Policies
As Design and Implementation Artifacts For Non Functional Requirements

by

Feng Chen

A thesis submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements
for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

September 8, 2002

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

Policies
As Design And Implementation Artifacts
For Non Functional Requirements.

Submitted by Feng Chen

in partial fulfillment of the requirements

for the degree of Masters of Engineering

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University

September 8, 2002

ABSTRACT

The implementation of Non Functional requirements (NFRs) often results in scattered code in the whole system, because there are no modular design and implementation artifacts for NFRs. This thesis proposes to use policies as the design and implementation artifacts for NFRs. Relevant policy mechanisms are surveyed and characterized through a list of attributes. Two policy mechanisms PEOCL and Aspect are proposed to be used for designing and implementing NFRs. PEOCL is extended from Object Constraint Language and is used to represent design-level policies for NFRs. PEOCL policies are further mapped to aspects in AspectJ at the code level. An abstract aspect library is also developed to support this methodology. This methodology is validated and illustrated through a case study. This approach realizes modular design and implementation for NFRs and the decoupling of the design and implementation for NFRs and those for functional features, thus achieves readability, tracability, non-intrusive adaptation, evolvability, and reusability.

Keywords:

Non Functional Requirements (NFR), Quality Attributes, Policy, Rule-based system, Constraint, Advanced separation of Concerns, Aspect-Oriented Programming (AOP), Reflection, Program Transformation/meta-programming, Software Development Methodology

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Professor Babak Esfandiari, for his constant guidance, patience, inspiration, advice and encouragement throughout the research work and the preparation of this thesis.

I would also like to thank my family for their support, understanding and love during my study and research.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

ABBRIEVIATIONS AND ACRONYMS

AOP	Aspect Oriented Programming.
FR	Functional Requirement.
NFR	Non Functional Requirement.
OCL	Object Constraint Language.
PEOCL	Policy Extension to OCL.
UML	Unified Modeling Language.

CHAPTER 1 INTRODUCTION

1.1 Motivation

Much of systems quality is expressed as Non-Functional Requirements [Chung00a, Chung00b, Gross00, Chung94], also called Quality Attributes [Babacci95, Kazman99, Kazman00]. Examples of Non Functional Requirements (NFRs) include performance, usability, reliability, security, maintainability, etc. Non Functional Requirements are crucial for system success, but they are hard to deal with since they

- Impact the design and implementation in many different modules in a scattered fashion, and
- Often come or change at a later stage in the software life cycle

The result of the above two factors is a costly evolution path toward a highly coupled complex system.

In order to address this problem, we need to identify the NFRs as early and clearly as possible and we need to understand fully how NFRs affect the traditional object-oriented designs. Work in this area includes Rational Unified Process [Rup00], NFR Framework [Chung00a, Chung00b, Chung94] and Architecture Tradeoff Analysis Method [Kazman00]).

A functional requirement (FR) can be expressed in a simple formula:

Output = F (Input);

While NFRs can not be easily expressed in that type of formula. A common characteristic of all NFRs is that NFRs are about how *well* the Function F works, not about *what* F does.

The traditional approaches are mainly aiming at architecting F's design with all NFRs considered. The implicit assumption is that we could identify NFRs before designing F. The shortcoming of the traditional approach stems from the fact that NFRs, like other requirements, often come or change at a later stage in the software life cycle.

1.2 Problem Statement

This research work addresses this issue from a different angle. Assuming the design and implementation for FRs are done without worrying too much about NFRs, we want to seek a way to design and implement for NFRs in totally separate modules. The design and implementation artifacts for NFRs are expected to reference the design and implementation artifacts for FRs, because NFRs are about how well the functional features are running.

Essentially the objective is to address the issue through propagating the separation of concerns at the requirement level (i.e., NFRs and FRs are considered separately) down to the design and implementation levels (i.e., separate the design and implementation of NFRs from the design and implementation of FRs). So we reduce the problem to this question: How to design and implement NFRs in a clean and modular way just like what

we are doing with Functional Requirements? A more specific question is: what kind of design and code artifacts can implement NFRs in a modular way?

1.3 Proposed Solution

A major characteristic of a NFR is "*crosscutting semantically but centralized syntactically"*, i.e., a NFR is typically described in one place but crosscuts many parts of the system semantically. For example, a security NFR states that all transmitted messages must be encrypted. It is a simple statement at the requirement level, but all the subsystems that transmit messages must implement such a requirement. The design and implementation for such a requirement will be scattered throughout the entire system. A NFR could involve many different modules at the design level and the code level. So the desirable characteristic of a NFR's design and implementation artifacts is that they should be able to reference and control multiple modules in the design and implementation of FR, without actually modifying those modules.

We use the term *policy* for any mechanisms that are "*crosscutting semantically but centralized syntactically*" (For details see section "3.1 Definitions of Policies"). The *policy* mechanism in general provides the ability to express constraints and rules with respect to an existing system. There are many different forms of policy mechanisms (e.g., OCL, PIB, COO, R++, Exception, ILOG JRules, AspectJ, etc., see Chapter 2 and Chapter 3). A common feature of those policy mechanisms is that they are all crosscutting semantically but modularized syntactically. This is exactly what the NFR's ideal design

and implementation should be.

We propose to use *policies* as the design and implementation artifacts for NFRs. Specifically, after studying various forms of policy mechanisms, we extended OCL [OCL97] to represent design level artifacts for NFRs, and then use aspects [AOP01] as implementation level artifacts for NFRs. Policy Extension to OCL (PEOCL) includes OCL plus the NFR ontology (see "**Error! Reference source not found.**") and UML Metamodel [UMLMeta97]. "Figure 1" illustrates our approach graphically.

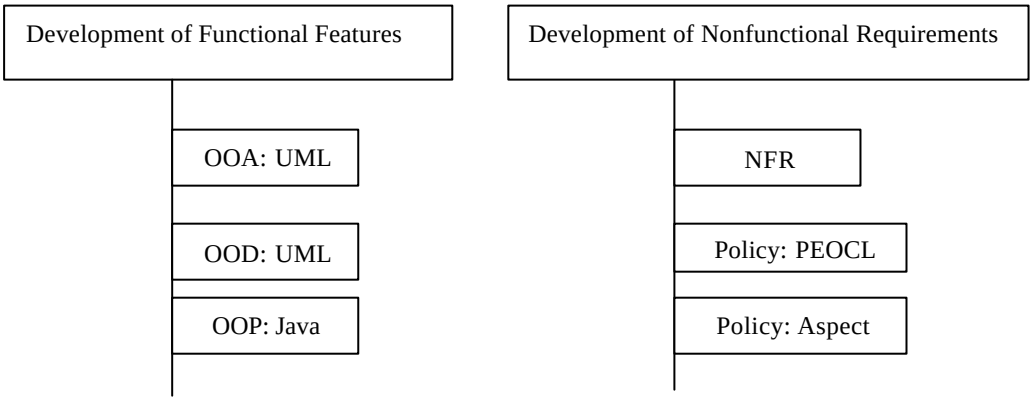

Figure 1 Separate design and implementation for NFRs from those for FRs

We propose to use Policy-Extension to OCL (PEOCL) to capture the NFR's design level policies. Object Constraint Language [OCL97] is extended to include the ontology of NFRs [Chung00a, Chung00b, Chung94, Babacci95] and to include UML Metamodel [UMLMeta97]. The extension of ontology helps to enrich the predicates of OCL to express NFR concerns easily. UML Metamodel enables us to reference collections of UML model elements when expressing NFR policies. The ability to reference collections

of UML model elements is essential due the crosscutting nature of NFRs.

PEOCL policies can be implemented in either design patterns [Gamma97] or aspects. The focus of this research work is to implement policies in aspects. More specifically, we use AspectJ [AspectJ02]. AspectJ extends the popular object-oriented language Java and has many language supports to address crosscutting concerns at the code level. We also developed a generic abstract aspect library for common NFRs by using AspectJ.

As a case study of this methodology, we developed an online chat room client-server system to illustrate and validate this approach.

1.4 Thesis Contribution

The contribution of this thesis includes:

- Identified a problem of NFR's *scattered* impact to design and code, and raised the question of how to design and implement NFRs in a modular way. Specifically, use the term "policy" to capture all design and implementation mechanisms with the characteristics of "crosscutting semantically and centralized syntactically"

- Surveyed *policy mechanisms* at the design level and the implementation level, characterized policy mechanisms through a list of attributes. This not only helped ourselves choosing the best mechanisms (i.e., PEOCL and AspectJ) for design and

implementation level artifacts for NFRs, but also will be useful for future research on improvements to existing policy mechanisms to better suit the need of designing and implementing NFRs

- Proposed a software development methodology to design and implement NFRs. Specifically proposed to use PEOCL to capture design level policies for NFRs and to use aspects to implement PEOCL policies. This methodology realizes the benefits of the S*eparation of Concerns* principle

- Designed and implemented a generic abstract aspect library for common NFRs

- Conducted a case study through implementing a distributed chat room system by using the proposed methodology

1.5 Organization Of The Thesis

The rest of the thesis is organized as follows:

Chapter 2 reviews the state of the art in the areas of requirement analysis and definition, design, and implementation for NFRs. Reasons are given informally on why OCL and AspectJ are good candidates for representing the design and implementation artifacts for NFRs. Each related work is presented one by one individually to provide some background information for the readers who are not familiar with that particular work.

Chapter 3 analyzes policy mechanisms more generally through defining a list of attributes of policy mechanisms. Various forms of concrete policy mechanisms are positioned by using the list of attributes from this formal analysis.

Chapter 4 uses the result from chapter 3 to explain why AspectJ is ideal for implementing NFRs, and why OCL is not sufficient for representing design level artifacts for NFRs, and then introduces Policy Extension to OCL (PEOCL). Then our proposed methodology is explained through examples.

Chapter 5 presents a case study of the development of chat room system. The typical artifacts by using the traditional object-oriented methodology are presented first, then new NFRs are introduced, they are mapped to policies in PEOCL, and then PEOCL policies are further mapped to AspectJ code. We can achieve one to one modularized mapping for most common NFRs.

Chapter 6 summarizes the overall work and points out the future work directions.

The appendix describes the NFR ontology, and an example of using design patterns to implement policies.

CHAPTER 2 THE STATE OF THE ART IN ANALYSIS, DESIGN, AND IMPLEMENTATION OF NFRS

This chapter reviews the background information on existing NFR-related work at the requirement level, design level, and implementation level.

Readers who are familiar with those related works can skip the corresponding sections.

2.1 Requirement Definition And Analysis For NFRs

This section summarises these related works: non-functional requirement (NFR) framework, quality attribute and architecture trade-off method. Each of the related works is discussed in one section. The key features, weaknesses and relevance to our work are also discussed in each section.

2.1.1 Non-Functional Requirement Framework

The NFR Framework [Chung94, Chung00a, Chung00b] treats non functional requirements as goals to be addressed during the development process. NFRs, major design decisions, and their relations (e.g., refine, support, object to, etc.) are captured in a Goal Graph. The nodes in the goal graph are either goals (i.e., NFRs) or design decisions. Goals can be refined into detailed concrete goals. Design decisions can impact goals positively or negatively.

A tool "NFR Assistant" is also provided by this research work, it supports:

- Refining initial high-level goals to detailed concrete goals
- Identifying the decision points (need for tradeoffs)
- Evaluating and choosing among alternatives
- Recording arguments for or against particular development decisions and tradeoffs
- Detecting and correcting omissions, ambiguities, conflicts and redundancies

NFR Framework provides a body of NFR-related vocabulary, allowing us to succinctly capture a large number of NFR-specific concepts in an organized manner. It also makes the relationships between NFRs and intended decisions exp licit, this helps us to understand fully the impact of every design decision, typically one design decision may impact multiple NFRs.

Figure 2 shows an example of a NFR Goal Graph [Gross00]. The example captures the analysis and design on how to provide a compact representation of the state of the system, i.e., only the deviation from the normal state is stored, instead of all the states for all the objects.

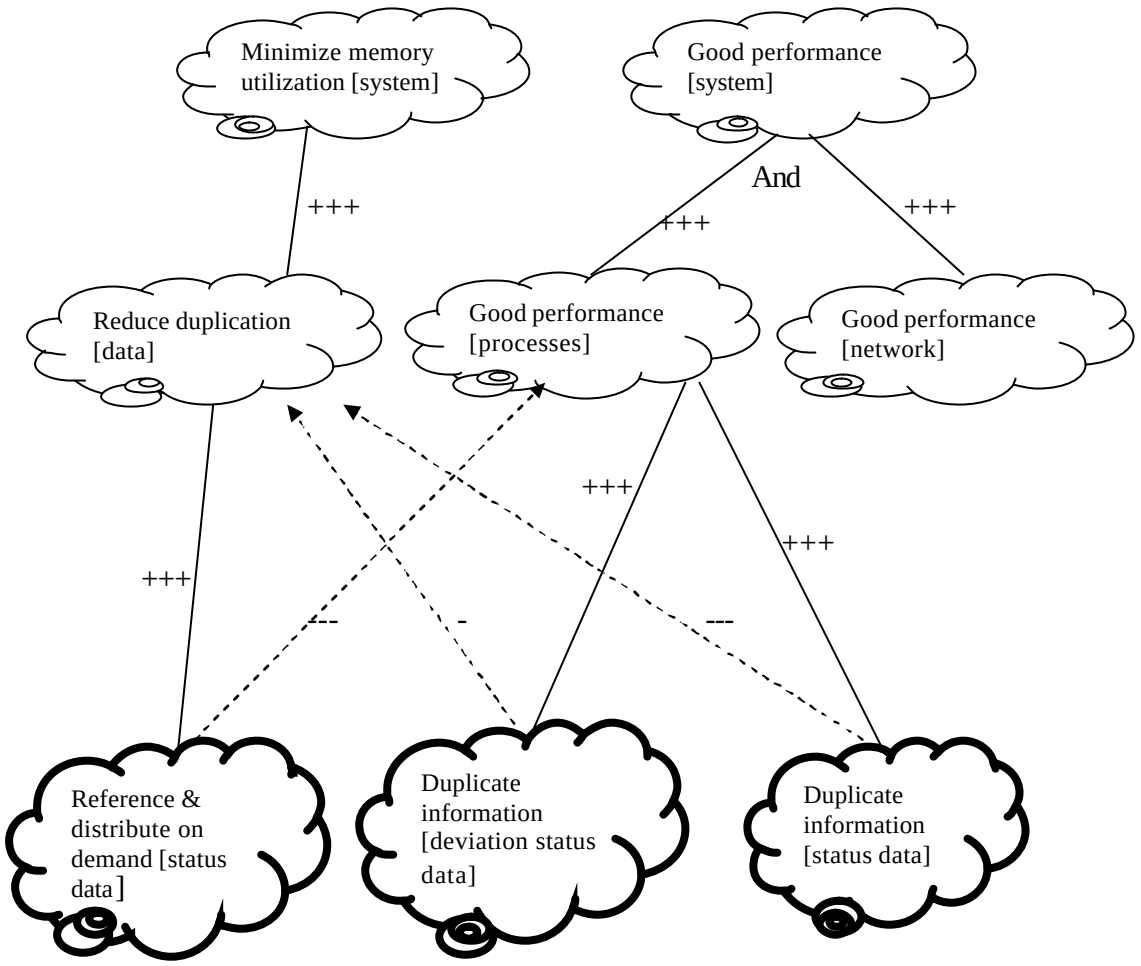

Figure 2 Use "NFR Goal Graph" to represent "Deviation Design Pattern"

This work provides a solid framework to formally analyze and define non functional requirements and associate the non functional requirements with major design decisions. It addresses mostly architectural issues at the requirement analysis and architectural-design level. It does not address any issues at the coding phase.

2.1.2 Quality Attributes Taxonomy And Architecture Tradeoff Analysis Method

The quality attributes Taxonomy is the result of CMU SEI's research work on how quality attributes impact the software architecture [Babacci95, Kazman97, Kazman00]. The taxonomy is divided into these areas: performance, dependability, security, and safety. As illustration, the security quality attribute taxonomy is presented in "Figure 3 Security Taxonomy" [Babacci95].

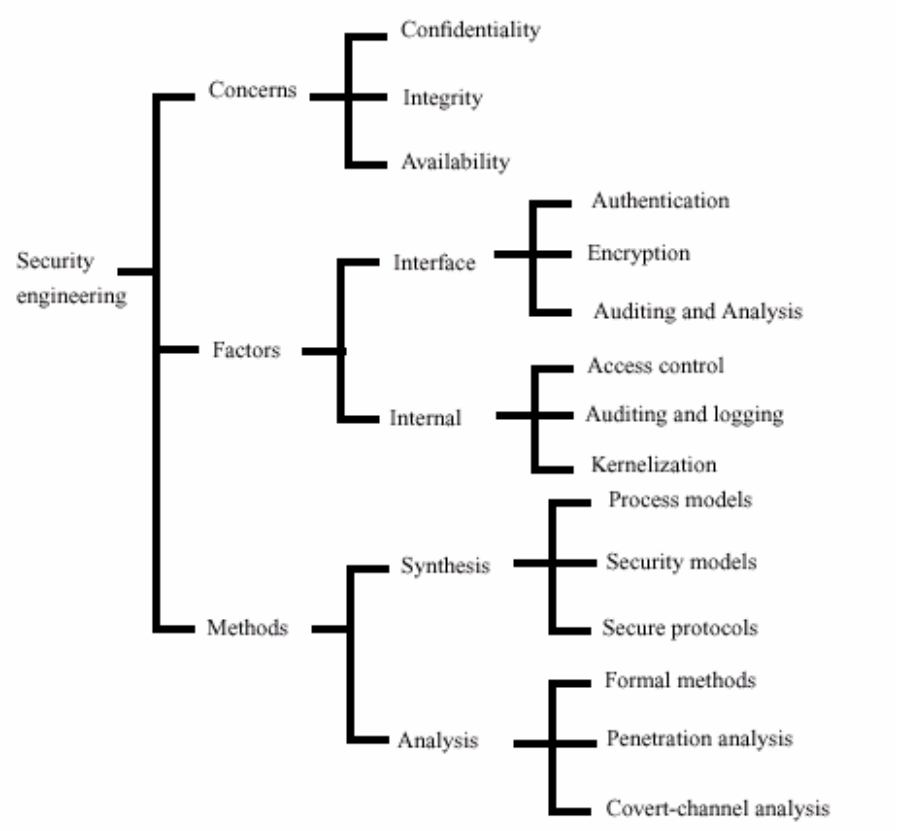

Figure 3 Security Taxonomy

All quality attributes are analyzed through three dimensions: concerns, factors, and

methods.

Concerns are the parameters by which the attributes of a system are judged, specified and measured. Requirements are expressed in terms of concerns.

Factors are the properties of the system and its environment that have an impact on the concerns. Depending on the attribute, the attribute-specific factors are internal or external properties affecting the concerns. Factors might not be independent and might have cause/effect relationships. Factors and their relationships should be included in the system's architecture. Security factors are the aspects of the system that contribute to security. These include system/environment interface features and internal features such as auditing.

Methods specify how we address the concerns: analysis and synthesis processes during the development of the system, and procedures and training for users and operators. Methods can be for analysis and/or synthesis, procedures and/or training, or procedures used at development or execution time.

The terminology used in these taxonomies can serve as a vocabulary to specify a NFR, and then drive the design of the architecture.

The Architecture Tradeoff Analysis Method (ATAM) proposes to identify *sensitive points* and *tradeoff points* when designing the architecture of a system. *Sensitive points*

are the alternatives for which a slight change makes a significant difference in some quality attributes. *Tradeoffs* are decisions affecting more than one quality attribute. The identification, analysis, and documentation of sensitive point and tradeoffs improve the chance of the overall architecture meets the required quality attributes. The direction of this work (and research on NFR Framework) can be best illustrated through "Figure 4 ".

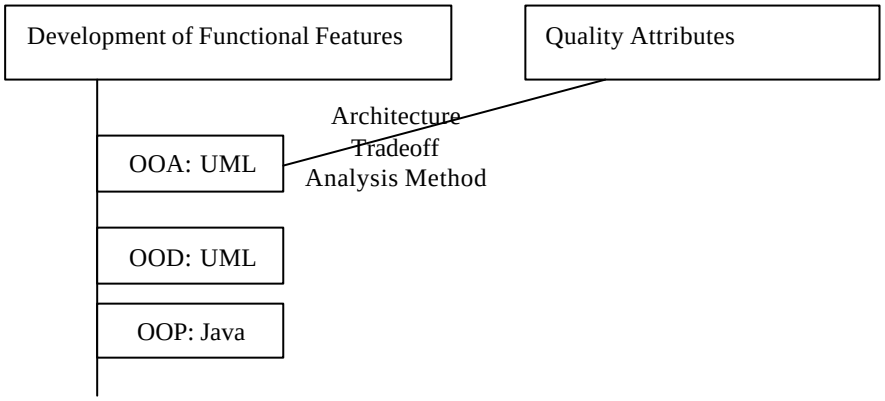

Figure 4 Quality Attributes and Architecture Tradeoff Analysis Method

2.1.3 Conclusion

ATAM and NFR Framework are still relying on the traditional ways of designing and implementing software, they try to uncover and fully understand more NFRs up-front and design a software architecture that satisfies all the NFRs. They do not address either the issue of NFR's scattered impact to design and code, nor the issue of evolution (e.g., to minimize changes when NFRs change or new NFRs come).

Overall, the NFR Framework and Quality Attributes Taxonomy/ATAM work provides a solid foundation for the analysis and definition of NFRs at the requirement level. Our work will not further address issues already addressed by those works. Our work will reuse the ontology used by NFR Framework and Quality Attribute Taxonomy.

2.2 Design For NFRs

This section provides background information on some existing mechanisms that can represent design artifacts for NFRs.

2.2.1 Desirable Characteristics of A Design For NFRs

This is the list of characteristics that we think a good design for NFRs should have, the rationale for them are further described below.

- The design artifacts for NFRs shall be separated from the design artifacts for functional feature
- The design artifacts for NFRs shall reference design artifacts for functional features, ideally the design artifacts for functional feature shall be from object-oriented method
- The design artifact shall be formal

- The notation shall be easy to use

The design artifacts for NFRs shall be separated from the design artifacts for functional feature. *Separation of Concerns* [Dijkstra76] is one of the most important software engineering principles that helps to manage the complexity of a software system. Many benefits can be derived from it: readability, tracability, non-intrusive adaptation, evolvability, and reusability. NFRs and FRs are typically stated and considered separately at the requirement level. It is very natural to map them separately into separate design and code modules.

The design artifacts for NFRs shall reference design artifacts for functional features, because NFRs describe how well those functional features should behave. Ideally we think the design artifacts for functional feature shall be from object-oriented method, because object-oriented method is the most widely adopted software development method today.

A formal notation gives us rigid designs. But notations that require high degree of mathematical background typically will not get wide adoption. Thus we emphasis on usability of the notation.

Base on those criteria, we will discuss OCL (Object Constraint Language) and PIB (Policy Information Base) in the next two sections.

