent>
      <requirement-reference id="Traceability"/>
    </consequent>
  </force>
</catalogue>

```

Finally, the three candidate patterns can be represented as follows. It should be noted that the descriptions of the functional requirements are empty; this is for simplicity. A complete catalogue would contain textual descriptions not only of the forces but also of

the functional requirement in the pattern that gives rise to the force. This information is not necessary for graph evaluation.

```
<catalogue>
  <requirement id="FR1">
    <description/>
  </requirement>
  <pattern name="Abstract Factory">
    <forces>
      <force
        id="F1"
        type="primary"
        satisfied="true"
        contribution="hurts">
        <description>Supporting new kinds of products
          is difficult</description>
        <antecedent>
          <requirement-reference id="FR1"/>
        </antecedent>
        <consequent>
          <requirement-reference
            id="Extensibility"/>
        </consequent>
      </force>
    </forces>
  </pattern>

  <requirement id="FR2">
    <description/>
  </requirement>
  <requirement id="FR3">
    <description/>
  </requirement>
  <pattern name="Factory Method">
    <forces>
      <force
        id="F2"
        type="primary"
        satisfied="true"
        contribution="helps">
        <description>Provides hooks for
          subclasses</description>
        <antecedent>
          <requirement-reference id="FR2"/>
        </antecedent>
        <consequent>
          <requirement-reference
            id="Extensibility"/>
        </consequent>
      </force>
      <force
        id="F3"
        type="primary"
        satisfied="true"
```

```

        contribution="hurts">
        <description>Client must subclass even if
            they don't want to</description>
        <antecedent>
            <requirement-reference id="FR3"/>
        </antecedent>
        <consequent>
            <requirement-reference id="Simplicity"/>
        </consequent>
    </force>
</forces>
</pattern>

<requirement id="FR4">
    <description/>
</requirement>
<requirement id="FR5">
    <description/>
</requirement>
<pattern name="Singleton">
    <forces>
        <force
            id="F4"
            type="primary"
            satisfied="true"
            contribution="helps">
            <description>More flexible than class
                operations, permits refinement
                of operations and representation
            </description>
            <antecedent>
                <requirement-reference id="FR4"/>
            </antecedent>
            <consequent>
                <requirement-reference
                    id="Extensibility"/>
            </consequent>
        </force>
        <force
            id="F5"
            type="primary"
            satisfied="true"
            contribution="helps">
            <description>
                Implementation is simple
            </description>
            <antecedent>
                <requirement-reference id="FR5"/>
            </antecedent>
            <consequent>
                <requirement-reference id="Simplicity"/>
            </consequent>
        </force>
    </forces>
</pattern>
</catalogue>

```

4.4.2 Obtaining Results from the Software Tool

The software tool is run and the three catalogues presented above are supplied as input, as shown in Figure 13.

Input Catalogs
Please specify the location of the input catalogues.

NFR Catalog:	d:\catalogues\requirements.xml	Browse...
Pattern Catalog:	d:\catalogues\patterns.xml	Browse...
Force Catalog:	d:\catalogues\forces.xml	Browse...

< Back Next > Finish Cancel

Figure 13: Locating the Sample Input

The tool requires a list of target non-functional requirements before it can proceed. All of the requirements in the catalogue are listed as candidates; for this example, development cost and traceability are chosen, as shown in Figure 14.

NFR Selection
Please select one or more target requirements.

- ☐ Extensibility
- ☐ Reusability
- ☒ Development Cost
- ☐ Simplicity
- ☐ Verifiability
- ☒ Traceability

< Back Next > **Finish** Cancel

Figure 14: Selecting Development Cost and Traceability

By selecting the catalogues described above as input for the software tool, and by selecting the requirements of development cost and traceability as target requirements, the tool provides the following output:

```
Labels for pattern Abstract Factory:
  Label for requirement Development Cost is WEAKLY DENIED
  Label for requirement Traceability is UNKNOWN
Labels for pattern Factory Method:
  Label for requirement Development Cost is WEAKLY SATISFICED
  Label for requirement Traceability is WEAKLY DENIED
Labels for pattern Singleton:
  Label for requirement Development Cost is WEAKLY SATISFICED
  Label for requirement Traceability is WEAKLY SATISFICED
```

The results obtained from the software tool are identical to the manual results deduced earlier. Also, the sample catalogue contains no conflicting contributions; therefore, the software tool completes without the need for any manual conflict resolution.