2.2.2 Object Constraint Language

OCL [OCL97] is a formal language to express side effect-free constraints. It can be associated with UML [UML00]. OCL overcomes the disadvantage of traditional formal languages, it does not require the user to have a strong mathematical background.

OCL is *typed*, each OCL expression has a type. Each OCL expression is conceptually *atomic* (i.e., the state of the objects in the system cannot change during evaluation of the expression). OCL *does not have a flow control mechanism*, it is not intended to be a programming language. As a modeling language, all implementation issues are out of scope and cannot be expressed in OCL.

OCL can be used to specify invariant on classes and types in the class model, specify type invariant for Stereotypes, describe pre and post conditions on operations and methods, describe guards, as a navigation language (navigating to attributes, operations, association ends, associations), specify constraints on operations, etc.

The language constructs of OCL are listed below to give the reader a detailed view of the language:

- The basic . and -> notation for getting the property (including attributes, operations, associations, and association ends) of an object
- Conditional expression

- Relational expression (relational operations include =, >, <, >=, >=, and <>)
- Logical expression (logical operators include ‘and‘, ‘or‘, ‘xor‘, ‘not’, and ‘implies‘)
- Arithmetical expression (operators include +, -, *, /)

  Examples:

  Wife’s sex is female:

  self.wife->notEmpty implies self.wife.sex=female

  A person can not both have a wife and a husband:

  not ((self.wife->size=1) and (self.husband->size=1))

- Types: basic types (integer, real, string, boolean), enum, all class specifiers in the associated UML model, collections (set, bag, sequence), and “OclAny” (super-type of all types in OCL).
- Notation for previous values in Post-Conditions (time expression), e.g.,

  Person:: birthdayHappens()

  post: age = age@pre + 1

  age@pre represents the values of ‘age’ in precondition.

- Operations on collections include “forAll“, “exists“, “select“, and “reject“. e.g.,

  employee->forAll(age>18) -- true if everyone is over 18 (a boolean value)

  employee->exists(age>58) -- true if at least one is over 58 (a boolean value)

  employee->select(age>50) -- all employees who are under 50 (a collection)

  employee->reject(isMarried) -- all employees who are not married (a collection)

The work by OMG on OCL 2.0 is being done right now, many proposals are being reviewed and not finalized. We will only use OCL1.1 in this thesis.

2.2.3 Policy Based Management MIB

Policy Based Management MIB is a domain-specific example of how NFRs can be mapped to policies. [Waldbusser00] is a draft for the MIB definition of Policy-Based Network Management. Some of the relevant concepts are presented as follows.

Policy-based network management is the practice of applying management operations globally on all managed objects that share certain attributes. Policies always express a notion of:

if (an object has certain characteristics) then (apply operation to that object)

PIB (Policy Information Base) restricts Policies to take the following normal form:

if (*policyFilter*) then (*policyAction*)

A *policyFilter* is program code which results in a boolean to determine whether or not an object is a member of a set of objects upon which an action is to be performed.

A *policyAction* is an operation performed on an object or a set of objects.

The *execution model* for policies on a managed device is:

foreach element for which policyFilter returns true

execute policyAction on that element

Policy examples:

If (interface is fast ethernet) then (apply full-duplex mode)

If (interface is access) then (apply security filters)

If (gold service paid for on circuit) then (apply special queueing)

Policy filters and policy actions are expressed with the *policy language*. The policy language is intended to be familiar to programmers in a variety of languages, including Perl and C. *This language is formally defined as a subset of ISO C.* Some examples of the features that have been removed from the C language are: function definitions, pointer variables, structures, enums, typedefs, floating point and pre-processor functions.

The possible *attributes that could be filtered on are defined (by using ASN.1 notation) as nodes of a MIB tree*. Also a set of convenience C functions are predefined in the draft.

The PIB for differentiated service QoS (see [PIB00]) describes a structure for specifying policy information that can then be transmitted to a network device for the purpose of configuring policies at that device. The model underlying this structure is one of well-defined policy rule classes and instances of these classes residing in a virtual information store called the Policy Information Base (PIB).

The PIB consists of classes that represent functional elements in the data path (e.g. classifiers, meters, and actions), and classes that specify parameters that apply to a certain type of functional element (e.g. a Token Bucket meter or a Mark action). Parameters are typically specified separately to enable the use of parameter classes by multiple policies. Overall, this approach summarizes the frequently used rules in the "Differentiated Services" problem domain, and then encodes all those rules into metadata represented in a Policy Information Base.

2.2.4 Design Patterns

Some design patterns [Gamma97, Weiss01] can be used to express policies as well. The Adapter pattern can be used for adding actions before and after functional calls. The Visitor pattern can be used for adding new crosscutting features on a complex data structure. The Subject and Observer pattern can be used to implement automatically-triggered rules that monitor the state of the system. The Pipe and Filter pattern and Chain of Responsibility pattern can be used to enable the addition of new responsibilities without modifying the original code.

It is a common practice by the industry to use design patterns (e.g., adapter, visitor, observer, chain of responsibilities, etc.) to facilitate the non-intrusive addition of design and code. The main drawback of this approach is its anticipatory nature. It assumes that at the time of the initial design, the future expansions in every feature have been anticipated. So the hooks are built in the very beginning. That is not necessarily always

true. First, the future extensions may not be anticipated. Second, the anticipated extensions may never happen, and the unnecessary complex design and implementation becomes the 'fat' of the system that incurs unnecessary cost in both the initial implementation and future maintenance.

Also the above mentioned design patterns tend to be more suitable for functional features than for NFRs. There are functional features that are crosscutting, e.g., synchronization. But the scope and pervasiveness of NFRs' crosscutting nature tend to demand more flexibility than crosscutting functional features. For example, adapter pattern allows us to add extra behaviors before and after a method invocation, so it is possible to add logging messages through adapters to log the entrance and exit of a method. But what if the requirement is to log all method invocations? Then we will have difficulties to log the invocation of the methods in the adapter itself. Policy mechanisms like OCL or AspectJ do not have this type of difficulty.

2.2.5 Conclusion

OCL is associated with UML class diagrams and is formal but not too formal, both are very desirable features that we want (see section 2.2.1). But it lacks the vocabulary for NFRs and it can not specify constraints on a collection of UML model elements. These drawbacks will be addressed in section 4.1.

Design patterns are typically summarised from widely-used practices, they are proven and can be adopted without extra programming language or notation. But for the problem we are trying to address here, i.e., to create separate design and implementation artifacts for NFRs, the two characteristics of design patterns -- anticipatory nature and non-pervasiveness -- makes it less attractive than the other policy mechanisms.

Policy Information Base is a good design approach to address NFR concerns. It separates the design decision for NFRs from those for FRs. This is the ideal approach to the design and implementation of NFRs. We will continue to follow and generalize the idea of using policies to address NFR concerns in the later sections: section 2.3 will further review many different forms of policies at the implementation level, and section Chapter 3 will analyze various policy mechanisms formally through defining a list of characteristics.

2.3 Implementation for NFRs

This section reviews various policy mechanisms that can be used to implement NFRs.

2.3.1 Desirable Characteristics of An Implementation For NFRs

Quite similar to the criteria of a desirable design for NFRs, the implementation for NFRs should have these characteristics:

- The Implementation for NFRs shall be separated from the implementation for

functional features

- The implementation for NFRs references implementation artifacts (i.e., code) of functional features from object-oriented programming methodology

Those two criteria will be described as "syntactically modularized/centralized, while semantically crosscutting" in section 3.1.6.

Specifically we will review these related works: Constraint object-oriented (COO) programming style, ILOG JRULES, R++, Exceptions, and AspectJ.

2.3.2 Constraint Object-Oriented Programming Style

[Bolognesi00] introduced a new programming style – Constraint-Oriented Style --into the existing object-oriented language Java. The new method is called "Constraint and Object Oriented" (COO) programming style. The main concepts of COO are explained below.

Constraint-oriented decomposition models abstract aspects of behaviour, or functionality, that ignores physical boundaries. Constraint-oriented decomposition is a form of functional decomposition, it could be regarded as orthogonal to object-oriented decomposition (where system is divided into self-contained objects that has both data and associated functionality).

A *constraint* is modeled as an object. There are two types of constraints: D-constraints,

which are instances of D-classes, and CO-constraints, which are instances of CO-classes. D-class and CO-class are defined below.

An *observable method* of a class is a public method, whose return value type is always Boolean, whose parameters are read-only (i.e., no out parameters).

A D-Class (Data-encapsulating class) is a class which contains at least one data field and at least one observable method, and contains two public methods *Store*() and *Restore*(), and possibly some private store variables (to implement a recovery mechanism). A D-class may include other generic methods (this falls into the traditional object-oriented programming paradigm).

The syntactic structure of a D-class is as follows:

```
class D implements Recoverable {
        //---------------------Data Fields ---------------------------
        ...
        //---------------------Observable Methods ------------------
        public Boolean M (Type1 param1, ......,  TypeN paramN) { ..... }
        ...
        //---------------- Store Variables and Methods ------------------
        ...
        public void Store() { ......}
        public void Restore() { ..... }
        //----- Other Methods (including Traditional object-oriented code) ------
        ...
}

interface Recoverable {
        public void Store() { ......}
        public void Restore() { ..... }
}
```

A CO-class (Constrain-Oriented class) is a class that must contain:

1) One or more *constraints*, that is, encapsulated variables of some CO-class or D-class;

2) One or more *CO-methods* (these are always observable);

3) Zero or more private *test methods*;

4) Two public methods *Store*() and *Restore*().

The syntactic structure of a CO-class is:

```
class C implements Recoverable {
        //-----------------Constraints-----------------------------------
        CO-class1 c = new CO-class1();
        D-class2 d = new D-class2();
        ...
        //----------------------CO-methods-----------------------------
        public Boolean M (...) { ....}
        ...
        //-------------------------Test Methods --------------------------
        private Boolean T (...) { ....}
        ...
        // ----------------------Store methods -------------------------------
        public void Store() { ... }
        public void Restore() { ... }
}
```

A *Test Method* is a private, Boolean, parameterized, read-only method without side effects, those methods are used exclusively for testing conditions over their parameters.

A CO-method (Constraint-Oriented method) is a method of a CO-class, and defined as the composition of one or more observable methods of the constraints declared in this CO-class, and zero or more test methods in this CO-class.

The syntactic structure of the definition of a generic CO-method is:

```
public Boolean M (Type1 param1, ...., TypeN paramN) {
        Boolean bi = N (paramX1, ..., paramXn);
        ...
        Boolean bj = T (paramY1, ..., paramYm);
        ...
        return ( bi & ... & bj & ...);
}
```

Where M, N, and T are arbitrary method names, Type1 to TypeN are arbitrary types.

CO-classes and D-classes are different. A CO-class does not encapsulate directly data variables, but only constraints, its CO-methods, which are the only observable methods in the class, can therefore only affect the constraints. Conversely, a D-class directly encapsulates data variables, which can be modified by the observable methods of the class.

COO program: A constraint and object-oriented (COO) program is a program where all user interactions are implemented as calls to the CO-methods of a (top) CO-constraint.

Overall, this approach wraps the traditional object-oriented classes with extra methods and classes, so that the different conditions can be checked, specifically:

- *CO-classes and their instances (CO-constraints)* express *structured (or composite)* constraints involving one or more actions (CO-methods);
- *D-Classes and their Instances (D-Constraints)* express *basic (or primitive)*

constraints involving one or more actions (observable methods) and one or more encapsulated state variables;

- *Observable Methods* express *basic* constraints on the *parameters* of one action *and* on their relations with *state variables*;
- *Test Methods* express *basic* constraints on the *parameters* of one action.

The COO enables us to express a form of functional decomposition that is orthogonal to object decomposition. The functional constraints expressed by COO crosscuts many different types of objects. The main weakness of COO is that the resulting code of applying COO style is not very readable, more guidelines are required to make it easy to understand.

2.3.3 ILOG JRules

ILOG JRules [JLOGREF02, JLOGUSER02] is a general-purpose expert-system generator that combines rule-based techniques and object-oriented programming to help the programmers add rule-based modules to applications.

JRules does not require a proprietary language to define the objects used by the rules, ILOG JRules directly use the Java objects. The design of the application and Java classes are independent of whether ILOG JRules are used.

JRules are <pattern, action> pairs. The pattern serves as a condition, and it is often used

to decide which objects the action should operate on. The pattern matching is performed on ‘working memory’, which consists of all the current ‘working objects’ (Jrule provides commands to add/remove objects into/from the “working memory”). JRule instances are created and put into ‘agenda’ based on the matched object set. Jrule instances in the ‘agenda’ can be fired explicitly.

The agenda is a place that stores rule instances that are ready to be fired. A rule instance is fired when its action part is executed. Rule instances placed in the agenda are said to be *eligible*.

In the agenda, rule instances are ordered according to four criteria that determine which rule should be fired first.

- *Refraction*--A rule instance that has been fired cannot be re-inserted into the agenda if no new fact has occurred, that is, if none of the objects matched by the rule is modified, or if no new object is matched by the rule.
- *Priority*--The second criterion, which is taken into account to decide at which position a rule instance should be placed in the agenda, is the rule priority.
- *Recency*--If two rule instances have the same priority, the rule which matches the most recent object (the most recently asserted, modified or retracted object) will be fired first.
- *Lexicographic order of rule names*--At this level, if two rules have the same priority and the same recency, the next rule to be fired will be the one that appears first if the rules are sorted according to the lexicographic order of their names.

Priority, recency, and lexicographic order are used to resolve conflicts when several rule instances are candidates for firing at the same time.

Jrule also supports temporal reasoning: The “wait**”** statement is used in the condition part of a rule. The wait statement allows you to test if conditions become valid during a designated waiting period. It may also be used to test whether conditions remain true for a waiting period.

Jrules are organized into groups called “packets”. "Packet" is represented as a property of a rule.

2.3.4 R++

R++ is introduced as an extension to C++. Its major new language construct is "*Rule*" [Ahmed97, Litman97]. R++ rules are triggered automatically upon relevant data change. R++ rules can be used to implement crosscutting constraints or rules that monitor data in many different objects. The following sections will first give a simple example, and then will describe the R++ rule, its usage, and its implementation.

This is a simple example of using R++ rules in a C++ class "Person".

```
class Person {
        private:
                String name;
                monitored int age;              // a monitored member data
```

```
                monitored Person *spouse;
                monitored Set_of_p<Person> children;
                rule reflexive_spouse;          // a rule as a member of a class
                rule child_age_check;
};

// If X's spouse is Y, then Y's spouse is X.
rule Person::reflexive_spouse {                 // the definition of a rule
        Person *s = spouse                      // the <condition> part
      =>
        s->set_spouse(this);                    // the < action> part
}

// Check for child older than parent.
rule Person::child_age_check {
        // branch binding: for all 'child' in set 'children'
        Person *child @ children &&
        child->age > this->age
      =>
        cout    << "Error: " << child name
                << " is older than parent "
                << this->name << endl;
}
```

The key points of the new construct ‘Rule’ in R++ are presented as follows.

One important contribution of R++ is that it introduced *rule as member of class*. R++ Rules are introduced as a *natural extension to object-oriented classes*, they support inheritance, overriding, and visibility rules.

R++ rules are also called *path-base rules.* A path-based rule only uses things visible in this object (i.e., data and functional members of itself, and visible members of or pointed to by its members, and so on), it does not violate encapsulation.

A R++ *Rule* is defined as a <*Condition, Action*> pair. The rule is triggered automatically

and implemented in one centralized place (not scattered as in procedural code). The rule also resides in the same place as the data.

A "*Condition*" in a R++ rule can contain:

- Monitored member data
- Null (as a symbol that variables can compare to)
- Function call (shall be side-effect free, because the condition could be evaluated many times before the rule is triggered)
- Qualifier (there are two qualifiers: "all" and "exist")
- Simple binding (bind a value to a variable)
- Branch binding (bind a sequence of values to a variable)
- Global and/or static data

Rules are *triggered by* either "*relevant construction*" or "*relevant change*" of data, i.e., whenever the related data is constructed or modified, the rules shall be re-evaluated.

The major steps in the execution of one rule include: Trigger --> Evaluate --> Fire --> Return. A relevant change or construction triggers the re-evaluation of a rule's condition, if the condition is evaluated to true, then the rule is executed (i.e., fired).

The order of execution for multiple rules follows three principles: "specific-to-general", "depth-first", and "forward-chaining":

- *Specific-to-general*: derived class rules are evaluated before base class rules.

- *Depth-first*: a rule's action can be temporarily interrupted when it performs a triggering event, causing other rule(s) to be evaluated and possibly fired. The original rule's action will be resumed once those other rules complete.
- The order of execution also follows *Forward-chaining*, i.e., a "chain" of rule firings as the action of one rule triggers another rule, and that rule fires and triggers another rule, etc. This is in contrast to "backward chaining" where rules move backward from a desired goal to a state that confirms the goal (e.g. prolog).

R++ rules can be used to enforce invariant, detect constraint violations, express business rules and engineering rules, monitor for important state and events, and propagate information.

R++ is converted to C++ through a Translator. The Translator expand s the predefined get/set methods on the monitored data member. The expanded code evaluates all related rules automatically.

R++ rule is very simple and natural to use. This is its strong point, but the simplicity is also its weak point, e.g., it can not monitor primitive data types, and only top level class and attribute can be monitored.

2.3.5 Exception Mechanism

Many programming languages like C++ and Java have built-in exception mechanisms (see [Java00]). Exception mechanism separates the normal control flow from the exceptional control flow under error conditions. This separation of concerns and centralized exception handling reduce the complexity of programming. Exception mechanism can be viewed as a special form of policy: it provides a mechanism to specify the policies about how to handle faults.

For example, if there is a block of code that uses references to many objects, those references could potentially be NULL. Instead of checking every reference before using it, we can use any references freely without any checking and then use exception mechanism (the 'catch' statement) to specify the NULL reference handling policy:

```
try { // a block of code that uses many references
        …… refX.attributeA …..
        …… refX.attributeA …..
        …… refY.attributeB …..
        …… refZ.attributeC …..
}
catch (NullReferenceException  e) {
        ……  // Exception handling code here
}
```

Examples of the exception handling behaviour:
- logging,
- raise a different exception,
- roll back a database transaction that was started after try,
- free memory created before the exception (to avoid memory leak),
- release a lock that was obtained before the exception,
- raise a different exception, etc.

The native exception mechanisms in programming languages impose certain restrictions on where the exception handling code shall be put (e.g., in Java, ‘catch’ block must follow the ‘try’ block), and the exception handling is at code level. The control follow is sequential and will jump to the catch block once an exception happens.

Programmer can define new types of exceptions, and raise them programmatically. Uncaught exceptions are further propagated to the next higher-level nesting block until there is a corresponding ‘try-catch’ block. The program exits if there is no corresponding ‘catch’ block.

A typical exception mechanism follows the <Event, condition, action> pattern, and syntactically, the code for ‘event’, ‘condition’, and ‘action’ are restricted to be in the same place (e.g., in Java: ‘catch’ must follow ‘try’).

The ‘exception’ mechanism and the typical <condition, action> rule mechanism both require a ‘jump’ of the control flow. When the exceptional situation raises or the condition of a rule is satisfied, the normal control flow will be interrupted (synchronously or asynchronously) by the exception handling code or the rule action code.

Overall, native exception mechanisms in programming languages are restrictive but very simple and elegant. They are meant mainly for error handling (not for arbitrary policy).

2.3.6 AspectJ

Aspect Oriented Programming [AOP01] employs special abstractions known as *aspects* to separate crosscutting concerns throughout the software life cycle. Crosscutting concerns are features that cannot otherwise be cleanly encapsulated in one development artifact and are tangled over several artifacts. Special composition rules combine the aspects with artifacts (crosscut by features encapsulated by the aspects) with respect to reference points in the artifacts. These reference points are termed as join points. Separation of crosscutting features makes it possible to localize changes during maintenance, customization and extension and helps improve productivity and quality. Some aspects can also be highly reusable e.g. domain specific aspects such as those encapsulating platform specific features.

AspectJ is a result of many people's 10 years of research [AspectJ02]. It is an elegant extension to Java programming language that supports Aspect Oriented Programming. AspectJ provides meta-level language constructs that allows the program to manage (monitor, enhance, modify) another program.

AspectJ introduces some new language constructs into Java: Join Point, Point Cut, Advice, Cflow, Introduction, and Aspect. A *Join Point* refers to one of set/get method, constructor, method call, or cflow. *Point cut* combines a collection of join points. A *cflow* is a primitive pointcut that includes all join points within the dynamic control flow of any join point in a specified pointcut. An *Advice* adds additional actions to take at join pints.

Introduction adds additional members into classes. An *aspect* is composed of pointcuts, introductions, and advice. An abstract aspect does not provide full details on every pointcut or method, i.e., some of its pointcuts or methods can be partially defined, so that derived aspects can inherit and reuse its interface, defined pointcuts, and defined methods by filling in the undefined portion.

"Figure 5 AspectJ major concepts" illustrates the relationship among Aspect, Point Cut, Class, Join Point, and Cflow, as discussed in the previous paragraph.

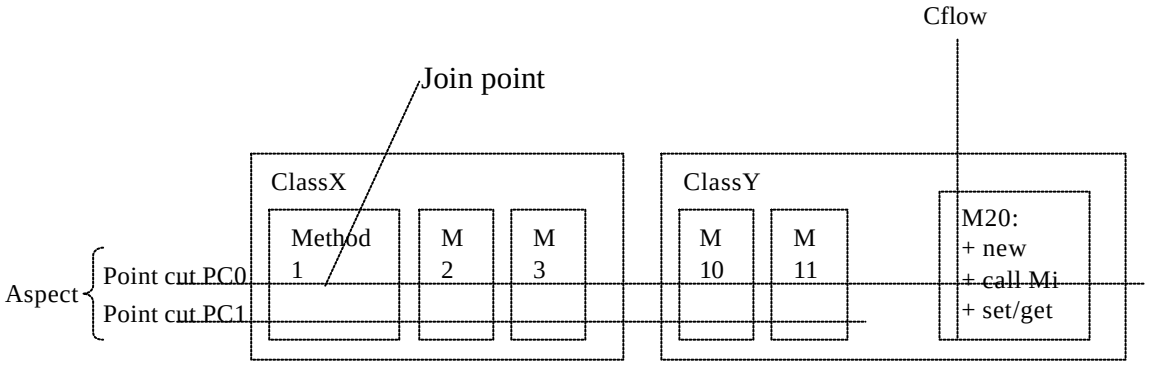

Figure 5 AspectJ major concepts

The concept of 'crosscutting' is best illustrated by the lines (i.e., pointcuts) that cut through Class X and Class Y.

The main issue of this area of work is that there is no methodology on how to develop software by using AspectJ

2.3.7 Conclusion

AspectJ is an extension to Java, which is designed to address crosscutting concerns. AspectJ meets our criteria (see section 2.3.1): it can reference the implementation artifacts for functional features (i.e., reference Java classes and methods through pointcuts) and add extra behaviour (i.e., advice) without actually modifying those referenced artifacts. We will use AspectJ to implement a generic abstract aspect library for NFR concerns (see section 4.3.3) and to implement a chatroom system in our case study (see section 5.6).