4.5 Resolving a Conflict

The example does not contain any conflicting contributions; therefore the conflict resolution mechanism in the software tool is not exercised. Let us now consider a slight modification to the example that will serve to introduce such a conflict, and observe the results.

Imagine that Abstract Factory hurts the non-functional requirement of coupling. Consider also that coupling hurts reusability. This scenario is pictured in Figure 15.

Figure 15: Introducing a Conflict

In such a way, reusability has conflicting individual contributions when the Abstract Factory method is applied. It receives a negative contribution from extensibility, but positive contribution from coupling. In such a case, the developer must step in and decide what the resulting label for reusability should be. It is clear that the heuristics described in chapter three will not help in this situation; a single positive contribution and

a single negative contribution should be resolved manually. The XML additions to the input catalogues are straightforward and are not shown here.

If the developer resolves the conflict by deciding, based on domain knowledge, that reusability should be helped, then development cost should also be helped. Conversely, if the developer decides that reusability should be hurt, then development cost should also be hurt.

When the software tool detects the conflict using the fine-grained resolver, the developer is prompted to resolve the conflict based on the available information as well as any domain-specific information he or she possesses. In this example, consider that the developer decides that the requirement with conflicting contributions should be given a weakly satisfied label. This is shown in Figure 16.

Figure 16: Extensibility Conflicts with Coupling

The output of the software tool after resolving the conflict in this way is as follows:

```
Labels for pattern Abstract Factory:
    Label for requirement Development Cost is WEAKLY SATISFICED
    Label for requirement Traceability is UNKNOWN
Labels for pattern Factory Method:
```

Label for requirement Development Cost is WEAKLY SATISFICED
Label for requirement Traceability is WEAKLY DENIED
Labels for pattern Singleton:
Label for requirement Development Cost is WEAKLY SATISFICED
Label for requirement Traceability is WEAKLY SATISFICED

The output is the same as in the previous run except that Abstract Factory causes development cost to be weakly satisfied, instead of weakly denied; this is a direct result of the manual conflict resolution, and is exactly what is expected.

4.6 Comparing Results

A small example catalogue of non-functional requirements and some relationships between them was constructed. Three patterns from [Gamma 94] were analyzed, and certain impacts on non-functional requirements were extracted from them. Combining the catalogue and the patterns, the impact of each pattern on two target non-functional requirements was deduced. The manual analysis was quite easy, as the number of requirements and patterns was small, and no conflicting forces were present.

The patterns and the catalogue were specified in XML using the DTDs described in chapter three. They were then provided as input to the software tool. The tool correctly and automatically deduced the same results as were achieved manually.

The example catalogue and patterns were modified to introduce two conflicting forces. Manually analyzing the modified catalogue and patterns clearly indicated that the developer needed to resolve the conflict, which consisted of both positive and negative support for one of the requirements.

The modified catalogue and patterns were provided as input to the software tool. The tool correctly identified the presence of a conflict, and asked the user to resolve the conflict. After the conflict was resolved, the tool produced the expected output.

4.7 Scalability Issues

The sample catalogue presented in this chapter has a small number of patterns, requirements and forces. Since this catalogue is used to lend credibility to the evaluation procedure outlined in chapter three, we should investigate the implications of the size of the catalogue. Of particular interest is whether the evaluation procedure works on larger data sets.

Rather than presenting an improvement on an existing algorithm for determining the indirect impact of patterns on non-functional requirements, chapter three introduces the first known algorithm for doing so. The reviewed literature indicated that there are interesting relationships between the fields of patterns and non-functional requirements, but did not make any attempt to link them in a practical way.