The other mechanisms are not satisfactory enough. R++ and JLog Rules can only handle rules whose conditions are system-state-conditions (see section 3.2 for definition of "system-state-condition" versus "program-syntax-condition"). Exception mechanism is only for fault management. COO is not quite readable.

The next chapter will continue to analyze policy mechanisms in a more detailed and generic way, by defining a list of attributes of policy mechanisms.

CHAPTER 3 AN ANALYSIS OF POLICIES

Given the crosscutting nature of both policies and NFRs, policy mechanisms can be good candidates for designing and implementing NFRs. The previous chapter has reviewed some policy mechanisms one by one individually and made our selections informally. This chapter will analyze policy mechanisms more thoroughly. A detailed list of attributes of a policy mechanism will be presented. The characteristics of the various concrete policy mechanisms will be analyzed by filling in the values of those attributes.

This detailed analysis of policies gives us in-depth understanding of policy mechanisms. A generic in-depth understanding of the policy mechanisms has helped us finding the policy mechanisms for our particular problem, and also has helped to ascertain that our choices are the right ones.

The result of this analysis can also be viewed as a list of requirements that a comprehensive policy mechanism could have. This can be the base for future research on better policy mechanisms for designing and implementing NFRs.

3.1 Definitions of Policies

This section presents various definitions of the term “Policy”, and then presents the definition of “Policy” as used in this research work.

3.1.1 Various dictionary definitions of the word "Policy"

These are some dictionary definitions of the term 'policy'.

Webster's New World Dictionary

[Policy]: **a principle**, plan, or course of action, as pursued by a government, organization, individual, etc.

[Principle]: A rule of conduct (Especially the right one).

Merriam-Webster College Dictionary

[Policy]:

1 a : prudence or wisdom in the management of affairs b : management or procedure based primarily on material interest

2 a : **a definite course or method of action selected from among alternatives and in light of given conditions to guide and determine present and future decisions** b : a high-level overall plan embracing the general goals and acceptable procedures especially of a governmental body

3.1.2 Various forms of "Policies" in society

Variations of forms of "Policies" in the society include:

Constitution, Law, by-law, regulation, policy

They prescribe for elements of a society the authorizations (permitted and not permitted)

and the obligations (must do and must not do), the organizational and behavioral constraints and rules within a given context and a given scope. The differences among them are in the degree of formality and the degree of punishment when they are violated.

Some policies put constraints and rules on the definition and application of other policies. E.g., constitution decides how to make a law, some regulations give guideline on how to make other regulations, or on what to do when the policy itself is violated (e.g., a law about "law enforcement"). Some policies refine the detailed aspects of a given policy (A high-level policy is translated into many low-level policies)

A policy states a condition that must always hold true, or a rule that describes actions to be taken within a context, or simply a procedure of actions to be taken. A policy of an organization is stated at the highest possible level where it can be applied. All sub-organizations shall interpret that policy within their own context and execute it accordingly. The form of policy is a 'constraint' or 'rule' or 'procedure', but policy is beyond simply constraint or rule or procedure, policy is stated at a high abstraction level, and is only stated once but enforced everywhere it applies.

3.1.3 Definition of policies as rules

IETF work on policy MIB [Waldbusser00] defines a *Policy* as a rule with this format:

If (*policyFilter*) then (*policyAction*)

A *policyFilter* is program code which results in a boolean to determine whether or not an object is a member of a set of objects upon which an action is to be performed.

A *policyAction* is an operation performed on an object or a set of objects.

3.1.4 Definition of policies as rules and expressions

The DMTF (Distributed Management Task Force) Service Level Agreement (SLA) Working Group [DMTFSLA02] defines *policies* as "rules and expressions that represent management goals, desired system states or the commitments of a Service Level Agreement".

3.1.5 Definition of a policy as either a goal or a strategy to achieve a goal

[Bearden01] defines *policy* as "a specification of management goal or the strategy to achieve a goal". Where *policy goal* specifies what to achieve, *policy rule* specifies how to achieve the goal. It also defines *policy refinement* as "the mapping from policy goal to policy rule".

3.1.6 Policies as Semantically-Crosscutting and Syntactically-Centralized Constraints or Rules

The following is our high-level definition of “policy”. It has been used to direct our research work.

In the domain of computer software programming, a policy is a *semantically-crosscutting* and *syntactically-centralized constraint or rule*.

"Semantically-crosscutting" means that the policy imposes constraints and rules on items in many different modules. e.g., given a UML class diagram of a GUI design, the access-control policy imposes constraints on every class that should be access-controlled based on certain criteria. A straight-forward design is to modify every access-controlled class to enforce this policy. But that is intrusive and not maintainable. Thus we have this “syntaically-centralized" restriction in the definition of policy. "Syntactically-centralized" means that the policy's definition must not be scattered, but must be centralized or modularized in one place.

Note that just a rule or a constraint does not qualify as a policy, at least not a policy of interest to this research work. For example, a simple constraint about an attribute value must be within the range of 0 to 100, assuming the enforcement does not have scattered impact to implementation, then it could be viewed as a policy, but it is a trivial case and of no great interest to this work.

There are many mechanisms that are capable of supporting polcies according to the above definition. As reviewed one by one individually in the previous chapter, they all

look very different and use many differnt terms and differnt mechanisms. The next section will provide a list of attributes for policy mechanisms, that list will be used to characterize those concrete policy mechnaisms, so that we can compare and evaluate those policy mechanisms through a common set of criteria.

3.2 Attributes Of Policy Mechanisms

In order to find concrete policy mechanisms that can be used as the design and implementation artifacts for NFRs, we developed a list of attributes of policy mechanisms. Those attributes are defined in this section. The next section will fill in the values of those attributes for each concrete policy mechanism that we have reviewed in chapter 2. The result of this analysis will be used in the next chapter to select our representations for design and implementation artifacts for NFRs.

The next two subsections will present the attributes of policy mechanisms for a single policy and for a collection of policies respectively. The definition of each attribute is presented and examples are also given to illustrate the concept.

3.2.1 Attributes Of Policy Mechanisms For A Single Policy

This is the list of attributes that we will discuss below:

1. Domain Specific	2. Operational	3. Scope
4. Structure	5. Stateless	6. Prioritized
7. Presentation Style	8. OPI Type	9. Condition Type
10. Development Phase	11. Active	12. Triggering Direction
13. Triggering Focusness	14. Data Location	15. Run Time Changeable
16. Encoding Method	17. Code Generated	18. Parameterized
19. Delayable On Conflict	20. Cancelable On Conflict	21. PDP and PEP distributed
22. Modularity		

A[1] Domain Specific

This attribute indicates whether this policy mechanism is for a specific domain or for generic programming. A policy mechanism can be for generic software development (e.g., OCL, AspectJ), or for a specific domain (e.g., Ponder [Damianou01] is for Security/RBAC, PIB is for network management).

A[2] Operational

Policies can be classified into two major categories: Non-operational Policies, and Operational Policies. A constraint is a form of non-operational policy.

Non-operational policies (also called 'goal' in [Bearden01]) do not have actions in them,

and are usually side-effect free (e.g., OCL expression is side-effect free). They are useful at the modelling level. e.g., constraints are useful at the modelling level. An example of non-operation policy is:

Automated Teller Machine should respond within 10 seconds.

But when a non-operational policy is mapped to the code, the implementation will have to make decisions to make it operational. Because even though the action is not mentioned explicitly at the specification level (you can consider it as an incomplete specification, but that happens frequently in the real life), the piece of code that has no side-effect is equivalent to nothing.

Operational Policies have actions in them, typically they are 'rules', i.e., a <condition, action> pair. They are useful at both the implementation level and the modeling level. This is an example:

If the ATM does not respond within 10 seconds, then notify the teller by raising both visual and audio alarms.

Non-operational policies can be viewed as a degenerated or special case of Operational Policies (i.e., constraints can be translated into rules with action part is always "raise fatal error exception").

Non-operational policies typically are high level policies (specify 'what') and will be eventually mapped to operational policies (specify 'how'). The relationship between non-

operational policies and operational policies shall be specified during policy-refinement process.

A[3] Scope

The scope of a policy can be inter-object (i.e., about several objects) (e.g., AspectJ), or intra-object (i.e., about one object) (e.g., R++); Local (i.e., in one machine), or distributed (across many machines) (e.g., PIB); System behaviour (i.e., about the behaviour of the system being developed), or DesignPhase (i.e., about the behaviour of the designer) (e.g., design patterns).

An "intra-object policy" uses and impacts local data (also known as "access-limited"). For example, in R++, UML/OCL, and Object Oriented Constraint Programming style, the policies (rules and constraints) follow the visibility rule of object-oriented language. This type of rules is also called "intra-object" rules.

An "inter-object policy" uses and impacts data globally (in the same process or address space). For example, in ILog, the rules work on in-memory data, it could be any objects. This type of rules is also called "inter-object" rules. ILog rules use the condition (it is called "pattern" in ILog) to decide which objects should be acted upon (see ILog Rule Language User Manual).

A[4] Structure

The policy structure can be "event, condition, action" (e.g., ILOG JRules), "condition,

action" (e.g., AspectJ, R++), or just a constraint (e.g., OCL).

A[5] Stateless

A policy is stateless when the execution of this policy does not affect the next execution of this policy or other policies.

A[6] Prioritized

Whether the policies in this policy mechanism is prioritized or not. Priority can be used for conflict resolution.

A[7] OPI Type

OPI type can be one or the combination of Obligation, Permission, and Interdiction [Barbuceanu98].

Permission, obligation, and interdiction can be converted from each other:

$$P(X) = +\ O\ (+X)$$

$$I(X) = +\ P\ (X)$$

Permission can be viewed as the negation of non-action; Interdiction can be viewed as the negation of permission [Barbuceanu98]. But some mechanisms provide explicit support for all forms of permission, obligation and interdiction. Others do not.

For example, Ponder supports obligation, positive and negative permission and subject-enforced refrain; OCL supports obligation and positive or negative permission, Java/C++

exception mechanism supports negative permission. All policy mechanisms support at least positive obligation of the object (i.e., what the system should do).

Policy can be used in a positive way: to specify what the system should do (positive obligations), or to specify what the system is permitted to do (positive permissions). Policy can be also used in a negative way: to specify what the system should not do (interdiction or the obligation of negation of the action).

Policy can be used to specify what the system should do or is permitted to do. It can also be used to specify what an external entity should or is permitted to do onto this system.

A[8] Presentation Style

In the logical style, the policy is defined as logic statements. This is typical in expert systems and traditional rule-based programming languages.

In the procedural style, the policy is defined procedurally, typically in the syntax of a popular programming language. This reduces the barrier to introducing new language structures.

In the object-oriented style (e.g., R++, AspectJ), the policy is defined procedurally but in association with objects and following certain visibility rules, typically by extending a popular programming language (e.g., R++ extends C++ and AspectJ extends Java).

OCL is a hybrid, it extends UML Class diagrams, but it has some logical expressions.

A[9] Condition Type

There is usually a condition in the structure of the policy, the condition could be about the syntax of another piece of program (e.g., AspectJ), or about the system state (e.g., R++). If the "condition type" is "program syntax", then it has meta-programming ability and can be used to constrain, enhance or modify another piece of program.

For example, in AspectJ, you can specify a rule like this: "For all methods of classes in package X, if the name of the method matches pattern Y, then run function Z to validate the input parameters of that method". This rule references multiple methods in another piece of code, it is a meta-level programming statement. The condition in the rule is about the syntax of another piece of code rather than the values of particular variables.

The condition types "program syntax" and "system state" could be combined and used in one policy mechanism, even though most existing policy mechanisms tend to emphasize on only one of them.

A[10] Development Phase

A policy mechanism can be used as design Specification (e.g., OCL, Ponder [Damianou01]) or Code (e.g., AspectJ, R++, and Exception in Java).

A[11] Active

An active policy is enforced automatically, while a passive policy needs to be triggered (like a function call) by an external entity.

A passive policy can only be explicitly triggered. You can view a procedure with only a conditional statement as a rule or policy, but the procedure will not run until it is called.

The goal-driven rules in prolog[Clocksin87] is actually passive, rules always explicitly mention other rules in their definitions so that they can be triggered.

An active policy will be triggered automatically whenever appropriate, e.g., in R++, the rule is triggered whenever the relevant data referenced in the rule's condition are changed, no explicit triggering to the rules are required. An R++ rule can be triggered even if it is not mentioned anywhere else (beyond where it is defined).

Another example of automatically triggered rule is the 'triggers' that can be created on a database server [Oracle99]. The DBMS guarantees that a 'trigger' is automatically called whenever the specified situation arises (e.g., relevant data is created, updated, or deleted). The syntax of a trigger on a typical relational database:

create trigger xyz on delete begin <body> end

A[12] Triggering Direction

A data-driven policy is triggered by the change of the data that are referenced in its condition. When the action in one rule changes some other data, some other policies may

be triggered. This kind of effect is also called "forward-chaining". R++ rules and ILog rules are both data-driven.

A goal-driven policy is triggered by the request to satisfy a given goal. To satisfy the goal as specified by a policy, other policies may be triggered. It is also called "backward-chaining". An example is the prolog rule.

A[13] Triggering Focusness

Loosely-focused policy-triggering criterion may evaluate a rule's condition even if it is not necessary (e.g., no data used in the rule's condition is changed) (e.g., COO). Tightly-focused policy-triggering criterion only evaluates a rule if it is necessary (e.g., the data used in the rule's condition are changed, or called by another rule). Most operational policy mechanisms have a tightly-focused policy-triggering criterion.

A[14] Data Location

The data location of a policy can be classified into two categories. Policy for persistent data: Policies are triggered by changes in persistent data (e.g., exception mechanism in Java and C++). Policy for in-memory data: Policies are triggered by the changes of in-memory data (e.g., PIB).

A[15] Encoding Method

Encoding methods include "as code" or "as metadata", i.e., a policy can be represented by code (e.g., R++, AspectJ), or by metadata and interpreted at run time (e.g., PIB).

A[16] Code-generated vs Engine-based

In engine-based (i.e., interpreted) policy implementation scheme, there is a predefined and fixed rule engine (e.g., PIB) that reads rules as data and processes the rules.

In the code-generation-based (i.e., compiled) policy implementation scheme (e.g., AspectJ, R++), there is no fixed code, the code is generated based on the rule definition. When the rules change, the code changes too.

A[17] Run-time Changeable

Some policy mechanisms allow policies to be changed at run-time (e.g., PIB), others do not (e.g., AspectJ, R++, exception mechanism in Java/C++).

A[18] Parameterized

To support run-time changeable policy, either the entire policy is encoded as metadata (e.g., PIB), or the policy is parameterized to allow update at run time(e.g., ILOG JRules).

If a policy mechanism supports parameterized policies, and it implies that it supports run-time changeable policies. But not the other way around, since there are other ways to make things run-time changeable (e.g., in Java, classes can be loaded dynamically).

A[19] DelayableOnConflict

Whether the execution of this policy can be delayed upon conflict with another policy's

execution (also see: Conflict resolution method).

A[20] CancelableOnConflict

Whether the execution of this policy can be cancelled upon conflict with another policy's execution (also see: Conflict resolution method).

A[21] PDP and PEP Distributed

This attribute indicates whether PDP (Policy Decision Point) and PEP (Policy Enforcement Point) [Boutaba01, Corradi01] are separated into different machines.

A[22] Modularity

AspectJ provides extremely high modularity, each aspect is in a separate module. While Java/C++'s exception mechanism has relatively low modularity, it is slightly scattered (but still better than totally scattered code when not using exceptions at all).

3.2.2 Attributes Of Group Policy Mechanisms

This section presents a list of attributes that a group policy mechanism may have. A group policy mechanism is the mechanism that manages a collection of policies. The definition of each attribute is presented below and examples are also given to illustrate the concept.

This is the list of attributes that we will discuss below:

23. Organization Type	24. Transactional	25. Conflict Resolution Method
26. Allow Parallel Execution	27. Policy Combination Method	

A[23] Policy package organization type

In a system, there could be hundreds or thousands of policies. Some policy mechanisms do not provide any means to organize the entire set of policies, it is just a flat set. Some other policy mechanisms offer a way to organize the entire set of policies. A set of policies can be organized into a package. The organization type can be hierarchical (e.g., CIM Core Policy, PDL [Kanada01]), associated with classes (e.g., OCL, R++), an independent module (e.g., AspectJ), or simply an unordered collection (e.g., Java's exception mechanism).

A[24] Transactional

Policies in the same package are in one transaction, i.e., they are either all-executed, or none-executed (e.g., ILOG JRules). Most policy mechanisms do not have this feature.

A[25] Conflict Resolution Method

Some policy mechanisms do no allow conflicts among policies at run time (e.g., AspectJ, R++) (then conflicts shall be detected by tools, e.g., compilers).

Some policy mechanisms allow conflicts at run time, and the run-time conflicts are resolved by either canceling one of the conflicting policy (e.g., based on priority or

recency in ILOG JRules) or delaying and retrying (the conflicting situation may disappear after a while) (e.g., PDL [Kanada01]). The priority of the policy can be used to decide which one to cancel or delay (e.g., ILOG JRules).

A[26] Allow Parallel Execution

Policies in the same package can be executed in parallel (e.g., PDL [Kanada01]), or only sequentially (e.g., R++, AspectJ).

A[27] Policy Combination Method

Two or more policies can be combined in various ways to form a new policy: sequentially, or in parallel, or conditional (one policy takes effect under one condition and the other policy takes effect under another condition), or iterational (i.e., policies can be applied repeatedly for a specified number of times).

Finally, the last attribute is about something that is external to the mechanics of the policy itself.

A[28] Supported by Language or Tool

This attribute indicates whether this mechanism has been supported by a language or a tool. This is usually a concern during experimental work in research and practical work in the industry.

3.3 Positioning of Various Concrete Policy mechanisms

Now that we have the list of attributes that a policy mechanism may have, we can go back to summarize the characteristics of various forms of policies that were discussed in "Chapter 2".

The following table presents a summarization of all the characteristics of the various forms of policies. This summarization helps us to select the right mechanism for our particular problem right now. It shall also be able to direct building new and better policy mechanisms for research and development in the future.

For reader's convenience, some key features of each mechanism are highlighted in bold font.

Characteristics	AspectJ	R++	Exception	DMTF CIM Core Policy Model	PDL
[1]Domain	generic	generic	generic	QoS policy in networks	real time apps
[2]Operational	Yes	Yes	Yes	Yes	Yes
[3]Scope	Local (intra-object or inter-object), non-distributed	Intra-object	Intra-object	Distributed	generic, not specified

[4]Policy Structure	Condition-action	Condition--action	Condition-action	Condition-action	event-cond-action
[5]Stateless	Yes	Yes	Yes	Yes	**No**
[6]Prioritized	Yes	No	No	Yes	Yes
[7]Presentation Style [8]OPI Type	Procedural Positive Obligation	Procedural Positive Obligation	Procedural **Negative Permission**	Logical Obligation and +/- permission	procedural + logical Obligation
[9]Condition Type	**Program syntax**	System state	System state	System state	System state
[10]Software development phase	coding	Coding	Coding	Information modeling	Specification language
[11]Active	Yes, at compile time	yes	yes	yes	yes
[12]Triggering focus-ness	Tightly focused	Tightly focused	Tightly focused	Tightly focused	tightly focused
[13]Triggering direction	Data-driven bottom-up	Data-driven bottom-up	Data-driven bottom-up	Data-driven bottom-up	Data-driven bottom-up
[14]Data location [15]Run-time changeable	In memory No	In memory no	In memory no	Persistent distributed Yes	persistent distributed no
[16]Encoding method	As code	As code	As code	As data	As code
[17]Code-generated	Yes	yes	no	no	no
[18]Parameterized	Yes	yes	no	yes	Yes
[19]Delayable upon conflict	No	No	No	No	**Yes**
[20]Cancelable upon conflict	No	No	No	No	**Yes**
[21]Distributed PDP and PEP	No	no	no	**Yes**	yes
[22]Modularity	**High**	medium	**low**	Medium	medium

[23]Policy package organisation type	**Hierarchical, independent modules**	**Within C++ class**	Set	Hierarchical	Hierarchical
[24]Transactional	No	No	No	No	No
[25]Conflict resolution method	Predefined precedence and "dominate" keyword	None	None	Priority-based	**Through monitors (at run time)**
[26]Allow parallel execution	No	No	No	No	Yes
[27]Policy combination method at run time	Sequential	Sequential	Sequential	Sequential	Sequential
[28]Language/Tool supported	yes (extending Java)	yes (extending C++_	Yes (native to C++ & Java)	No	no

Table 1 Positioning Various Policy Mechanisms

Characteristics Table continued:

Characteristics	Ponder	COO	PIB	OCL	ILOG JRules
[1]Domain	RBAC (security)	Generic	Policy-based network mgmt	**Generic**	Generic
[2]Operational	obligation: yes; others: no	yes	yes	No, but can be simulated by post-condition	Yes

[3]Scope	Generic, not specified	intra-object	Distributed	Intra-object	**Inter-object**
[4]Policy Structure	Oblig: event-cond-action; Other: condition/ constraint	condition-action	event-condition-action	Condition /constraint	Condition (pattern) -action
[5]Stateless	yes	yes	Yes	Yes	Yes
[6]Prioritized	no	no	Yes	No	Yes
[7]Presentation Style	declarative language	**Logical + Procedural**	Procedural	**UML + logic**	Procedural
[8]OPI Type	**Obligation and +/- permission, also refrain (subject-enforced) & delegation**	positive obligation	Positive obligation	Positive or negative permissions (constraint) and obligation (post condition)	Positive obligation
[9]Condition Type	System state	System state	system state	System state	System state
[10]Software development phase	Specification Language	Coding	Specification & implementation	**Analysis & specification**	Coding
[11]Active	N/A, Specification	No	N/A, Sepcification	Yes	Yes
[12]Triggering focus-ness	not specified	loosely focused	tightly focused	n/a	Tightly focused
[13]Triggering direction	data-driven bottom-up	Goal-driven top-down	Data-driven bottom-up	n/a	Data-driven bottom-up
[14]Data location	not specified	in memory	**persistent + distributed**	Not specified	in memory

[15]Run-time changeable	not specified	no	Yes	No	no
[16]Encoding method	As data	as code	**as data**	as code	as code
[17]Code-generated	not specified	No	no	No	yes
[18]Parameterized	yes	yes	Yes	No	yes
[19]Delayable upon conflict	no	No	no	No	**yes**
[20]Cancelable upon conflict	no	No	**yes**	No	No
[21]Distributed PDP and PEP	Not specified	No	**yes**	No	no
[22]Modularity	Medium	medium	medium	Medium	Medium
[23]Policy package organisation type	**hierarchical, grouped into roles**	**hierarchical, extra layer of java classes**	Hierarchical	**Within class: Associated with UML model elements**	grouped into 'packets'
[24]Transactional	No	No	Yes	No	yes
[25]Conflict resolution method	**thru static analysis of spec**	None	priority-based	Thru static analysis of spec	**priority, recency, and lexicographic order**
[26]Allow parallel execution	yes	No	Yes	No	No
[27]Policy combination method at run time	sequential	sequential	branch + sequential	Not defined	Sequential
[28]Language/Tool supported	yes (Tool)	no	no	Yes (tool)	yes (tool)

Table 2 Positioning Various Policy Mechanisms (Cont.)