Given the lack of a pre-existing formal method of achieving the results that the evaluation procedure claims to produce, an alternative method of validating its results was needed. The relationships between requirements, when taken individually, are easily understood, as they express simply positive or negative, partial or complete support of one requirement for another. When a small number of requirements, forces and patterns exist in a catalogue, the result of these relationships is easy to understand: each individual relationship can be understood, and there are a small number of such relationships.

More typical in real-world applications are catalogues containing dozens or even hundreds of patterns, requirements and forces. In such cases, even though the individual relationships between requirements are easy to understand, the cumulative result of these relationships is obscured. Obtaining results manually from such an example would be time-consuming, error-prone and non-obvious. With the goal of comparing the results of the automated tool to some results that we know to be correct, this approach would be questionable.

If the evaluation algorithm described in chapter three produces obviously correct results in a small sample, will it also produce correct results with a larger, more complex sample? It turns out that this is indeed the case. The procedure described in [Chung 99] is well studied and scales nicely to large samples. The algorithm in chapter three describes how patterns may be used as input to Chung's evaluation procedure as a basis for initial node labelling. The small sample catalogue in this chapter demonstrates that when patterns are used as such input, the results are as expected. Therefore, any scalability issues related to the algorithm in chapter three would also have appeared when investigating the scalability issues related to Chung's evaluation procedure.

4.8 Observations

The number of requirements, patterns and forces in the sample catalogues were quite small. While this example does not necessarily correspond to real-world problems, it does serve to provide an easily solvable problem against which the results of the software tool may be verified in a believable way. Typically, the number of requirements,

patterns and forces would all be much larger, leading to a problem that is less easily solved within the constraints of this thesis.

It should be noted that the results produced by the software tool, both on the original example and on the modified example after a conflict was introduced, were the same as the expected results achieved manually. The expected results were arrived at by a process of deductive reasoning, and are thus independent of the evaluation procedure described in chapter three. Therefore, the fact that the results of the software tool match the expected results lends credibility both to the evaluation procedure and to its implementation in the form of the software tool.

Chapter three describes that a coarse-grained resolver could be used in place of a fine-grained resolver. This option was not exercised in the example; the small number of interactions between requirements made nearly all of the heuristics provided by the coarse resolver non-applicable. Such a resolver would be more useful in real-world problems where the number of interactions, requirements and patterns are much larger. It is mentioned simply to show that the design of the software tool allows for pluggable resolution strategies that scale better than simply asking the developer every time there is a conflict.

Finally, it has been demonstrated that deducing the impact of a pattern on a non-functional requirement that is not described or implied in the text of the pattern is impossible without considering the existence of a catalogue of requirements and forces independent of the pattern. By combining the pattern with such a catalogue, we can easily, in a completely or largely automated way, extract valuable information about the indirect impact of the pattern on a set of non-functional requirements.

5. Conclusions and Future Work

5.1 Summary

We note that much research has been done in the domains of patterns and of non-functional requirements. While at least one study has touched on the apparent linkages between these fields, no thorough research in this combined area has been done. The forces in a pattern seem to have much in common with non-functional requirements, so it seems likely that benefit could be derived from research into the combined area. In particular, we set out to discover whether information contained in pattern forces, when combined with non-functional requirement catalogues, could tell the developer more about the indirect effects of applying a pattern than could be deduced with the pattern alone.

To accomplish this, we first investigated the similarities between patterns and non-functional requirements and discovered that the forces in a pattern often describe the direct impact of the pattern on one or more non-functional requirements. Earlier research into non-functional requirements yielded a method of representing requirements and their relationships in a graph, as well as a procedure for evaluating that graph to discover the impact of some non-functional requirements on others.

Given that a catalogue of non-functional requirements and relationships in a particular problem domain could be quite large, and given the large number of patterns in existence that could be considered candidates for a particular set of problems, it was obvious that a strictly manual solution was untenable. With the goal of creating an automated or semi-automated solution that required little user interaction, we introduced

a formal specification for patterns, non-functional requirements, and forces. This specification allows us to represent catalogues of patterns, NFRs and forces in a reusable, machine-readable manner.

The procedure for evaluating graphs of NFRs was extended to allow for more accurate propagation of node labelling. With these extensions in hand, we formalized a complete transition function to collect and propagate labels in the graph. Using the forces in a pattern to provide the initial input to the evaluation procedure, we postulated that one could determine the indirect impact of a set of patterns on a set of requirements in an automated or semi-automated way.

To demonstrate that the user could achieve the stated results, we introduced a software tool that operated on the formal specifications mentioned previously, and implemented the evaluation procedure with the described extensions. We then created a sample catalogue of patterns, NFRs and forces. After manually analyzing the sample catalogue to understand what the desired results were, we ran the software tool on this input. The results from the software tool matched exactly the manually achieved results.

We have thus demonstrated not only that the indirect impact of patterns on non-functional requirements can be deduced, but also that this procedure may be largely if not completely automated. This allows developers to more accurately understand the impact of applying a pattern. It also allows them to more easily choose between candidate patterns based on how well those patterns correspond with their non-functional goals.

The following goals were achieved:

- A relationship between patterns and non-functional requirements was established.

- A method of formally specifying patterns, non-functional requirements and forces was introduced.
- An existing evaluation procedure was augmented to provide an accurate evaluation of a pattern's impact on non-functional requirements.
- A software tool was implemented to demonstrate this principle.
- Results gathered from this tool matched the expected results in all cases.

5.2 Comparison to Previous Work

A great deal of research in the field of non-functional requirements in software engineering provided the groundwork for the augmented evaluation procedure [Chung 99]. They identify numerous common non-functional requirements and describe how they may be arranged in a graph that expresses the relationships between them in a particular domain. They introduce an evaluation procedure that propagates the values of certain requirements to those that they directly and indirectly affect. They describe informally certain extensions to this evaluation procedure which might provide more accurate propagation of labels, but do not formalize them. This work did not investigate the relationship between non-functional requirements and patterns.

In [Gross 01], Daniel Gross and Eric Yu begin to identify the interesting relationship between design patterns and non-functional requirements. Although they do not formally recognize the encoding of NFRs in pattern forces, they do identify forces as being an important source of information regarding the impact of a pattern. They suggest that future work might be focused on determining the applicability of patterns during

design, and that retrieval of patterns may be improved by combining pattern knowledge with the cataloging of non-functional requirements.

Numerous forays into the formal specification of patterns are investigated [Eden 98a, Eden 98b, Deugo 01]. Most involve the specification of the pattern solutions in such a way that the template solution may be automatically or semi-automatically generated into code. Although at least one of these methods involves a complete representation of all aspects of a pattern in XML [Deugo 01], none concentrate on the formalization of pattern forces in a sufficient way as to render them accessible to an automated tool. The method described in [Deugo 01] provides good groundwork for extension to this application.

5.3 Limitations

In [Chung 99], the authors describe that non-functional requirements in the context of an application domain always have a subject. This subject is the part of the system to which the requirement applies. This characteristic is called argumentation. In this study, argumentation is limited to a single, implied argument. That is, all non-functional requirements in a context apply to the same argument. This is obviously not a realistic expectation of a real-world development scenario.

The decision to limit this thesis to discussion of a single subject was made for simplicity. Furthermore, in no way does this decision impact the validity of the results. With more research and development time, the evaluation procedure described in chapter three, and the sample implementation described in chapters three and four, could be extended to accommodate any number of subjects. Such subjects serve as a further

identifier of a requirement instance, and as such two requirements describing the same quality (e.g. Usability) on different subjects would appear as two nodes in the graph. The evaluation procedure remains unaffected. [Chung 99] describes the construction of such graphs.

5.4 Extensions and Future Research

As described in the previous section, the formal specification descriptions should be extended to accommodate requirements having subjects. This extension would be straightforward and would allow for a richer catalogue to be evaluated.