3.4 Conclusion

The detailed dissection of policy mechanisms provides a benchmark for future research and a framework to understand policy mechanisms better. The list of attributes of a generic policy mechanism helps the evaluation of any particular forms of policy mechanisms, or serves as a checklist for the elicitation of requirements when you are looking for a policy mechanism or developing a new policy mechanism. New policy mechanisms might be required when the existing ones are not satisfactory for designing and implementing NFRs in a particular domain.

The result of the analysis of a generic policy mechanism and the positioning of those concrete forms of policy mechanisms will be used in the next chapter to explain why OCL is selected and why it needs to be extended, and to explain why AspectJ is used to implement NFRs.

CHAPTER 4 POLICIES AS ARTIFACTS OF DESIGN AND IMPLEMENTATION FOR NFRS

Let us be reminded that our objective is to achieve the *Separation of Concern* for NFRs at the design level and the code level. *Separation of Concern* is one major principle in the discipline of Software Engineering to manage the complexity of software systems [Dijkstra76]. At the requirement level, NFRs are considered separately. Our work is intended to map this separation into the design level and code level, i.e., to create design artifacts that are just for NFR concerns, and to create code modules that are just for NFR-designs. Because the design and code for NFRs are modularized and separated, we can easily understand the design and the code, easily trace across them, easily add or modify NFRs (non-invasive adaptation and evolution), and potentially even reuse the modularized design and code for NFRs (they can not be easily reused if they are scattered in many parts of the system).

We use two forms of policy mechanisms as the artifacts for implementing non-functional requirements: Policy Extension to OCL (PEOCL) at the design level and Aspect at the code level.

OCL and AspectJ have been described in sections 2.2.1 and 2.2.3 respectively. This chapter will discuss further the details of PEOCL and Aspect, why they are adopted to represent design and implementation artifacts for NFRs, and how they can be used. The sections will also illustrate through examples how NFRs are mapped to PEOCL Policies, and then to Aspects.

4.1 Extending OCL With The UML Meta Model And The NFR Ontology

This section describes and rationalizes PEOCL as the design artifacts for NFRs.

4.1.1 Why OCL And Why Extending OCL

For reader's convenience, the differentiating characteristics of OCL [OCL97] are extracted from Table 2 in the previous chapter, and outlined in Table 3 below. The main features are highlighted in bold font.

Characteristic Name	Characteristic Value For OCL
1. Policy package organisation type	**Within class: Associated with UML model elements**
2. Policy combination method at run time	Not defined
3. Conflict resolution method	Through static analysis of the spec
4. OPI Type	Positive or negative permissions (constraint) and obligation (post condition)
5. Scope	Intra-object
6. Policy Structure	Condition/constraint
7. Operational	No
8. Prioritized	No
9. Condition Type	**System state**
10. Presentation Style	**UML + logic**
11. Encoding method	As code
12. Modularity	Medium
13. Language/Tool supported	Yes
14. Domain	**Generic**
15. Software development phase	**Analysis & specification**

Table 3 Differentiating Characteristics Of OCL

OCL is chosen as the base of the representation of design artifacts for NFRs, because:

- It is associated with UML class diagrams, which is the mainstream notation for representing design artifacts.
- It uses a combination of UML and logic, so it is formal but not too formal. It does not require the user of the notation to have a strong mathematical background like what pure logical programming requires.
- It is at the specification-level
- It is not specific to a particular domain

A study of the characteristics of OCL reveals that two characteristics of OCL are not satisfactory for the purpose of representing design artifacts for NFRs. First, the "*condition type*" of OCL is limited to "*system state*", what is needed is "*program syntax*" (see section 3.2 "Attributes Of Policy Mechanisms " for the definition of "condition type"). i.e., only the values held by the attributes of the classes or the instances of classes can be referenced. However the crosscutting nature of NFRs requires the ability to reference a collection of model elements (classes, methods, attributes, etc.) in a UML class diagram for functional features. We need the meta-programming-level expressive power to do that. That is why the UML Metamodel will be used as part of the OCL expression. Second, OCL is not domain specific. That is good but also it is too generic and inconvenient for describing many NFR level concepts, thus we will reuse the ontology of NFR Framework and Quality Attribute Taxonomy. With those two extensions to OCL, PEOCL (Policy Extension to OCL) can be used to easily describe constraints imposed by NFRs on the UML class diagrams for FRs.

The next two sections will describe NFR ontologies and the UML Metamodel. Examples of using PEOCL will also be given to illustrate the concepts.

4.1.2 Extending OCL With The NFR Ontology

"Appendix: NFR Ontology" presents a portion of the concepts in the NFR ontology that are used in our case study (For the complete list, see [Chung00b]). The addition of the NFR Ontology into PEOCL is intended to make it more convenient to express NFR's constraints on FR's design artifacts. The terminology in the NFR ontology is allowed to appear in PEOCL expressions.

For example, the following PEOCL policy uses the keyword "encrypted" from the NFR ontology:

```
<designPolicy name="Outgoing Message Encryption Policy">
  <category>Security</category>
  <target> DataOutputStream::writeUTF(msg : String) </target>
  <preCondition>
      <oclExpression> encrypted (msg)   </oclExpression>
  </preCondition>
</designPolicy>
```

The above PEOCL expression specifies that:

The pre-condition of the method "writeUTF" of class "DataOutputStream" is that the parameter "msg" must be encrypted. "DataOutputStream" is a class in the UML class

diagram for FRs. "encrypted" is a terminology from the NFR ontology and is used as a predicate in this PEOCL expression. The exact definition of "encrypted" is already decided by the NFR ontology.

4.1.3 Extending OCL With The UML Metamodel

The UML Metamodel [UMLMeta97] is used to define UML. It can be used to specify the UML language constructs at the meta level. Example class names in UML Metamodel include: class, attribute, operation, etc. We reuse that work to gain the ability to represent collections of UML model elements (e.g., a collection of classes, methods, or attributes). This is essential to the crosscutting nature of NFRs, which usually refers to many components of the system.

For example, the following PEOCL policy uses a class named "Method" from UMLMetaModel's package "Core":

```
<designPolicy name="Trace All Method Calls Policy">
 <category>Maintainability</category>
 <target>
  UML.MetaModel.Core.Method::invoke()
 </target>
 <postCondition>
    <oclExpression>
      (log - log@pre) -> notEmpty
    </oclExpression>
 </postCondition>
</designPolicy>
```

The above PEOCL policy specifies that:

The post condition of method "invoke" of class "UMLMetaModel.Core.Method" is that

the new log contains more information than the original log before calling that method. Since UMLMetaModel.Core.Method is a meta-level class, all methods in a UML Class Diagram are instances of UMLMetaModel.Core.Method. So basically this PEOCL policy specifies that each method invocation should be logged.

4.1.4 PEOCL Syntax And Semantics

A PEOCL specification has one or many designPolicy specifications. A designPolicy has an attribute "name", and has optionally these items: category, target, introduction, preCondition, postCondition, invariant, and zero or more designPolicies. The following UML diagram (Figure 6) presents a graphical view of this structure.

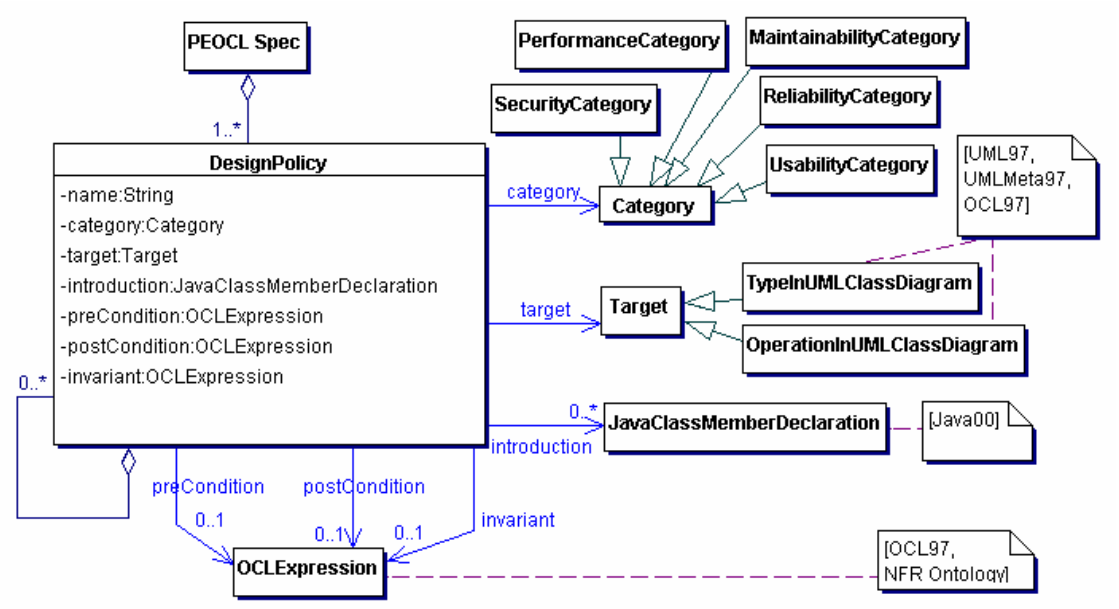

Figure 6 UML Class Diagram For PEOCL DesignPolicy Structure

The following DTD[1] outlines the syntax of PEOCL[2].

```
<!--  PEOCL DTD -->
<!DOCTYPE peocl [
 <!ELEMENT peocl (designPolicy+)>

 <!ELEMENT designPolicy (category?, target?, introduction?, preCondition?,
                         postCondition?, invariant?, designPolicy*)>
 <!ATTLIST designPolicy name CDATA #REQUIRED>

 <!ELEMENT category (#PCDATA)>
 <!ELEMENT target (#PCDATA)>
 <!ELEMENT introduction (#PCDATA)>
 <!ELEMENT preCondition (oclExpression)>
 <!ELEMENT postCondition (oclExpression)>
 <!ELEMENT invariant (oclExpression)>
 <!ELEMENT oclExpression (#PCDATA)>
]>
```

The DTD defines the high level structure of the PEOCL specification.

The *category* of a PEOCL DesignPolicy can be (but not limited to) "performance", "security", "maintainability", "reliability", and "usability".

The *target* of a PEOCL designPolicy is either the name of a type (e.g., class, interface) or the signature of an operation in the UML class diagram. The types and operations in the UML Metamodel can also be used as discussed in the previous sections.

---

[1] XML Schema can be used to define the syntax as well. The syntax here is simple enough to be easily represented by either notations.

[2] All the XML specification has been checked by using XMLWriter [XMLWriter02] to ensure the well-formedness and validity with respect to the given DTD.

The *introduction* contains some Java class member declaration. "Introduction" is optional. Sometimes we have to add new attributes or methods into the existing classes to express design for newly added NFRs. "Introduction" provides us with a way to add new members without touching the original class diagrams for functional features, and allows us to put all specification mapped from the same NFR in the same PEOCL policy. The following DesignPolicy is an example of "introduction". In order to support a newly added NFR: role-based GUI access control, we have to add a new data member in the existing class "LoginDialog" (see the case study for more details).

```
<designPolicy name="Introducing logingRole">
  <category>Security</category>
  <target>LoginDialog</target>
  <introduction> String loginRole; </introduction>
</designPolicy>
```

The syntax of the variable or method declaration in "introduction" follows that of Java class member declaration.

The *preCondition*, *postCondition*, and *invariant* are all expressed in OCL expressions. [OCL97] has the BNF definition for OCL Expressions. The predicates from NFR ontology can be used within OCL expressions. The preCondition, postCondition, and invariant are all optional.

Finally, the designPolicy can be a composite policy, i.e., it can have some other designPolicies as sub-policies. For example, the following designPolicy is a composite

one:

```
<designPolicy name="Message Encryption Policy" >
  <category>Security</category>
  <designPolicy name="Outgoing Message Encryption Policy"/>
  <designPolicy name="Incoming Message Encryption Policy"/>
</designPolicy>
```

4.1.5 Usage of PEOCL

Given a UML class diagram as the design artifact for FRs, NFRs can be mapped to invariant on those classes, pre and post conditions on a collection of operations and methods. PEOCL can be used to express invariant, pre and post conditions.

The examples in the previous section already demonstrated that PEOCL can be used to express the pre and post conditions on a collection of methods. We will give another example below to show how to use PEOCL to express the invariant in a class.

```
<designPolicy name="GUI Access Control Core Policy">
   <category>Security</category>
   <target>ChatroomClientWindow</target>
   <invariant>
       <oclExpression>
            self.loginDialog.loginRole = "admin" implies
            self.menuItemManageUsers.enabled  = true
       </oclExpression>
   </invariant>
</designPolicy>
```

The above PEOCL policy specifies that:

If the user's login role is "administrative role", then the "ManageUsers" menu item is always enabled (See the case study in the next chapter for more details).

4.2 Mapping NFRs to PEOCL Policies

Now that we have discussed what PEOCL is and how to use it, we can discuss how to map NFRs systematically to PEOCL design policies. We are mainly using the work from NFR Framework [Chung00a, Chung00b, Gross00, Chung94] and Quality Attributes Taxonomy [Babacci95, Kazman99, Kazman00]. "NFR framework" provides the notation to represent the refinement process. "Quality Attributes Taxonomy" provides the possible refinements.

The steps to perform the mapping from NFRs to PEOCL policies are:

- Refine high level NFRs into detailed NFRs if necessary
- Decide the design strategy to meet the detailed NFRs: Should the design artifact be in design policies that are separated from the original design artifacts or should the new design modify the original design artifacts directly?
- If the strategy is to use separate design policies, then

  Represent the design in design-policies by using natural language.

  Else

  Follow the traditional object oriented methodology.
- Refine the design policies in natural language into more detailed design policies if necessary
- Represent the natural language version of policy in PEOCL design policies
- Map the PEOCL design policies into aspects: This last step will be discussed in the

following sections.

Figure 7 illustrates the mapping procedure from NFRs to PEOCL design policies graphically.

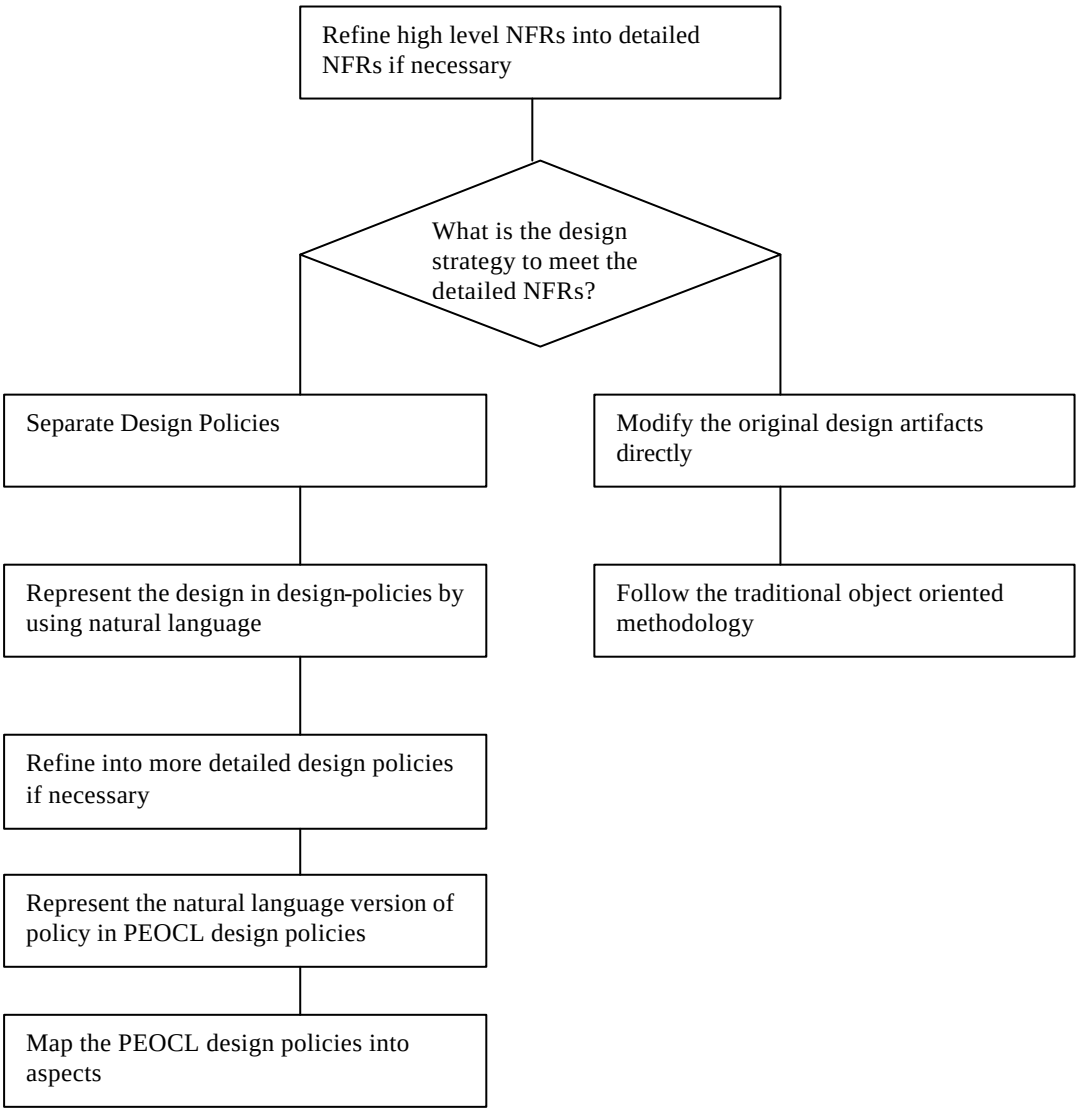

Figure 7 Mapping From NFRs To PEOCL Design Policies

Figure 8 illustrates an example of this mapping process. This example is taken from our

case study (see chapter 5 for more details).

The high level NFR is "security", i.e., the chat room system shall be secure. Based on the 'concerns' and 'methods' in the Quality Attribute Taxonomy (reference "Figure 3 Security Taxonomy" in section 2.1), we refine the security NFR into four detailed NFRs for the chat room system:

- *GUI Access Control*: Only those GUI items that the user has permission to use are enabled, e.g., non-administrative user cannot modify other user accounts, so the user management GUI is disabled for non-administrative users
- *Message Encryption*: Messages transmitted over the network shall be encrypted
- *Login/Logout*: Users shall go through a login procedure which prompts for user name and password, and only authenticated users can proceed to use the chat room
- *Block out user*: The administrative user shall be able block out a bad user

We decide that only "GUI Access Control" and "Message Encryption" will be mapped to design policies. We will modify the original design and code to meet the other two NFRs, because they can be implemented in relatively straightforward independent modules by using the traditional object oriented method (they could be done through policy mechanisms as well, but there is no obvious benefit in doing so).

Then the "GUI Access Control" and "Message Encryption" NFRs are mapped to design policies in natural language, as illustrated in "Design Policy 1.1" and "Design Policy 1.2" respectively (Figure 8).

The lengthy details of the PEOCL policies are omitted for clarity. Section 5.5 will provide the complete PEOCL design policies for chat room NFRs.

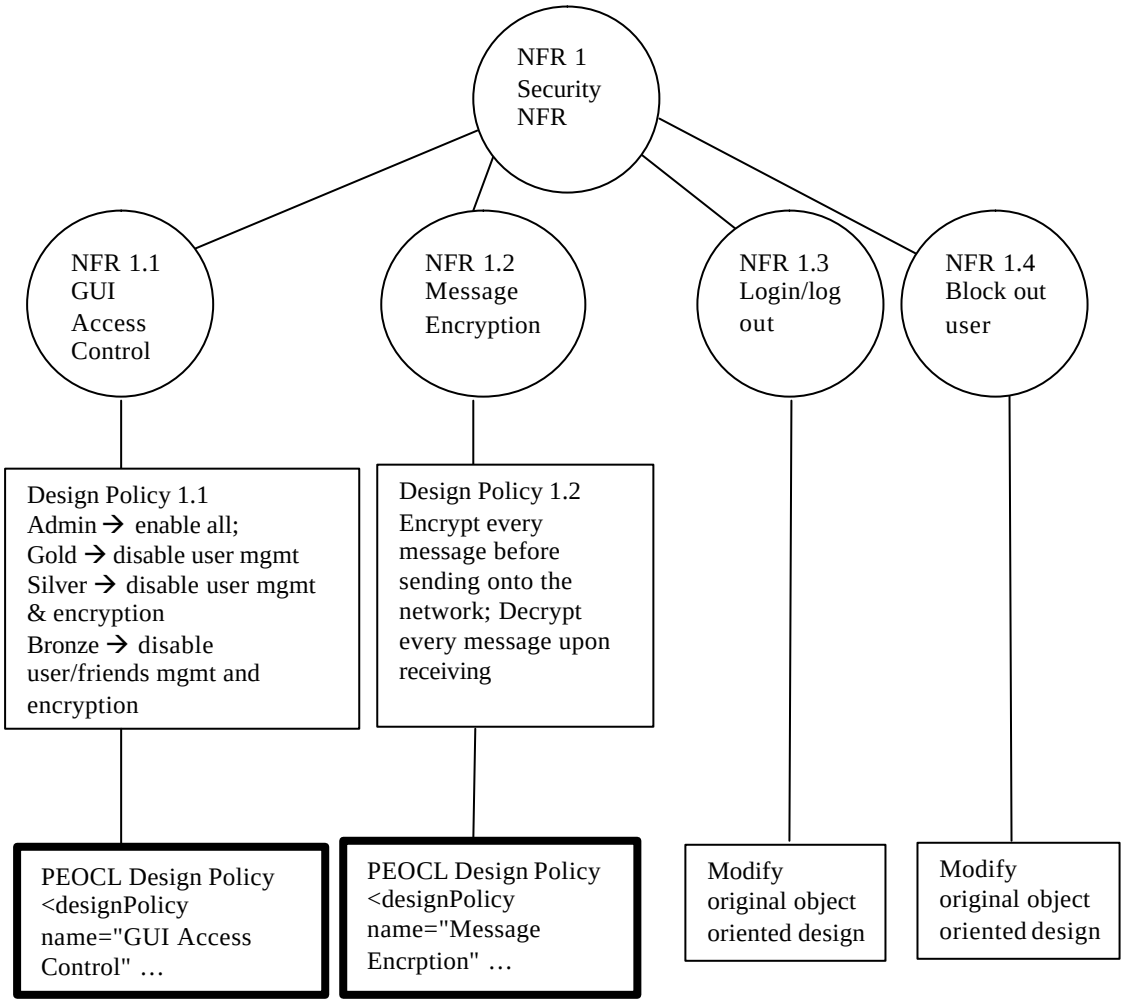

Figure 8 Design For Security NFR

4.3 Aspects And Abstract Aspect Library For NFRs

4.3.1 What Are Aspects

For reader's convenience, the differentiating characteristics of AspectJ are extracted from Table 1 in the previous chapter, and outlined in Table 4. The main features are highlighted.

Characteristics	AspectJ
1. Policy package organisation type	**Hierarchical modules that support inheritance and information hiding**
2. Policy combination method at run time	Sequential
3. Conflict resolution method	Predefined precedence and "dominate" keyword
4. OPI Type	Positive Obligation
5. Scope	Local (intra-object or inter-object), non-distributed
6. Policy Structure	Condition-action
7. Operational	Yes
8. Prioritized	Yes
9. Condition Type	**Program syntax**
10. Presentation Style	Procedural
11. Encoding method	As code
12. Modularity	**High**
13. Language/Tool supported	Yes
14. Domain	**Generic programming**
15. Software development phase	**Coding, as extension to Java**

Table 4 Differentiating Characteristics of AspectJ

An Aspect is a relatively new unit of programming module that crosscuts traditional boundaries like subroutines, functions, procedures, methods, classes, and packages. Section 2.3.1 has described some of the major concepts in AspectJ, an aspect-oriented

extension to the Java programming language.

The main feature or strength of this new modularity "Aspect" is its "crosscutting" characteristic. "Crosscutting" is with respect to the main modularity "class" in the current mainstream methodology "Object Oriented Method".

4.3.2 Why Aspects

According to the object-oriented methodology, the functional requirements are modeled into classes, and those classes will be further refined into design level classes, and eventually implemented by classes in an object-oriented programming language. The fundamental modularity of object oriented method is "class" [Martin98].

A NFR covers typically many classes of the system. A PEOCL expression can specify a policy on a collection of classes and/or methods, the expressive power introduced by the addition of UML metamodel. Aspect's crosscutting power fits well as a mechanism to implement PEOCL policies for NFRs. In particular, we will use AspectJ in all our examples and case study. As listed in "Table 4 Differentiating Characteristics of AspectJ", AspectJ supports "program-syntax" as "condition type" (see section 3.2), and it has extremely high modularity, it supports "aspect" as a unit of programming that has information-hiding and inheritance features. AspectJ has also managed to put "aspect" as a minor extension (syntax-wise) to the popular object-oriented language Java. "Aspect" (and in particular: AspectJ's aspect) fits into our needs of implementing PEOCL policies

for NFRs because all the three things -- NFR, PEOCL, and Aspect -- have a common characteristic: crosscutting.

4.3.3 A Generic Aspect Library For Common NFR Concerns

Similar to reusing NFR Ontology and UML Metamodel to easy the specification of design artifacts for NFRs, we want to find a way to ease the coding of aspects. We developed an abstract aspect library for common NFR concerns, this shall help the mapping from PEOCL policies to aspects and promote reuse of generic aspects.

This section presents some sample abstract aspects that many distributed systems can reuse when implementing NFRs. They are generic because they do not reference to any domain specific concepts. These abstract aspects will be reused in our case study as well.

The abstract aspects presented here are "Encryption Aspect", "Timing Aspect", and "Logging Aspect". They address issues in common NFRs "Security", "Performance", and "Maintainability" respectively.

4.3.3.1 Encryption Aspect

The Encryption Aspect helps to implement the Security NFR by ensuring the confidentiality of the message sent through a network. The Encryption aspect defines two pointcuts: sendMsg and recvMsg. It encrypts every outgoing message and decrypts every

incoming message. Since both the sending and receiving ends' behaviours are modified by this aspect, Encryption aspect shall be always shared between the client and the server.

The encryption aspect defines two abstract pointcuts sendMsg and recvMsg, and add advice around both pointcuts. "proceed" is a keyword in AspectJ, it means "proceed to execute the original pointcut" [AspectJ02]. This is how to read the advice around sendMsg: encrypt the input parameter "msg" first, then proceed to execute the pointcut with the encrypted parameter. This is how to read the advice around recvMsg: proceed to execute the pointcut normally and get the return value, decrypt the return value and return it. Basically the first advice modifies (encrypts) the input and the second advice modifies (decrypts) the return value.

```
public abstract aspect BaseEncryption
{
  public abstract pointcut sendMsg(String msg);
  public abstract pointcut recvMsg();

  public void around(String msg): sendMsg(msg) {
      String encryptedMsg = encrypt (msg);
      proceed(encryptedMsg);
  }

  public String around(): recvMsg() {
      String result = proceed();
      String decryptedMsg = decrypt (result);
      return decryptedMsg;
  }

  public abstract String encrypt(String t);

  public abstract String decrypt (String t);

}
```

The following aspect "Encryption" is derived from "BaseEncryption". "Encryption"

aspect uses BlowFish encryption algorithm [Blowfish02] to encrypt and decrypt messages. Blowfish was designed as a fast, free alternative to existing encryption algorithms (e.g., DES).

```
public abstract aspect Encryption extends BaseEncryption
{

  // uses BlowFish encryptor algorithms
  BlowFishEncryptor encryptor;

  /**
   * encrypt a string by using BlowFish Encryptor
   * @return java.lang.String
   * @param m java.lang.String
   */
  public String encrypt(String t) {
       String m = encryptor.encryptBlock(t);
        return m;
  }

  /**
   * decrypt a string by using BlowFish Encryptor
   * @return java.lang.String
   * @param m java.lang.String
   */
  public String decrypt (String t) {
     String m = encryptor.decyptBlock(t);
      return m;
  }
}
```

To reuse this abstract aspect, a derived aspect can specify the concrete pointcuts for sendMsg and recvMsg. Alternative encryption algorithms can also be adopted by overriding the encrypt and decrypt methods.

4.3.3.2 Timing Aspect

The Timing Aspect helps to implement the performance NFR by measuring the duration

of messaging and providing enforcement points in its interface. The Timing aspect defines two pointcuts: sendMsg and recvMsg. It adds timestamp to every outgoing message and the timestamp is removed upon receiving the message. Since both the sending and receiving ends' behaviours are modified by this aspect, The Timing aspect shall be always shared between the client and the server.

The two aspects Timing and Encryption are defined on potentially the same set of pointcuts. In order to avoid the potential undefined order of execution, we specify that Timing aspect *dominates* Encryption aspect. It means that when both aspects specify advice around the same method, the advice from Timing aspect will be triggered first (but not necessarily completed first, especially if it calls *proceed*(), proceed() will make the rest of the pointcut complete first and then come back to execute the statements after proceed()).[3]

The advice around sendMsg adds time stamp into the input parameter "msg". The advice around recvMsg removes the added time stamp from the return value, and calls checkTimestamp method with the time stamp as the input parameter.

```
public abstract aspect Timing
       dominates BaseEncryption {

  public abstract pointcut sendMsg(String msg);
  public abstract pointcut recvMsg();

  public void around(String msg): sendMsg(msg) {
      String timeStampedMsg = timeStamping (msg);
      proceed(timeStampedMsg);
  }
```

[3] "dominates" and "proceed" are keywords from AspectJ

```
    public String around(): recvMsg() {
        String result = proceed();
        String msg = removeTimestamp (result);
        return msg;
    }

    /**
     * Time-Stamping a string
     * @return java.lang.String
     * @param t java.lang.String
     */
    public String timeStamping(String t) {
      /* actual implementation is omitted for clarity */
    }

    /**
     * remove time stamp from the string
     * validate the during of sending the message
     *
     * @return java.lang.String
     * @param t java.lang.String
     */
    public String removeTimestamp (String t) {
      /* actual implementation is omitted for clarity */
      String timeStamp = /* timestamp extracted from t*/
      String ret = /* t minus the timestamp */
      CheckTimestamp (timeStamp);
      Return ret;
    }

    /**
     * The derived aspect could override this method
     * and provide the actual enforcement for constraints on
     * timing to meet the application-specific performance NFRs
     *
     * @return java.lang.boolean
     * @param ts java.lang.String
     */
    public boolean checkTimestamp (String ts) {
     /* actual implementation is omitted for clarity */
    }
  }
```

To reuse this abstract aspect, a derived aspect can specify the concrete pointcuts for sendMsg and recvMsg. And then override the method checkTimeStamp() to perform the desired enforcement on the timing.

4.3.3.3 Logging Aspect

The following MethodTracing aspect helps to implement the maintainability NFR by providing trace for the execution of methods. i.e., the begin and end of the method invocations are logged. Log4J is used to provide the basic logging functionality [Log4J02].

The advice before the pointcut "callMethods" logs a message saying "Enter method", followed by the method's signature. The advice after the pointcut "callMethods" logs a message saying "Exit method", also followed by the method's signature.

```
public abstract aspect MethodTracing
{
    public abstract pointcut callMethods();

    //  use Logger from log4j
    private static Logger logger =
        Logger.getLogger(“MethodTracing”);

    // logging level
    private static int logLevel = 0;
    int getLogLevel() {
        return logLevel;
    }
    void setLogLevel(int level) {
        logLevel = level;
    }

    before (): callMethods()
    {
        if ( logLevel == 1 ) {
            logger.info("Enter method: " +
                        thisJoinPointStaticPart.
                        getSignature().
                        getName());
        } else if (logLevel == 0 ) {
            logger.info("Enter method: " +
                       thisJoinPointStaticPart.
                       getSignature());
        }
    }

    // trace the exit of a method invocation
    after (): callMethods()
```

```
    {
        if ( logLevel == 1 ) {
            logger.info("Exit method: " +
                        thisJoinPointStaticPart.
                        getSignature().
                        getName());
        } else if (logLevel == 0 ) {
            logger.info("Exit method: " +
                        thisJoinPointStaticPart.
                        getSignature());
        }
    }

}
```

The implementation of MethodTracing aspect combines the strengths of log4J and AspectJ. The logging level can be re-configured at run-time, through a configuration file. The logging configuration file is in XML. It specifies the destination of logging (e.g., a file name or console) and logging level for each logger. An example of the logging configuration file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd" >
<log4j:configuration>

        <!-- the standard appender -->
        <appender
           name="defaultFile"
           class="com.log4i.RollingFileAppenderWithPathCreate">
                <param name="File"
                    value="./logs/TestMessage.log" />
                <param name="MaxFileSize"
                    value="100000KB" />
                <param name="MaxBackupIndex"
                    value="3" />
                <layout
        class="com.log4i.ComprehensivePatternLayout">
                  <param
                   name="ConversionPattern"
                   value="%d\t%p\t%t\t%b\t%c\t%m%n"/>
                </layout>
        </appender>

        <!— capture all debug-level and higher for
            MethodTracing logger -->
        <category name=”MethodTracing"
                  class="com.log4i.Logger">
```

```
            <priority value="debug"/>
        </category>

        <!--  Root definitions -->
        <root>
            <priority value ="error"/>
            <appender-ref ref="defaultFile" />
        </root>

</log4j:configuration>
<!-- eof -->
```

To reuse this abstract aspect, a derived aspect can specify the concrete pointcut "callMethods". For example, "callMethods" can be defined as calls to a particular method, or a particular set of methods, or every method in a package, or every method in every class, etc.

4.3.4 Mapping PEOCL Policies To Aspects

The mapping from PEOCL policies to aspects is a relatively straightforward process. It is not totally a mechanical process, intelligent decisions have to be made during the mapping process (e.g., which of the 'before', 'after', or 'around' advice should be used). But there are some guidelines or informal rules that can be followed. The important parts of a PEOCL policy are "target", "introduction", "preCondition", "postCondition", and "invariant". Their mapping rules are outlined below.

- The *target* of a PEOCL design policy can be mapped to a *pointcut* of an aspect.
- The *introduction* of a PEOCL design policy can be mapped to an *introduction* of an aspect.
- The *preCondition, postCondition, and invariant* of a PEOCL design policy can be

mapped to *advice* on the pointcut as mapped from the design policy's target.

- The preCondition, postCondition and invariant are all OCL expressions. If the OCL expression filters on the targets as well then the filter in conjunction with the target of the PEOCL design policy can be mapped to a pointcut of an aspect. An example of OCL expression filters on the targets:

  (UML.MetaModel.Core.Method.name = ”setA” or

  UML.MetaModel.Core.Method.name = ”setB”)

  implies

  <OCL_expression_X>.

  This overall OCL expression specifies that the <OCL_expression_X> should be true if the method names are either setA or setB.

The following example demonstrates how to map policies expressed in PEOCL to Aspects. We will use the abstract aspect "Encryption" in the previous section to implement the PEOCL encryption policy presented in section 4.1.2. The PEOCL design policy for encryption is repeated below for reader's convenience:

```
<designPolicy name="Outgoing Message Encryption Policy">
  <category>Security</category>
  <target> DataOutputStream::writeUTF(msg : String) </target>
  <preCondition>
      <oclExpression> encrypted (msg)   </oclExpression>
  </preCondition>
</designPolicy>
```

The above PEOCL design policy can be mapped to an aspect "SocketMessageEncryption":