The sample implementation presented, while used solely to validate the evaluation procedure described in chapter three, could be extended into more useful tool. A more graphical presentation of the results would be helpful to the user. In addition, it does not allow for the user to make individual subjective modifications to the graph and re-run the procedure with those changes in effect. Such an extension would be useful given the subjective nature of requirement impacts in a particular domain. The tool could also be extended to provide an editing facility by which the catalogues of patterns, requirements and forces could be entered and modified in a user-friendly way.

The tool would also benefit from the ability to allow the user to define heuristic rules. Currently, it contains a number of heuristics, but these are hard-coded into the software. This additional flexibility would allow the user to have more fine-grained control over the behaviour of the software, and tailor it to a specific scenario.

The current formal specification for patterns, NFRs and forces assumes that all three catalogues are local. That is, the entire contents of the three catalogues are

implicitly accessible to the software tool. An interesting area of future research would be to investigate the implications of distributing these catalogues over a network. The DTDs would be extended to implement a resource location mechanism by which other patterns, NFRs and forces could be located in a network environment. Finally, recent work has focussed on the use of schemas in place of DTDs in applications similar to this; it would be useful to investigate whether the formal specification here could benefit from taking this direction.

The tool is currently geared toward a scenario where the user selects one or more patterns, and one or more NFRs. It provides output describing how each selected pattern affects each selected NFR. However, it is clear that the tool computes much more information that could be useful to the user. For example, a pattern that the user did not select could result in strongly positive contributions to all selected NFRs, but the user would not know this. It would be useful to investigate what information the tool has but does not present, or what useful scenarios the tool prevents the user from performing simply because of its user interface.

Alexander describes a group of patterns, which he calls a pattern language. These patterns are related in some larger way, either because of the common task they collectively assist in, or because of the relationship between the problems they solve. It is possible that by performing graph evaluation, we can gain useful information about the relationship between two or more patterns. In particular, it may be possible to partially derive pattern languages from a group of patterns by thorough analysis of the results. This interesting proposition deserves future research.

With this software tool in hand, the majority of the work in determining the indirect impact of patterns on NFRs lies in specifying these patterns and NFRs in XML. For patterns, the forces must be first identified and then extracted and represented in XML in such a way that they identify requirements in another catalogue. This work, however, needs to be performed only once. A shareable, publicly accessible catalogue of patterns and their forces would be a valuable resource. Also, NFRs and forces for a particular domain must also be identified and represented in XML. It has been noted that many NFRs and their interactions are domain-independent. As such, with the formal specification scheme defined here, it would be a worthy endeavour to catalogue and provide definitions for many common NFRs, and to document their domain-independent interactions. Such a catalogue could be used as the basis of a domain-dependent catalogue, or used as input to the software tool directly.

Chung et al. describe their sample implementation [Chung 99]. It would be interesting to perform side-by-side comparisons of their implementation and the software tool described here. In addition, it would be beneficial to see how they have addressed certain usability issues, such as the resolution of conflicts and the presentation of the interdependency graphs.

5.5 Conclusions

Non-functional requirements are a critical component of software engineering. Many of them are highly desirable goals, yet it is difficult to achieve them in a systematic way. Often, they are achieved as after-the-fact improvements. Building the satisfaction of

these requirements into the design process helps to create software correctly in the first place, saving time and money in the process and resulting in a system that has the desired characteristics.

Design patterns have become an integral part of software engineering in recent years. Serving as templates for proven solutions to recurring problems, patterns also encode useful information about the impact of a pattern on certain attributes of the system. This information is often neglected in the search for a pattern that simply solves the problem.

The information encoded in pattern forces tells us about the direct impact of a pattern on the system. Due to a complex, often-overlooked series of interactions between non-functional requirements in a system, there is more to the story than simple direct impact: a pattern can have far-reaching effects that are not readily apparent. Combining the information encoded in a pattern with a catalogue of non-functional requirements and their interactions can gain the developer valuable insight into these effects.