```
aspect SocketMessageEncryption extends Encryption {
```

```
        public pointcut sendMsg(String msg):
            call(void java.io.DataOutputStream.writeUTF
                    (String)) && args(msg) ;
}
```

In this example, all we need to do is to introduce a new aspect "SocketMessageEncryption", which inherits from the abstract aspect “Encryption”. SocketMessageEncryption aspect specifies the two pointcuts sendMsg and recvMsg to be the calls to two socket operations writeUTF and readUTF from java.io package. Every message through the socket interface will be encrypted before sending and decrypted after receiving.

The next chapter will present a case study that uses the methodology discussed in this chapter.

CHAPTER 5 CASE STUDY -- THE DEVELOPMENT OF A CHAT ROOM SYSTEM

This chapter presents a case study on using the policy-based methodology to create modularized design and implementation artifacts for NFRs. The methodology is illustrated through the development of an on-line chat room client-server system.

The chat room system was implemented first without the NFRs by using the traditional object-oriented method. NFRs were added gradually as the implementation went on.

The main artifacts from object-oriented method are presented first because they will be referenced when implementing NFRs. After the Non Functional Requirements are introduced, they will be expressed as policies at the design level, first in plain English, then in PEOCL. The PEOCL policies then are mapped to Aspects.

5.1 Design by Using Object-Oriented Method

5.1.1 User-oriented Requirements

This is a description of the initial requirements of a chat room system at the highest level, in an informal plain English form:

> A chat room provides a communication facility for multiple users connected through a network. Each user can type in a message and send the message to all

other users that are currently using the chat room. Each user also sees all the messages sent by any other users in the chat room.

5.1.2 Architectural Design Decisions

Architectural design decisions impacts NFRs. As stated before, we are not pursuing this research direction, as they have been addressed very well by [Chung00a, Chung00b, Gross00, Kazman99, Kazman00, Weiss01]. So these high level architectural decisions are presented below as a given from the user.

The following diagram (Figure 9) illustrates the network view of the overall system.

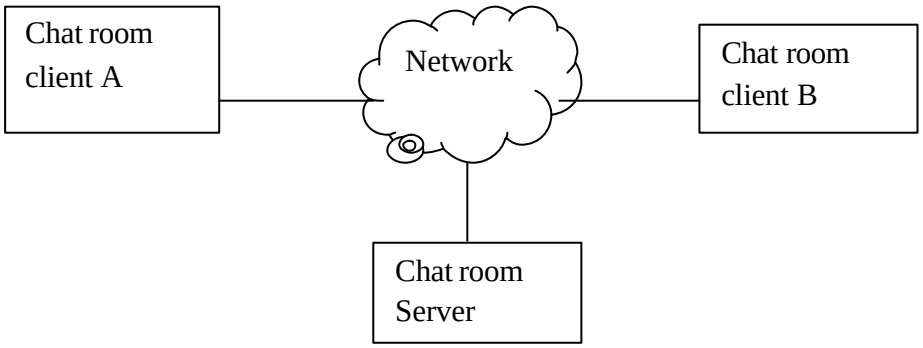

Figure 9 Network View of the Overall Chat Room System

It will be a client/server system. There is a client system on every end user's machine. There is a server system on the network that connects to and communicates with all the client systems.

- The client system sends messages to the server when the user enters a message. The

client system uses a separate thread to receive and display messages from the server.

- The server is a multithreaded application that accepts client connections and processes the received messages concurrently.

5.1.3 Main Use Case "Send a Message"

The following table outlines the main use case "Send a Message". It has 1 success scenario and some failure scenarios.

Use Case No.	001
Use Case Title	Send a message
Preconditions of the use case	The chat client system is running and connected to the chat server system successfully
Post conditions of the use case (success scenario)	The one line message has been sent to the receiving window of every client system that is currently connected to the chat server system
Use Case Starts when	The user types in a one-line message in the edit window and then hit the 'return' key
Normal Flow	The message is sent to the chat server The chat server broadcasts the message to every client that is currently connected to the chat server system
Alternate Flow #1: Fail to send message to server	The message can not be sent to the chat server The user is notified "failure in sending message to server
Alternate flow #2: Server fails to broadcast message	The message is sent to the chat server The chat server failed to broadcast the message to every client that is currently connected to the chat server system The chat server notifies the originator of the message The originating client system notifies the user "the server failed to broadcast the message"

Table 5 Use Case "Send a Message"

5.1.4 Overview of Classes

“Figure 10” and “Figure 11” present the classes in the Chat Room system.

The chat room server is implemented by two classes: ChatServer and ChatHandler (see “Figure 10”). ChatServer is the main program, it spawns a new thread for each client connection. The new thread runs class ChatHandler. A static member "handlers" maintains all the instances of ChatHanlders that are currently active. ChatHandler maintains the two way communication channels ("in" and "out") with the client through a "socket" connection. The main functional feature of ChatHandler is to "processMessage", i.e., to "broadcast" the message to all the other clients upon receiving a message.

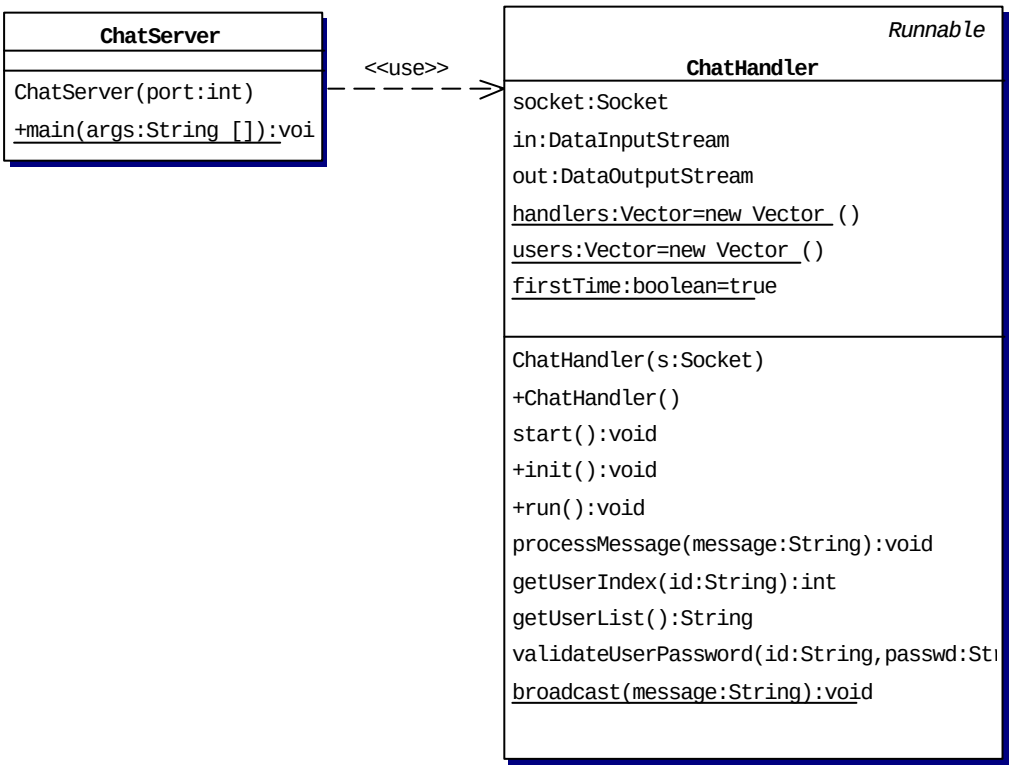

Figure 10 Class Diagram for Chat Room Server

The Chat room client is implemented by six classes (see "Figure 11"). ClientController is the main program, it establishes socket connection with the ChatServer, and then launches the LoginDialog. LoginDialog is responsible for authenticating the user and then launches the ChatrommClientWindow, which is the main window for the chat room client application. The ChatroomClientWindow can launch the other three dialogs: UserlistDialog, FriendListDialog, and PasswordDialog. The three dialog windows support the query and modification on users, friends, and password respectively.

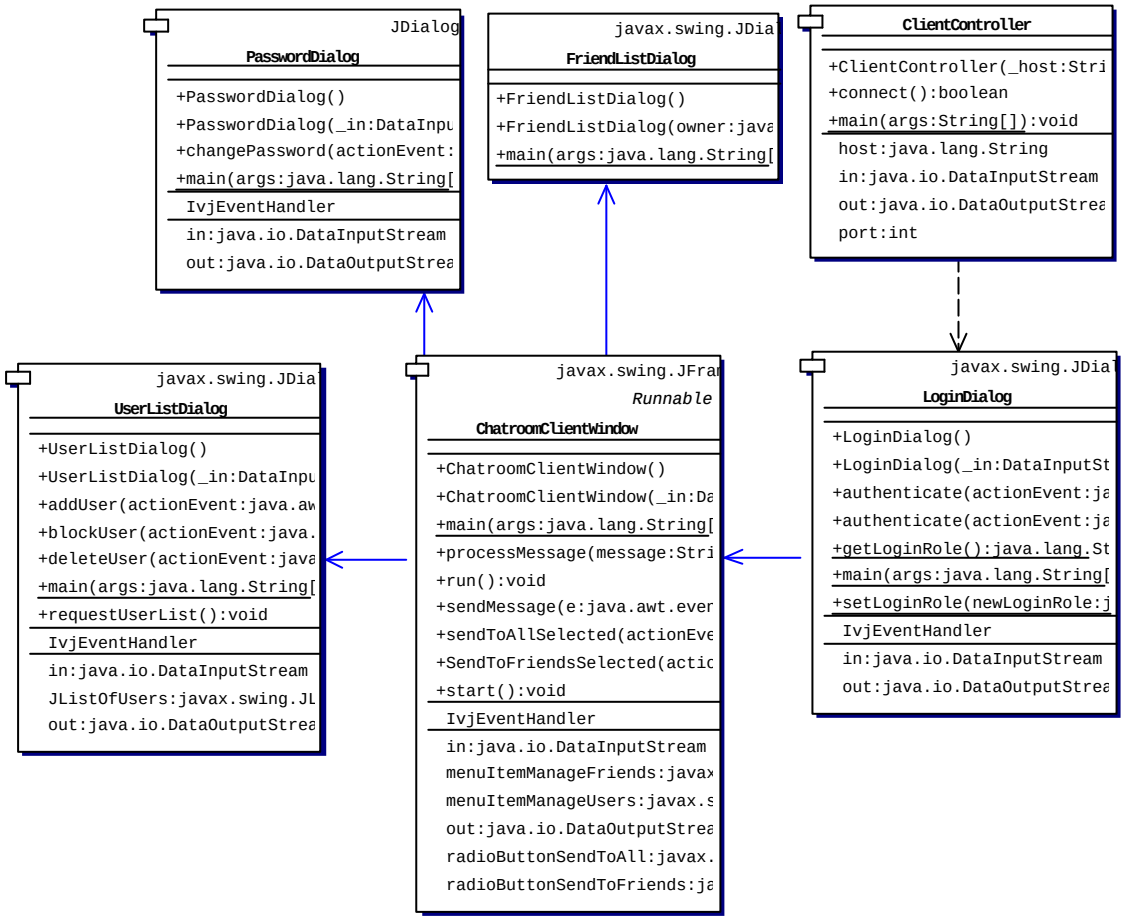

Figure 11 Class Diagram for Chat Room Client

5.1.5 Sequence Diagrams

This section presents some of the sequence diagrams to illustrate the design of the chat room system.

Figure 12 illustrates the sequence from making a connection to sending a message. Both activities are presented in the same diagram for easier correlation. When the user launches the chat room client application, the chat client makes a connect request to the chat server, the chat server then spawns a new thread chat handler to deal with the connection with this particular client. The message sending and GUI are in separate threads to avoid freezing the GUI activities. When the chat handler receives a message, it broadcasts it to every active chat client. The sending client will receive this broadcasted message as well, and display the message in its own GUI.

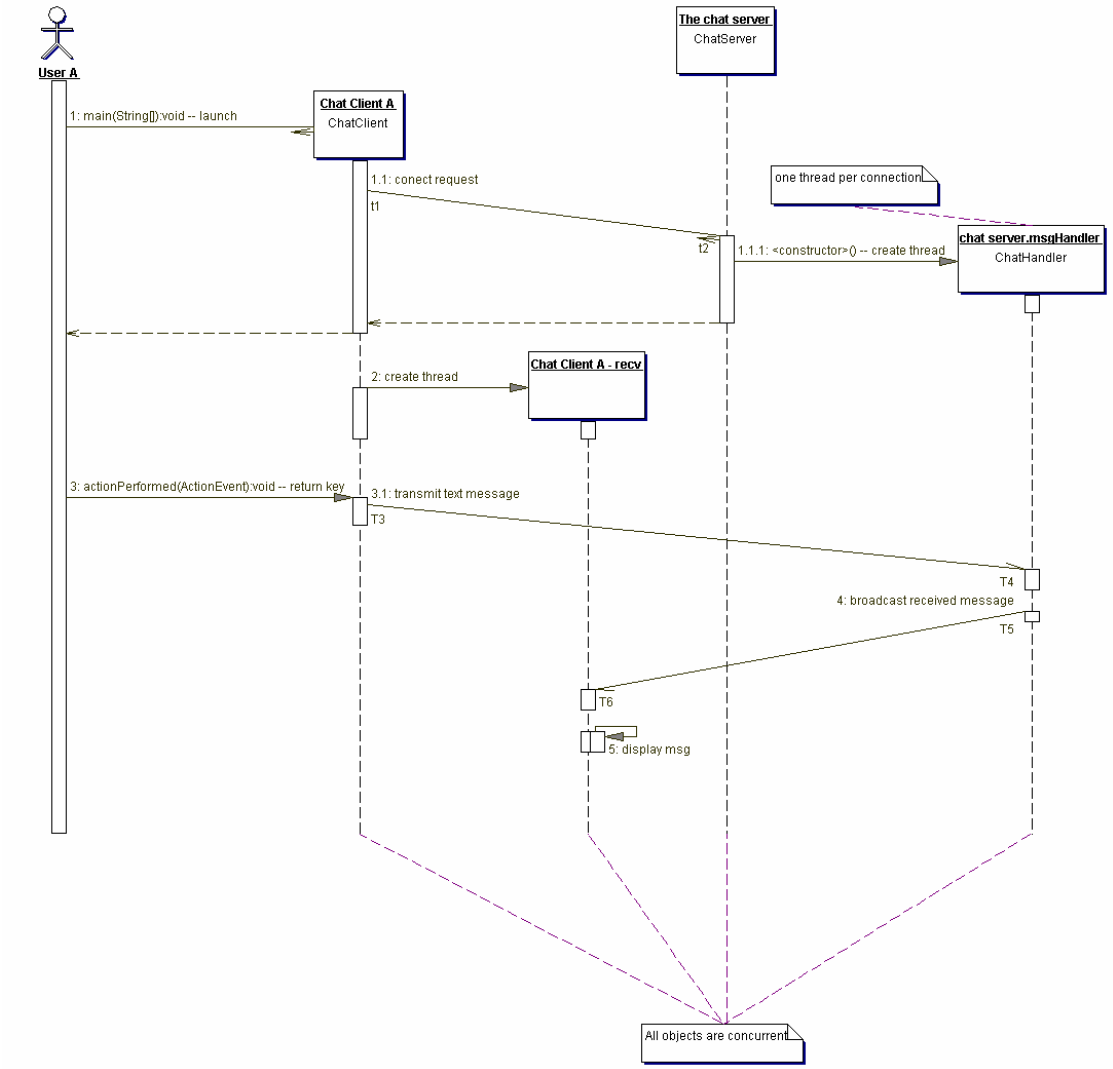

Figure 12 Sequence Diagram -- Send a message

"Figure 13" illustrates sequence diagram for "block out a user" activity. The interactions are among the administrative user, chat client X, and the ChatHandler on the server side. When the user clicks the "block out" button, the chat client sends the user name in the blockout request message to the chat handler, the chat handler then verifies and processes this message and sends back a response, finally the chat client displays GUI feedback to

the user.

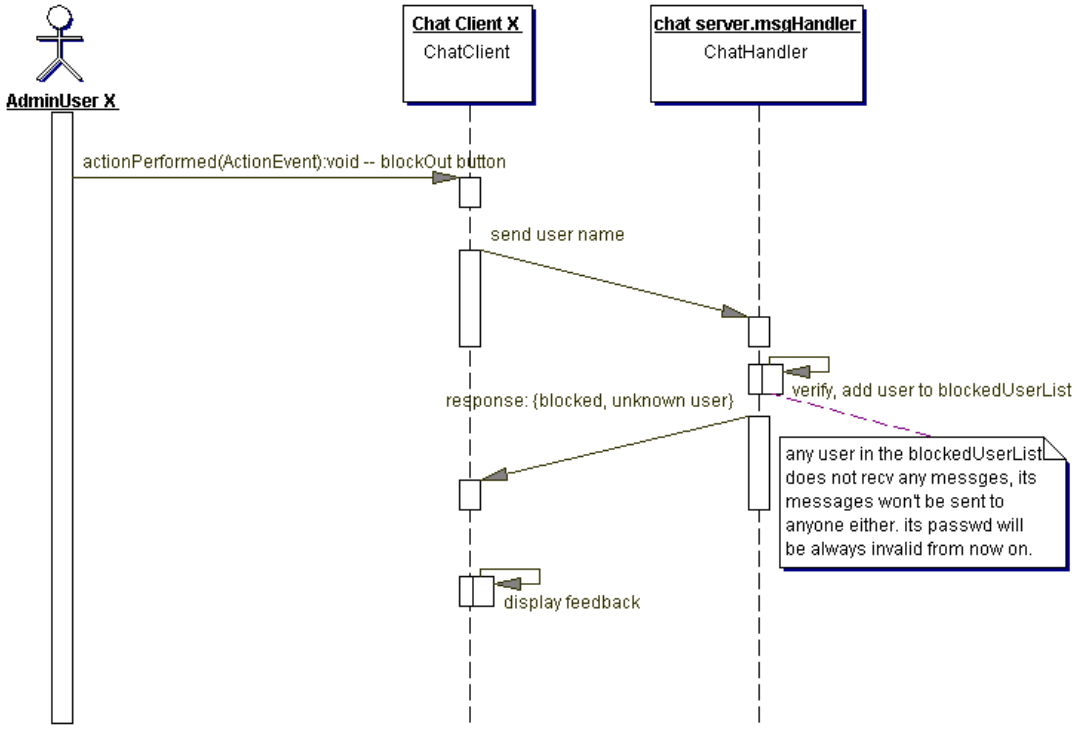

Figure 13 Sequence Diagram – Block Out a User

"Figure 14" illustrates sequence diagram for "change password" activity. The interactions are among a user, the chat client, and the ChatHandler on the server side. When the user clicks the "change password" button, the chat client will send to the chat handler the user name, old password and new password in the request message. The chat handler verifies and processes the request, and then sends back a response. Finally the GUI will display the feedback to the user.

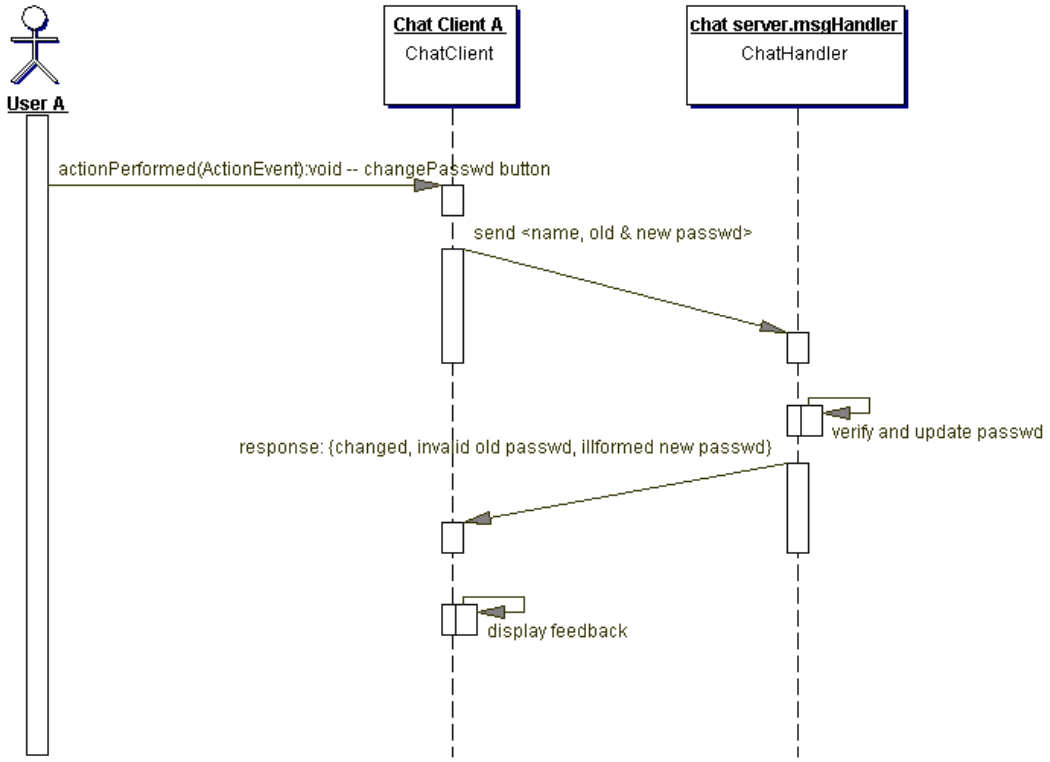

Figure 14 Sequence Diagram – Change Password

"Figure 15" illustrates sequence diagram for "login" activity. The interactions are among a user, the chat client, and the ChatHandler on the server side. When the user clicks the "login" button, the chat client will send to the chat handler the user name and password in the request message. The chat handler verifies and processes the request, and then sends back a response. Finally the GUI will display the feedback to the user.

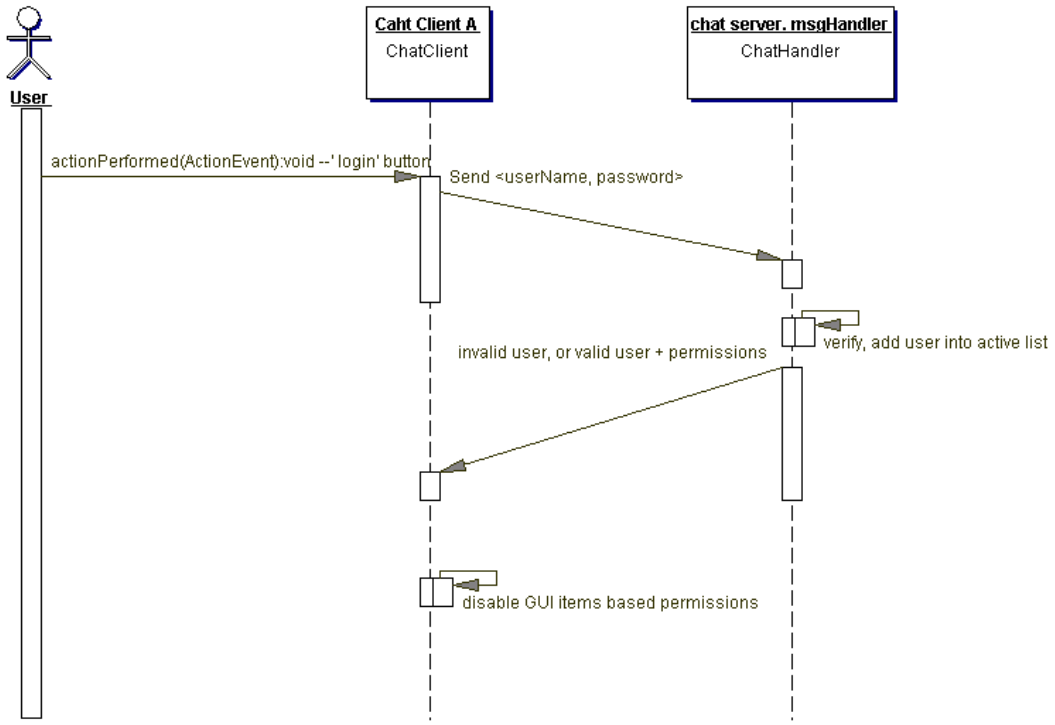

Figure 15 Sequence Diagram -- Login

5.2 Chat Room Client Application Graphical User Interface

This section presents some of the GUI windows of the client application in the final chat room system (with all NFRs added), to help the reader understand the overall requirements better.

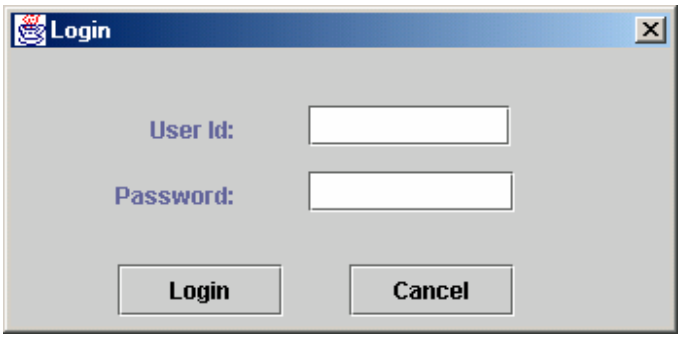

Figure 16 Authentication Window

"Figure 16 Authentication Window" is the very first window the user sees when launching the chat room client. Once the user name and password are authenticated, the authentication window disappears and the next window is shown in "Figure 17 Chat room client application main window".

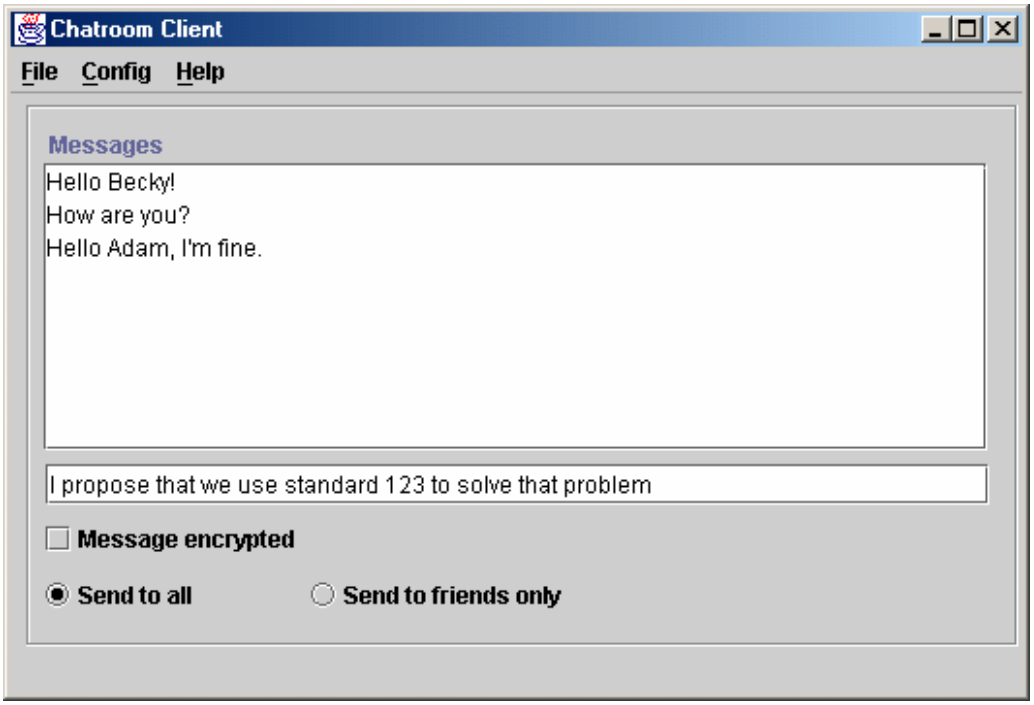

Figure 17 Chat room client application main window

The chat room client main window consists of (from top to bottom in Figure 17) a menu bar, an incoming message display area, an outgoing message line, and an option pane. The "Config" menu has three sub menu items "Manage Users", "Manage Friends", and "Change Password", as shown in 'Figure 18 Sub menu items for 'Config'". The incoming message display area displays all messages from all users of this chat room, including this user's own message. The outgoing message line is where the user can type in its own message. The message will be sent when return key is hit.

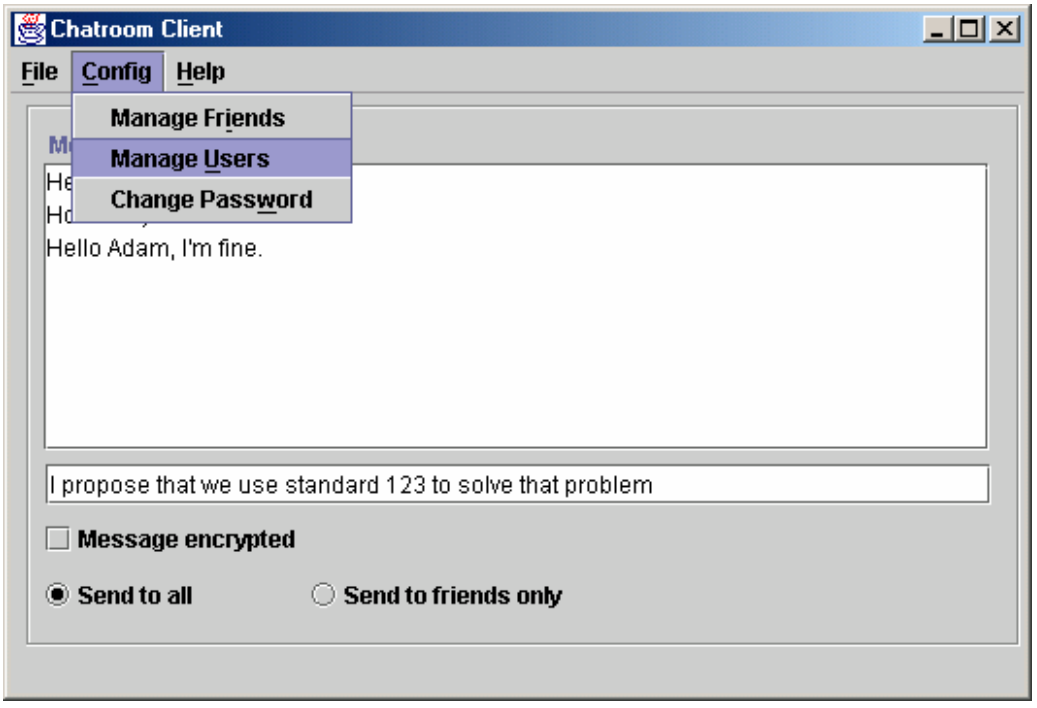

Figure 18 Sub menu items for 'Config'

The option pane has a check box that specifies whether the message should be encrypted or not. This check box is only available for gold and administrative users. The option pane also has two radio buttons that specify whether the message should be sent to

everyone in the chat room or just to the friends in the friend list. This option is not available to bronze users.

"Manage User" menu item will trigger "Figure 19 User List Management Window", where a list of <user, password, role, email> is displayed. New users can be added into the list. Existing users can be modified, or deleted from the list, or blocked out. A user can not login any more (authentication always fails) if it is blocked out. The "Manage User" menu item is disabled for all non-administrative users.

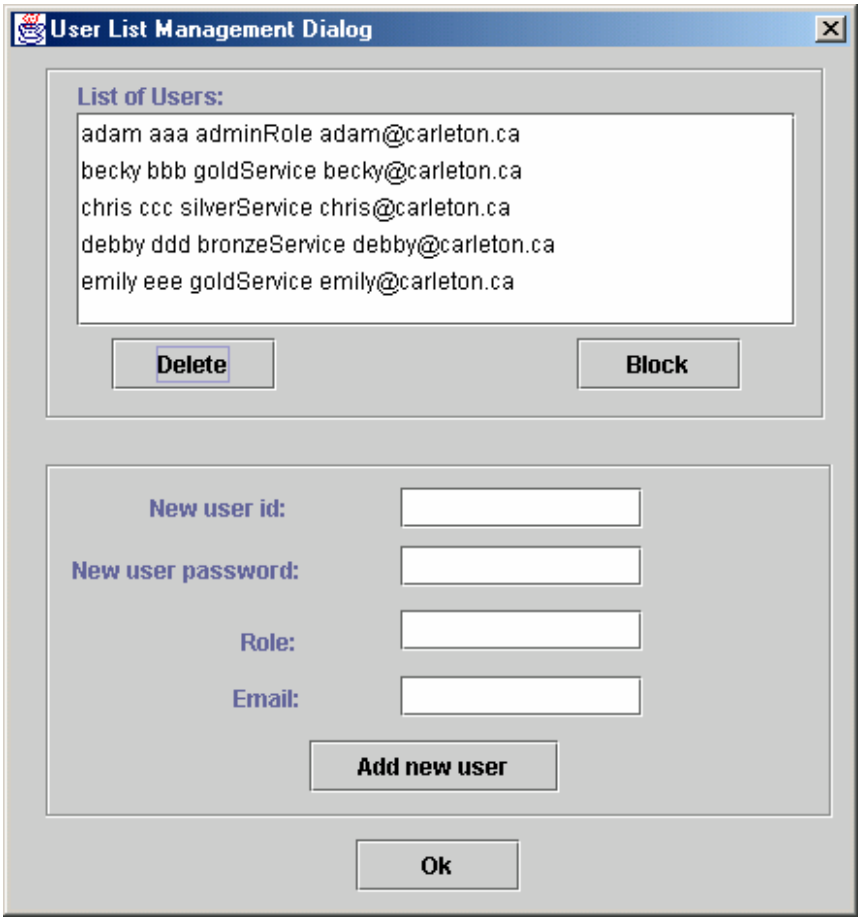

Figure 19 User List Management Window

"Change Password" menu item will trigger "Change Password Window", where this user's password can be updated.

"Manage Friends" menu item will trigger "Friend List Management Window", where a list of friends is displayed. Friends can be added or deleted. "Manage Friends" menu item is disabled for bronze users.

5.3 Adding Non Functional Requirements

This is the list of NFRs that we need to implement:

NFR #1, Security NFR:

User shall be limited to use features as permitted by his or her role

The message must be sent onto the network in a secure format

Only registered users can enter the chat room

Administrative user can block out a 'bad' user

NFR #2, Performance NFR:

The messages must be received in a reasonable amount of time, e.g. within 2 seconds

NFR #3, Accounting NFR:

The user shall be charged 1 cent per minute

NFR #4, Maintainability NFR:

All method calls shall be logged

The next section will describe how the NFRs are mapped to policies and then to aspects.

5.4 Mapping from NFRs to PEOCL Policies to Aspects

This section presents the overview picture of how NFRs in section 5.3 are further refined into detailed NFRs, and then expressed as design-level policies (i.e., PEOCL design policies), and finally implemented as code-level policies (i.e., aspects) or traditional Java code. This section only provides a high-level view. Section 5.5 will present the design policies formalized in PEOCL form. Section 5.6 will present the detailed code-level policies (i.e., the actual aspect code).

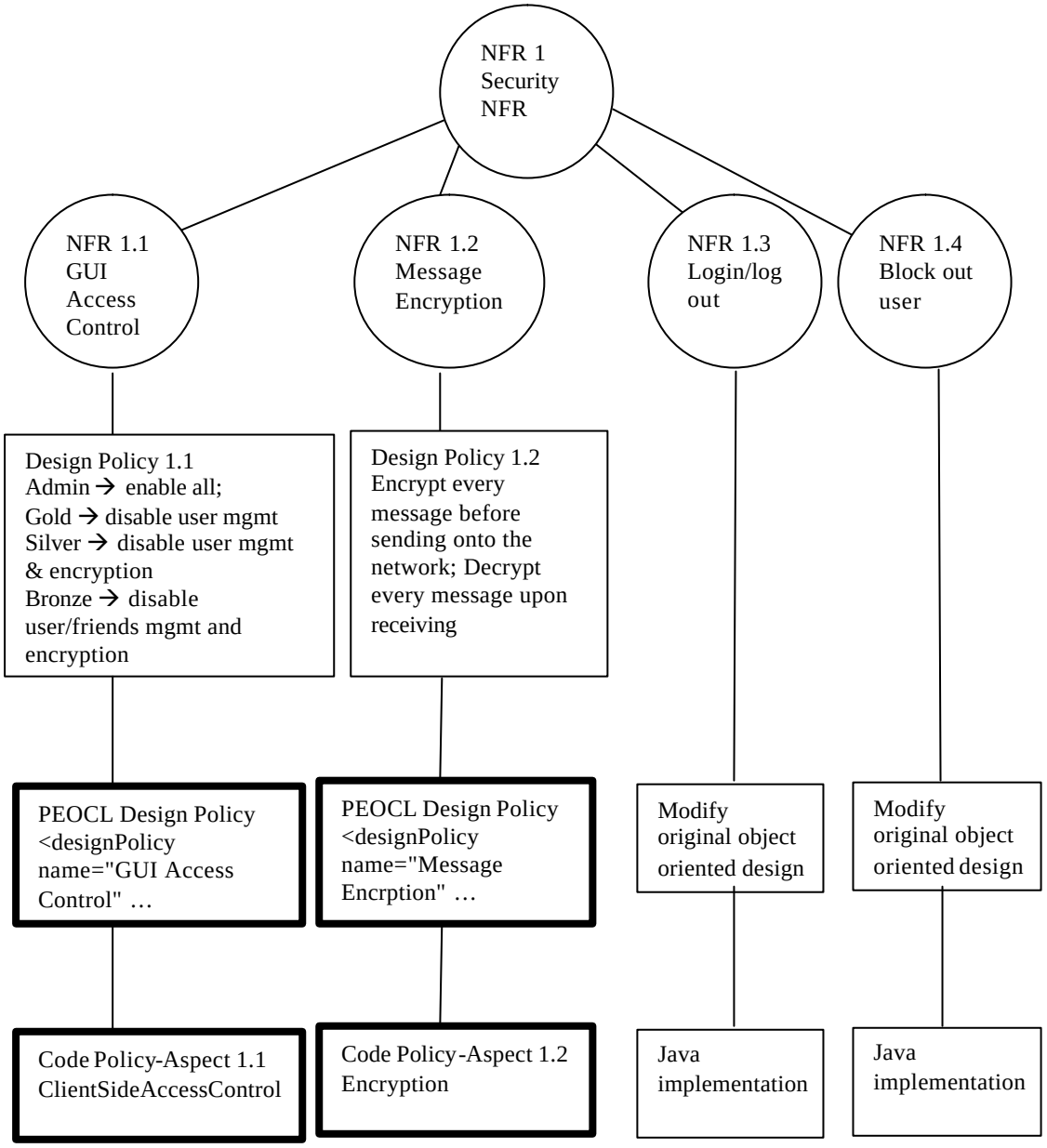

Figure 20 Design for Security NFR

"Figure 20 Design for Security NFR" illustrate how security NFR is refined into four NFRs, and then two of them are mapped into design policies (see sections 5.5.1 and 5.5.2

for details), and then mapped to aspects (see section 5.6.1 and 5.6.2 for details). The other two are mapped into traditional Java code and will not be further discussed in this thesis.

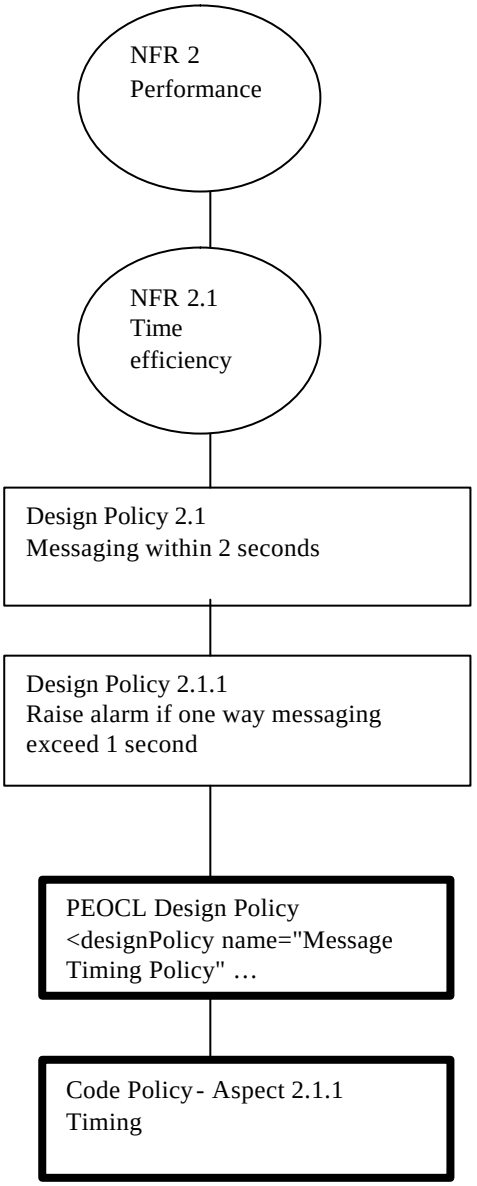

Figure 21 Design for Performance NFR

"Figure 21 Design for Performance NFR" illustrates how the performance NFR is refined into time efficiency and space efficiency, only time efficiency is relevant in our case

study, so it is further mapped into design policies (see section 5.5.3 for details) and a aspect (see section 5.6.3 for details).

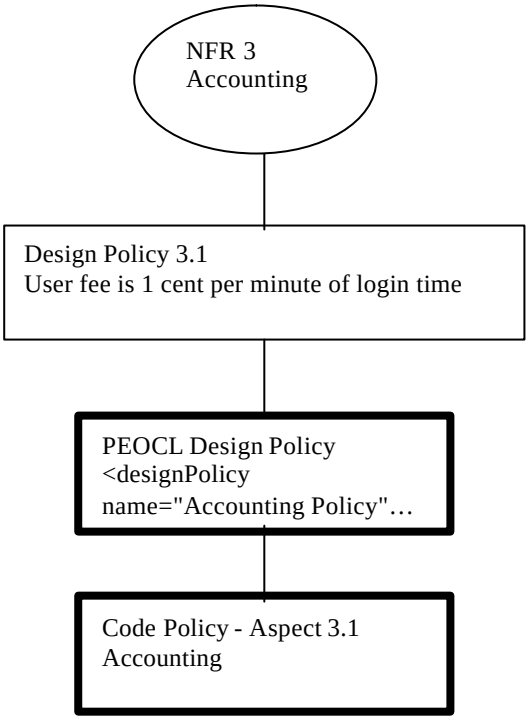

Figure 22 Design for Accounting NFR

"Figure 22 Design for Accounting NFR" illustrates how the accounting NFR is mapped to a design policy (see section 5.5.4 for details) and then to an aspect (see section 5.6.4 for details).

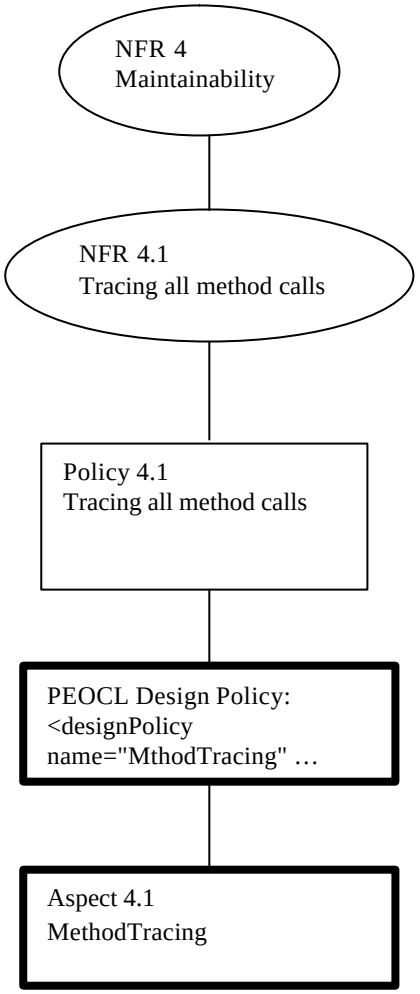

Figure 23 Design for Logging NFR

The design level policies in the above diagrams have been formalized by using PEOCL, and aspects have been all implemented in AspectJ. The next sections will present further details.

5.5 Capturing NFR-Related Policies by using PEOCL

The following sections present how NFRs are refined and then mapped to design policies, how the design policies are captured formally in PEOCL. All PEOCL expressions are based on the UML class diagrams presented in 5.1.4.

5.5.1 Access Control Policy for Security NFR

One of the Security NFRs is the access control NFR. We first refine it into finer granularity policies, then formalize them by expressing them in PEOCL, which refer to the previous UML classes diagrams.

GUI Access Control NFR

Each user shall be only allowed to access portions of GUI that he or she has permission to access.

Refining GUI Access Control NFR – GUI Access Control Policy in natural language

P-a. If user's privilege-level is 'admin', then the user can access all GUI windows

P-b. If user's privilege-level is 'Gold', then the user can access all GUI windows and items except user management window

P-c. If user's privilege-level is 'Silver', then the user can access all GUI windows and items except user management window and message encryption option.

P-d. If user's privilege-level is 'Bronze', then the user can access all GUI windows and items except these items: user management window, friend management window,

message encryption check box, friend-only option when sending messages.

Formalizing GUI Access Control Policy -- GUI Access Control Policy in PEOCL

The GUI Access Control Policy is a composite policy, it consists of two sub design policies: "Introducing loginRole", and "GUI Access Control Core Policy". It is necessary to introduce a new attribute to the LoginDialog class, because LoginDialog does not concern about the concept of "role" before having this requirement of GUI Access Control.

```
<designPolicy name="GUI Access Control Policy">
  <category>Security</category>
  <designPolicy>Introducing loginRole</designPolicy>
  <designPolicy>GUI Access Control Core Policy</designPolicy>
</designPolicy>


<designPolicy name="Introducing loginRole">
  <category>Security</category>
  <target>LoginDialog</target>
  <introduction> String loginRole; </introduction>
</designPolicy>

<designPolicy name="GUI Access Control Core Policy">
  <category>Security</category>
  <target>ChatroomClientWindow</target>
  <invariant>
     <oclExpression>

        self.loginDialog.loginRole = "admin" implies
        ( self.menuItemManageFriends.enabled    = true and
          self.menuItemManageUsers.enabled      = true and
          self.radioButtonSendToAll.enabled     = true and
          self.radioButtonSendToFriends.enabled = true and
          self.checkBoxEncryption.enabled       = true )
```

```
        self.loginDialog.loginRole = "gold"     implies
        ( self.menuItemManageFriends.enabled    = true and
          self.menuItemManageUsers.enabled      = false and
          self.radioButtonSendToAll.enabled     = true and
          self.radioButtonSendToFriends.enabled = true and
          self.checkBoxEncryption.enabled       = true )

        self.loginDialog.loginRole = "silver"   implies
        ( self.menuItemManageFriends.enabled    = true and
          self.menuItemManageUsers.enabled      = false and
          self.radioButtonSendToAll.enabled     = true and
          self.radioButtonSendToFriends.enabled = true and
          self.checkBoxEncryption.enabled       = false )

        self.loginDialog.loginRole = "bronze"   implies
        ( self.menuItemManageFriends.enabled    = false and
          self.menuItemManageUsers.enabled      = false and
          self.radioButtonSendToAll.enabled     = false and
          self.radioButtonSendToFriends.enabled = false and
          self.checkBoxEncryption.enabled       = false )
    </oclExpression>
  </invariant>
</designPolicy>
```

Note: When both the menu items "Send to all" and "Send to friends" are disabled, the user does not have the GUI selections any mo re, the default behavior is "always send to all".

5.5.2 Message Encryption Policy for Security NFR

Another NFR for security is encryption. Encryption can be further divided into

“encrypted when stored” and “encrypted when transmitted”. The PEOCL representation

for “encrypted when transmitted” is presented below.

Message Encryption NFR

The message shall be encrypted when transferred over the network.

Message Encryption Policy in PEOCL

```
<designPolicy name="Message Encryption Policy" >
 <category>Security</category>
 <designPolicy name="Outgoing Message Encryption Policy"/>
 <designPolicy name="Incoming Message Encryption Policy"/>
</designPolicy>

<designPolicy name="Outgoing Message Encryption Policy">
 <category>Security</category>
 <target> DataOutputStream::writeUTF(msg : String) </target>
 <preCondition>
     <oclExpression> encrypted (msg)   </oclExpression>
 </preCondition>
</designPolicy>


<designPolicy name="Incoming Message Encryption Policy">
 <category>Security</category>
 <target> DataInputStream::readUTF() : return msg : String </target>
 <postCondition>
     <oclExpression> encrypted (msg)   </oclExpression>
 </postCondition>
</designPolicy>
```

Note: the word "encrypted" is in the ontology introduced by the NFR taxonomy.

5.5.3 Timing Policy for Performance NFR

Performance can be space-efficiency or time-efficiency. The following timing policy addresses the time-efficiency aspect.

Design Policy for Time-efficiency Performance NFR

One-way message must arrive within 1 second, i.e., the message sent from client to

server, or from server to client, must arrive within 1 second.

Timing Design Policy In PEOCL

```
<designPolicy name="Message Timing Policy">
 <category>Performance</category>
 <target> DataOutputStream::readUTF ( msg : String ) </target>
 <preCondition>
    <oclExpression>
       timeStamped (msg) and
       (now - timeInMessage(msg)  lessThan  1000 milliseconds)
    </oclExpression>
 </preCondition>
</designPolicy>
```

Note: the terms "timeStamped", "now", and "timeInMessage" are from ontology of NFR taxonomy.

5.5.4 Accounting Policy for Accounting NFR

Accounting NFR is typically considered later in the development stage (unless it is accounting software). And it is considered an overhead or burden, and that is why it is considered a part of the NFRs, even though they are not necessarily a quality attribute as defined in [Babacci95], but frequently it impacts many parts of the system in a scattered fashion which makes the addition of the accounting NFR very difficult. We will show the ease and modularity of implementing accounting NFR through PEOCL and Aspect.

Design Policy for Accounting NFR

User fee is 1 cent per minute for the total duration from login to logout.

Accounting Policy in PEOCL

```
<designPolicy name="Accounting Policy">
 <category>Accounting</category>
  <designPolicy name="Record Login Time Policy"/>
 <designPolicy name="Fee Calculation Policy"/>
</designPolicy>


<designPolicy name="Fee Calculation Policy">
 <category>Accounting</category>
 <target>
   ChatHandler.run()
 </target>
 <introduction>
     loginTime Date;
     accounts Vector;
     userId String;
 </introduction>
 <postCondition>
     <oclExpression>
        self.accounts[self.userId] = (now - self.loginTime)/60*1
     </oclExpression>
 </postCondition>
</designPolicy>

<designPolicy name="Record Login Time Policy">
 <category>Accounting</category>
 <target>
   ChatHandler.validateUserPassword (user : String, password : String)
 </target>
 <postCondition>
     <oclExpression>
        self.loginTime = now and  self.userId = user
     </oclExpression>
 </postCondition>
</designPolicy>
```

5.5.5 Logging Policy for Maintainability NFR

Logging is the another NFR that is typically lacking in many systems, because they are not customer-facing features. It is not easy to enforce a system-wide logging policy. The usually approach is to ask every developer to go through every module and manually add logging statements. This is costly, difficult to change, and hard to ensure consistency and completeness. We will show how it can be done in a modularized way so that it is easy to do, easy to change, and easy to ensure consistency and completeness.

Logging Policy Tracing all method calls.

```
<designPolicy name="Trace All Method Calls Policy">
 <category>Maintainability</category>
 <target>
  UML.MetaModel.Core.Method::invoke()
 </target>
 <postCondition>
    <oclExpression>
      (log - log@pre) -> notEmpty
    </oclExpression>
 </postCondition>
</designPolicy>
```

5.6 Implementing NFR Policies By Using AspectJ

This section presents the implementation of the PEOCL policies for NFRs. The critical parts of the code are presented[4]. The examples demonstrate how the "policy-based programming" thinking helps the software development process. You will see the de-

[4] AspectJ release 1.0 rc2 and JDK1.3.1_01 have been used to compile and execute all

coupling between the normal control flow and policy checking and enforcement, and the centralization of otherwise scattered code.

5.6.1 Implementation for Access Control Policy

The following code is the access control aspect written in AspectJ. This aspect adds an advice on the start of the ChatroomClientWindow to decide the permission level based on the user's role (or service level). For example, bronze users can just send and receive message, they do not have access to features like encrypting outgoing message or sending messages to friends only.

```
aspect ClientSideAccessControl
{
  pointcut startMainWindow(client.ChatroomClientWindow win):
       call(void client.ChatroomClientWindow.start()) && target (win);

  after(client.ChatroomClientWindow win): startMainWindow(win) {
       clientSideAccessControl( win );
  }

  // client-side Role-Based Access Control (RBAC)
  void clientSideAccessControl( client.ChatroomClientWindow w) {
      if ( client.LoginDialog.getLoginRole().indexOf("admin") >=0 ) {
            w.getMenuItemManageFriends().setEnabled(true);
            w.getMenuItemManageUsers().setEnabled(true);
            w.getRadioButtonSendToAll().setEnabled(true);
            w.getRadioButtonSendToFriends().setEnabled(true);
            w.getCheckBoxEncryption().setEnabled(true);
      } else if (client.LoginDialog.getLoginRole().indexOf("gold") >= 0
) {
            w.getMenuItemManageFriends().setEnabled(true);
            w.getMenuItemManageUsers().setEnabled(false);
            w.getRadioButtonSendToAll().setEnabled(true);
            w.getRadioButtonSendToFriends().setEnabled(true);
            w.getCheckBoxEncryption().setEnabled(true);
      } else if (client.LoginDialog.getLoginRole().indexOf("silver")
>=0 ) {
            w.getMenuItemManageFriends().setEnabled(true);
            w.getMenuItemManageUsers().setEnabled(false);
```

AspectJ code

```
            w.getRadioButtonSendToAll().setEnabled(true);
            w.getRadioButtonSendToFriends().setEnabled(true);
            w.getCheckBoxEncryption().setEnabled(false);
      } else if (client.LoginDialog.getLoginRole().indexOf("bronze")
>=0 ) {
            w.getMenuItemManageFriends().setEnabled(false);
            w.getMenuItemManageUsers().setEnabled(false);
            w.getRadioButtonSendToAll().setEnabled(false);
            w.getRadioButtonSendToFriends().setEnabled(false);
            w.getCheckBoxEncryption().setEnabled(false);
      }
  }
}
```

5.6.2 Implementation for Encryption Policy

The implementation of encryption policy is done by SocketMessageEncryption aspect written in AspectJ.

```
aspect SocketMessageEncryption extends Encryption {

  public pointcut sendMsg(String msg):
                  call(void java.io.DataOutputStream.writeUTF(String))
                  && args(msg) ;
  public pointcut recvMsg():
                  call(String java.io.DataInputStream.readUTF()) ;
}
```

The SocketMessageEncryption aspect reuses the abstract aspect “Encryption”. SocketMessageEncryption aspect specifies the two pointcuts sendMsg and recvMsg to be the calls to two socket operations writeUTF and readUTF from java.io package. Every message through the socket interface will be encrypted before sending and decrypted after receiving, the encryption algorithm is BlowFish.

5.6.3 Implementation for Timing Policy

The implementation of timing policy is done by the SocketMessageTiming aspect written in AspectJ.