We have clearly identified the relationship between patterns and non-functional requirements, specified a formal method for representing patterns, non-functional requirements and their relationships, and introduced an evaluation method. The implementation of our software tool provides concrete evidence that this approach is tenable and correct.

The contribution of this thesis is both relevant and concrete. Specifying a set of non-functional goals at the outset of a software project is a common practice. Using design patterns where they are appropriate is likewise commonplace. Determining the

applicability of patterns is recognized as a difficult problem. This thesis provides helpful support to the software developer in such a situation.

References

- [Alexander 77] Christopher Alexander, *A Pattern Language*, Oxford University Press, 1977.
- [Alexander 79] Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.
- [Apache] The Apache Software Foundation, *Xerces Java Parser*, <<http://xml.apache.org/xerces-j>>
- [Appleton 00] Brad Appleton, *Patterns and Software: Essential Concepts and Terminology*, <<http://www.enteract.com/~bradapp/docs/patterns-intro.html>>
- [Boehm 76] B. W. Boehm, J. R. Brown, M. Lipow, "Quantitative Evaluation of Software Quality", in *Proceedings, 2nd International Conference on Software Engineering*, IEEE Computer Society, 1976, pp. 592-605.
- [ChiliPLoP 02] ChiliPLoP, <<http://www.agcs.com/supportv2/techpapers/patterns/chiliplop>>
- [Chung 99] Lawrence Chung, Brian A. Nixon, Eric Yu, John Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, 1999.
- [Coplien 96] James O. Coplien, *Software Patterns*, SIGS Books & Multimedia, 1996.
- [Cunningham] Ward Cunningham, *Portland Pattern Repository*, <<http://c2.com/ppr>>
- [Deugo 01] D. Deugo, D. Ferguson, "XML Specification for Design Patterns" in *Proceedings of the Second International Conference on Internet Computing*.
- [Eclipse] The Eclipse Project, <<http://www.eclipse.org>>

- [ECOOP 02] European Conference for Object-Oriented Programming, <<http://www.ecoop.org>>
- [Eden 98a] A. H. Eden, Y. Hirschfeld, A. Yehudai, *Precise Notation for Design Patterns*, Department of Computer Science, Tel Aviv University, 1998. Available as technical report 327/98.
- [Eden 98b] A. H. Eden, "Giving 'The Quality' a Name", in *Journal of Object Oriented Programming*, Vol. 11, No. 3, June 1998, pp. 5-11.
- [EuroPLoP 99] European Conference on Pattern Languages of Programming and Computing, <<http://www.argo.be/europlop>>
- [Gamma 94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: elements of reusable object-oriented software*, Addison-Wesley, 1994.
- [Gosling 96] James Gosling, Bill Joy, Guy Steele, *The Java Language Specification*, Addison-Wesley, 1996.
- [Gross 01] Daniel Gross, Eric Yu, *From Non-Functional Requirements to Design through Patterns*, Faculty of Information Studies, University of Toronto, 2001.
- [IEEE] IEEE Computer Society: Publications, software engineering standards, conferences, Washington.
- [Keller 90] Steven E. Keller, Laurence G. Kahn, Roger B. Panara, "Specifying Software Quality Requirements with Metrics", in *Tutorial: System and Software Requirements Engineering*, IEEE Computer Society Press, 1990, pp. 145-163.
- [Lea 00] Doug Lea, *Patterns-Discussion FAQ*, <<http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html>>
- [Merriam-Webster] Merriam-Webster Collegiate Dictionary, <<http://www.m-w.com>>
- [OOPSLA 02] ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, <<http://oopsla.acm.org>>

- [PLoP 00] The Pattern Languages of Programs Conference, <<http://st-www.cs.uiuc.edu/~plop>>
- [Sommerville 92] I. Sommerville, *Software Engineering*, Addison-Wesley, 1992.
- [W3Ca] World Wide Web Consortium, *Extensible Markup Language*, <<http://www.w3.org/xml>>
- [W3Cb] World Wide Web Consortium, *XML in 10 points*, <<http://www.w3.org/xml/1999/XML-in-10-points>>