```
aspect SocketMessageTiming extends Timing {

  public pointcut sendMsg(String msg):
           call(void java.io.DataOutputStream.writeUTF(String))
           && args(msg) ;
  public pointcut recvMsg():
           call(String java.io.DataInputStream.readUTF()) ;

  public boolean checkTimestamp (String ts) {
      Date currentTime = new Date();
      System.out.println ( "current time = " + currentTime );
      long d = currentTime.getTime() - timeSent.getTime();
      System.out.println ( "duration = " + d);

      if ( d > 2 ) {
            System.out.println ("warning: it took more than 2 seconds
to receive the message");
            return false;
      }

      return true;
  }
}
```

The SocketMessageTiming aspect reuses the abstract aspect “Timing”. The SocketMessageTiming aspect specifies the two pointcuts sendMsg and recvMsg to be the calls to two socket operations writeUTF and readUTF from java.io package. And then the SocketMessageTiming aspect overrides the method checkTimestamp() to check and report a warning message if the duration is too long.

5.6.4 Implementation for Accounting Policy

This is the Accounting aspect written in AspectJ. Some of the details are explained after the code.

```
aspect  Accounting
{
  // introductions
  static Vector ChatHandler.accounts = new Vector ();
  String ChatHandler.userId = "unknown";
  Date ChatHandler.loginTime;

  // The following pointcut and advice performs
  // “accounts-initialization”
  pointcut initChatHandler(ChatHandler h):
      call(void ChatHandler.init()) && target (h) ;

  void around(ChatHandler h): initChatHanlder(h) {
       proceed(h);
       /* init accounts with <id,duration> */
  }

  // The following pointcut and advice calculates
  // the login duration upon the
  // disconnection of the session”
  pointcut chatHandler_run_exception(ChatHandler h):
      within(ChatHandler) &&
      ( withincode (void ChatHandler.run()) && target (h) )
        &&  handler(IOException) ;

  after(ChatHandler h): chatHandler_run_exception(h) {
      long duration = (new Date()).getTime() - h.loginTime.getTime();
      h.bill(h.userId, duration);
  }
  // The following pointcut and advice sets the user name and
  // the start time of a session
  pointcut chatHandler_validateUserPassword (ChatHandler h,
                        String id, String passwd):
         call (boolean ChatHandler.validateUserPassword(
               String, String)) &&
         target(h) &&
         args (id, passwd);

  before (ChatHandler h, String id, String passwd) :
                     chatHandler_validateUserPassword(h, id, passwd) {
                            h.userId = id;
                     }

  after (ChatHandler h, String id, String passwd) :
                     chatHandler_validateUserPassword(h, id, passwd) {
                            h.loginTime = new Date();
                     }

  // The Accounting algorithm is based on duration of the usage,
```

```
  // the detail is omitted because it is not directly relevant
  void ChatHandler.bill (String id, long duration) {
       /* accounts[userId] is incremented by the fee of this session */
       /*   which is 1 cent per minute for the elapsed time
            since loginTime */
       accounts[userId] += (now - loginTime)/60*1;
  }
}
```

This aspect adds these attributes into the class ChatHandler: accounts, userId, and logingTime. Sometimes it is the natural thing to expand the existing classes to support a NFR. AspectJ provides a language construct called “introduction” that allows us to add extra members into an existing class without actually modifying the class. This helps to improve modularity by allowing clustering of functionality along different dimensions. It also helps the non-invasive adaptation of existing modules.

The advice on the pointcut initChatHanlder performs additional initialization for the newly introduced data members.

The advice on the pointcut chatHandler_run_exception calculates the duration of this session.

The advice on the pointcut chatHandler_validateUserPassword remembers the user id and the start time of the current session.

5.6.5 Implementation for Logging Policy

The implementation of the logging policy is through reusing the generic abstract aspect MethodTracing. All we need to do is to define a concrete pointcut “callMethods” which specifies all method calls shall be traced. This implementation is extremely simple, only three lines of code. This simplicity is helped by the power of aspect and log4j. The code for the aspect TraceAllMethods is listed below.

```
aspect TraceAllMethods extends MethodTracing {
    // declare the pointcut of interest, i.e., all method calls
    public pointcut callMethods() : execution (* *.*(..));
}
```

5.6.6 Evolution of Communication Protocol

The chat room system uses an XML-based text messaging format for the client and server to exchange PDUs, an example of the PDU format is presented below.

Example PDU for “authentication request” message:

```
<PDU TYPE=AUTHENTICATION>
        <USER>
                <NAME> adam </NAME>
                <ID> aaa </ID>
                <PASSWD> xyz </PASSWD>
                <EMAIL> adam@carleton.ca </EMAIL>
        </USER>
</PDU>
```

Some of our NFRs require the change of the communication protocol. The change to communication protocol is usually deemed to be a huge architectural change in a distributed system. It requires changes in both client and server. But with the help of

aspects, those changes have been made extremely simple and modular, without touching existing Java code. Our implementation supported the new PDU formats by intercepting every incoming and outgoing message on both client and server side (reference 5.6.2 Implementation for Encryption Policy and 5.6.3 Implementation for Timing Policy).

This is the new PDU format after adding the timing aspect:

```
<PDU TYPE=AUTHENTICATION  TIME=2001-10-08-10:05:02>
        <USER>
                <NAME> adam </NAME>
                <ID> aaa </ID>
                <PASSWD> xyz </PASSWD>
                <EMAIL> adam@carleton.ca </EMAIL>
        </USER>
</PDU>
```

5.7 Evaluation Of The Approach

One of the most important principles in software engineering is the *separation of concerns* principle [Dijkstra76]. This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity and to achieve the required engineering quality factors such as adaptability, maintainability, extendibility and reusability.

What we demonstrated in the previous sections are the separation of concerns on NFR's design and implementation from Functional Requirements' design and implementation. Based on the *separation of concerns* principle, we argue that our proposed methodology helps to improve the maintainability, adaptability, extendibility and reusability of a

software system. The next section will further support this argument by comparing the artifacts from the tradition approach and from the proposed approach.

5.7.1 Comparing The Traditional Approach And The Proposed Approach

By using the proposed methodology, we have observed the clean separation of concerns in many ways in the previous sections. First the design and implementation artifacts for NFRs are separated from those for FR. Second the design and implementation artifacts for each NFR is in a separate module. All the benefits of the separation of concerns are realized through the proposed methodology.

As a validation to the Separation Of Concerns principle, we now compare the artifacts from our proposed method (sections 5.5 and 5.6) against the artifacts from the traditional object-oriented method (see below).

We used the traditional object-oriented methodology and developed some of the design artifacts for the same NFRs. In particular, the sequence diagrams for three NFRs (logging, timing, and encryption) are presented below.

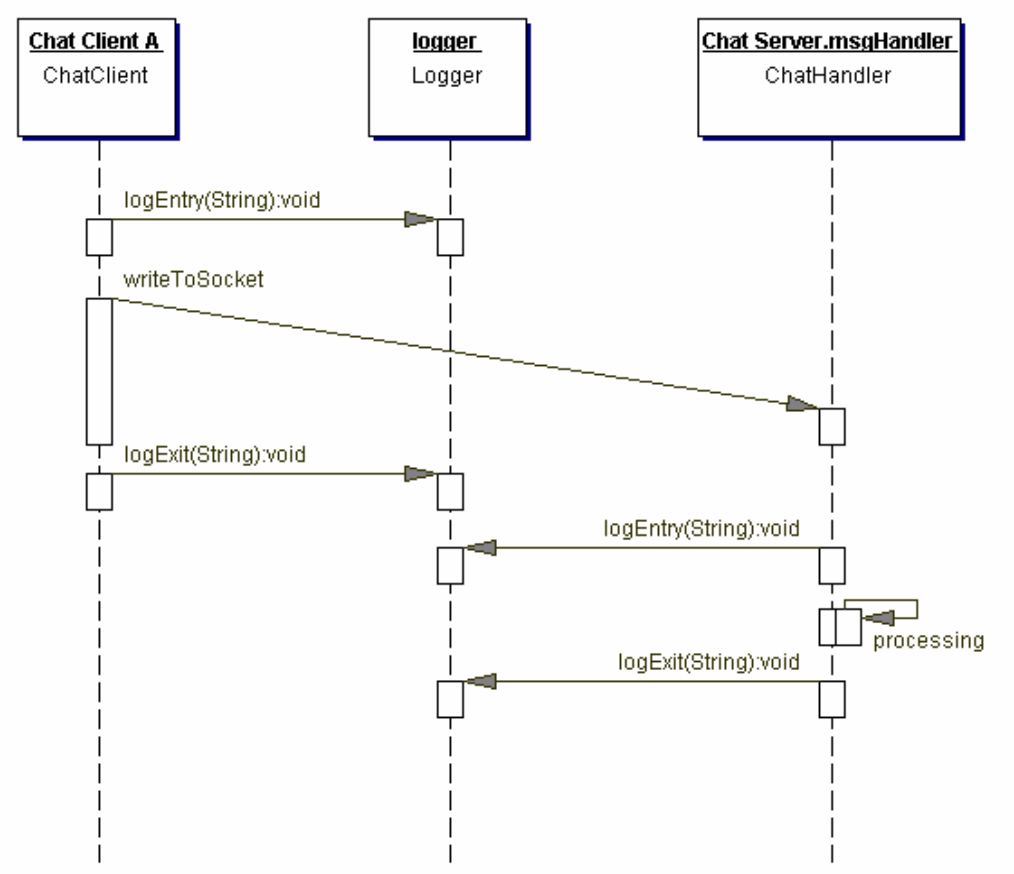

Figure 24 Sequence Diagram after adding logging NFR

"Figure 24 Sequence Diagram after adding logging NFR" specifies the need to log the entrance and exit of methods. Notice that the traditional object-oriented method requires that all the sequence diagrams (in section 5.1.5) need to be updated to support this simple NFR. This design (and thus code) impacts the original design (and code) in a scattered fashion.

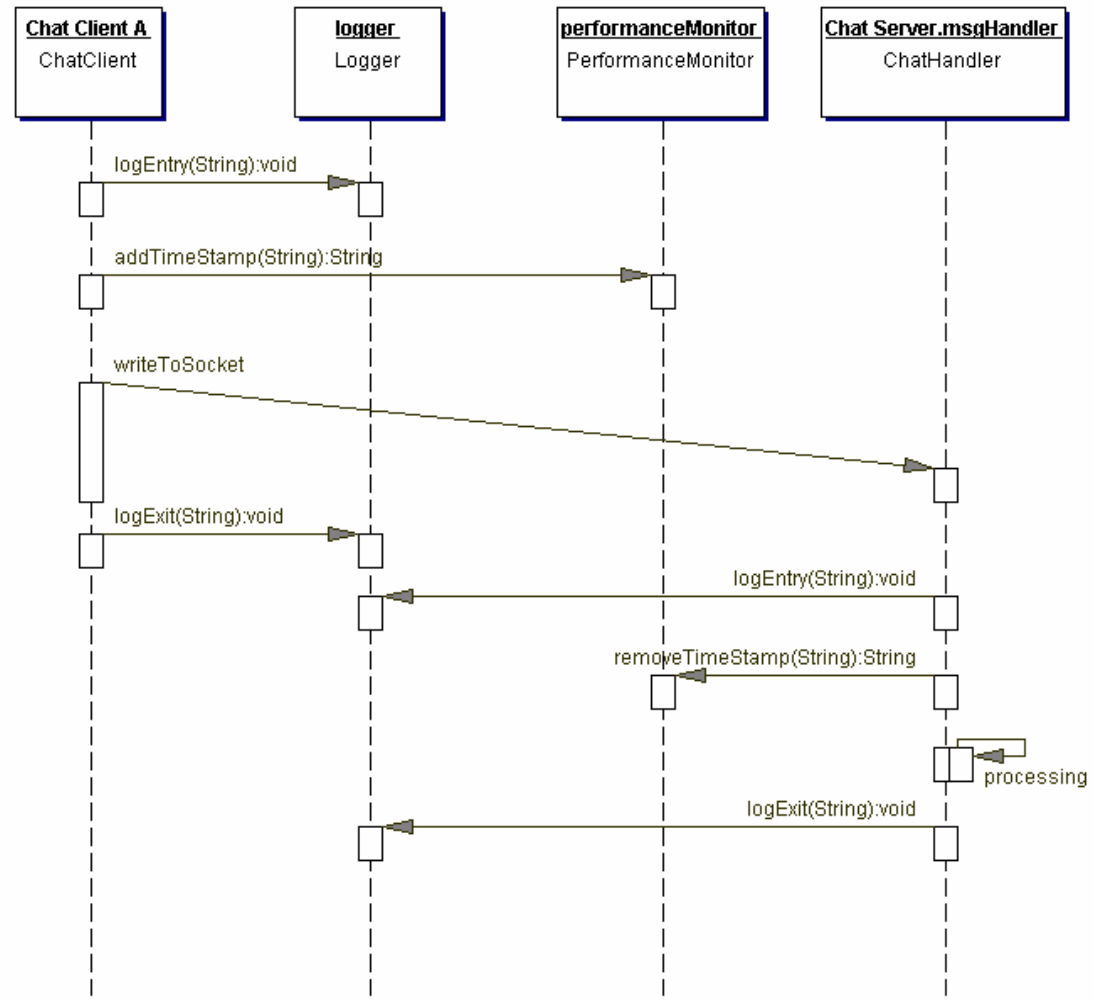

Figure 25 Sequence Diagram after adding timing NFR

"Figure 25 Sequence Diagram after adding timing NFR" specifies that a time stamp be added before the message is sent and removed after the message is received. Again, all the sequence diagrams need to be updated to reflect this new NFR. Also notice that the logging NFR and timing NFR are separated at the requirement level, but mangled together at the design level now. Following this design, the code will be mangled and

scattered as well.

"Figure 26 Sequence Diagram after adding encryption NFR" specifies that the message shall be encrypted before sending and decrypted after receiving. With three relatively simple NFRs, this sequence diagram has been modified three times and becomes more and more complex with each addition of a new NFR.

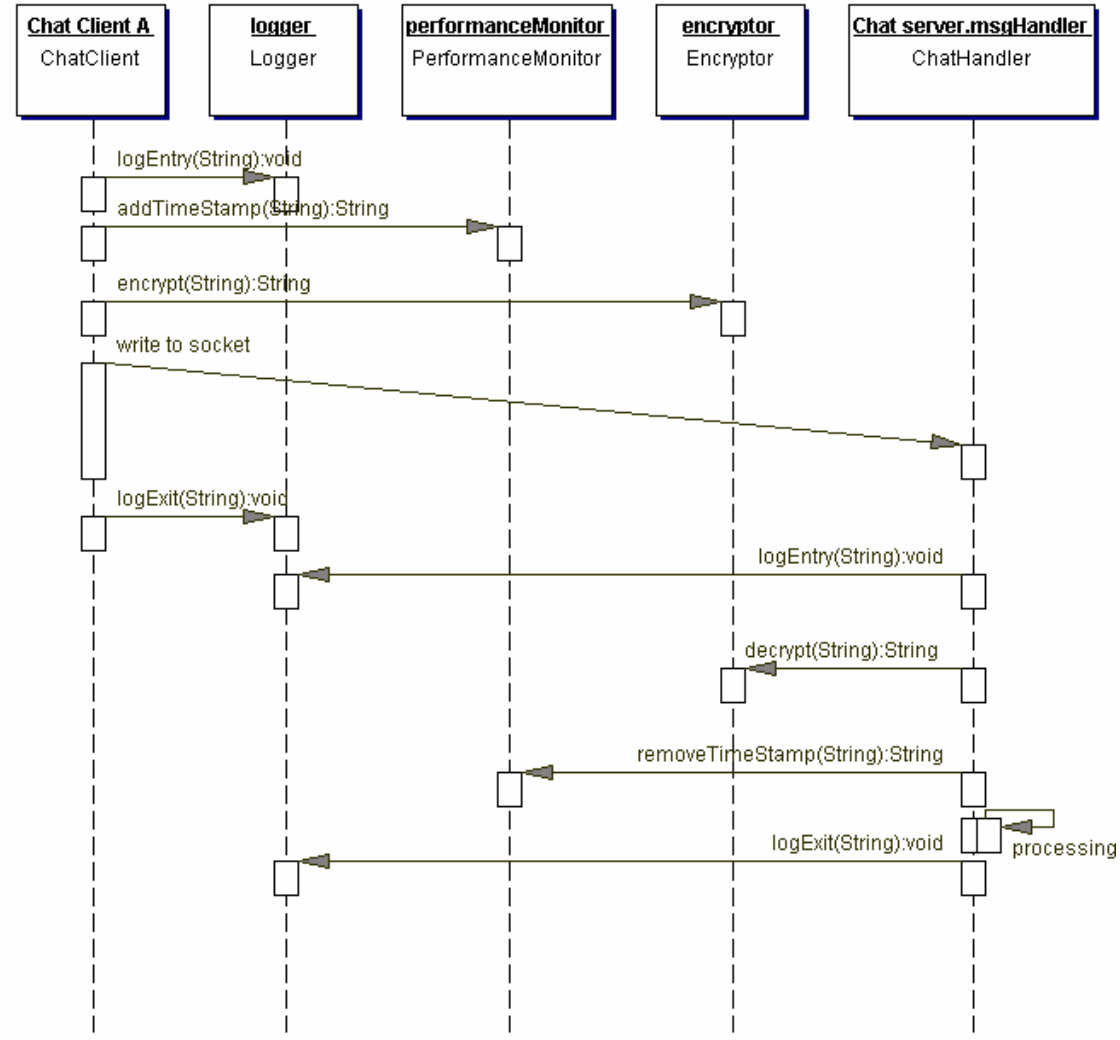

Figure 26 Sequence Diagram after adding encryption NFR

The design diagrams become more and more clumsy as we keep on modifying them with the additions of new NFRs. The modification to the code is even worse. Let us look at two examples for illustration purpose.

In order to support logging NFR: "trace method call", every method in the entire system has to be modified to add two statements at the beginning and at the end of the method body. This approach makes the number of lines of code much bigger, and requires huge amount of time, and is not very maintainable (every method needs modification if the logging API is changed). Comparing the simple and elegant MethodTracing Aspect (see section 5.6.5 and section 4.3.3.3) against modifying every method, the advantage of the proposed method is huge.

In order to support security NFR: "encryption", every call to the methods java.io.DataOutputStream.writeUTF() and java.io.DataInputStream.readUTF() must be modified. We need to define two new methods (e.g., sendSocketMessage() and recvSocketMessage()) first, and then replace every call to writeUTF() with sendSocketMessage(), replace every call to readUTF() with recvSocketMessage(). The implementation of sendSocketMessage and recvSocketMessage will handle the encryption and decryption of messages. This is an intrusive modification to the existing code and also design, it has scattered impact to the overall system.

The drawback of the traditional object-oriented method is that it does not have concepts

and mechanisms to crosscut its fundamental module -- object, while the design and implementation of NFRs requires exactly that ability to manage the complexity.

The proposed methodology uses the crosscutting nature of policy mechanisms to help manage the complexity, to achieve the separation between design and implementation artifacts for NFRs and those for FRs, and also to achieve the separation between the design and implementation artifacts for different NFRs.

A potential limitation of the proposed approach is the need to have a different compiler (i.e., AspectJ compiler instead of just Java compiler). If a particular project does not want to introduce the uncertainty of a new compiler, then this approach can not be used, at least not at the implementation phase.

CHAPTER 6 CONCLUSION

6.1 Summary

This thesis recognizes the problem that the changes of NFRs impact design and implementation in a scattered fashion. Then based on the Separation of Concerns principle, this thesis raises the question of how to modularize the design and implementation artifacts for NFRs.

Our initial hypothesis is that there are mechanisms to modularize the design and implementation artifacts for NFRs, and the fundamental nature of such mechanisms is "crosscutting semantically while centralized syntactically". We give a term to all such mechanisms: "Policy Mechanisms".

Then we study the characteristics of policy mechanisms. A list of attributes of policy mechanisms is provided. Each of the attributes is defined. All related policy mechanisms are analyzed by using the attribute list. Based on this analysis, we extend OCL and form PEOCL to represent design policies for NFRs, and use aspects (specifically AspectJ) to represent implementation level policies for NFRs.

PEOCL extends OCL by adding the UML metamodel and the NFR ontology. The UML metamodel provides us with the ability to reference a collection of model elements in UML class diagram. The NFR ontology makes it easy to express NFR-specific constraints.

Overall, PEOCL and AspectJ are suitable to represent design and implementation artifacts for NFRs, because of their crosscutting ability and their association with the main stream object-oriented methodology (i.e., UML and Java).

The case study has demonstrated how PEOCL can capture design policies for NFRs and how AspectJ aspects can implement PEOCL policies. The case study has also demonstrated the clean separation between the design and implementation artifacts for NFRs and those for FRs, and the separation among the design and implementation artifacts for different NFRs.

To sum up, this research work has done the following:

- Identified the problem of how to design and implement NFRs in a modular way
- Formally characterized policy mechanisms, and surveyed related policy mechanisms
- Extended OCL to form PEOCL
- Proposed a methodology to derive design policies from NFRs, and then implement design policies by using aspects
- Conducted a case study through implementing a distributed chat room system by using the proposed methodology
- Designed and implemented a generic abstract aspect library for common NFR concerns

The benefit of the proposed methodology stems mainly from the fact that the design and implementation for NFRs are separated from those for FRs. Separation of Concerns improves maintainability (modularity, non-intrusive evolution, readability, etc.) greatly [Dijkstra76], and thus helps to reduce the maintenance cost.

6.2 Future Work

The proposed methodology has some limitations and should be explored further in the future.

In our case study, most of the NFRs can be implemented by using policies with little effort, but some of them have not been implemented as policies. For example, all the GUI creation code is implemented by using traditional method and plain Java code, because it is simpler to modify the existing code to meet this new NFR and also there are GUI-generation tools readily available. It is not exactly clear to us what the general rule is, about when it is suitable to map NFRs to policies and when it is suitable to map NFRs to direct-modifications to the existing code.

We also attempted to create a graphical notation for policies at design level. The basic principle of designing a two-dimensional graphical notation is to use icons to represent concepts, and then to use one of these three ways to represent relations between two

concepts: a line, or attachment, or containment between the two icons. However, the crosscutting nature of policies makes it difficult to represent the relation between a policy and all the modules (classes, methods, attributes, etc.) that it crosscuts. The resulting diagram is too complicated to understand, even though the textual PEOCL and AspectJ policies are modularized and very easy to understand. The potential solution could lie in tool automation, i.e., to provide tools that provide multi-dimensional views of the classes and policies.

A chat room system is a non-trivial application, but larger case studies still are required to determine if this proposed methodology can reduce work, reduce the overall development time and cost. Our expectation is that the larger the system is, the more beneficial this methodology will be. Because the underlying features and modules that policies can crosscut increase as the system becomes bigger, it will be more costly to do it the traditional way (i.e., to modify them one by one), thus more savings are expected.

Larger case studies may also reveal any drawbacks in the proposed policy mechanisms for designing and implementing NFRs, and invent better policy mechanisms, the provided list of characteristics for policies could be useful to this work. For example, a known limitation of AspectJ is that it lacks strong conflict detection and resolution methods. Conflict resolution methods are typically required for specification level artifacts, where all statement should hold true simultaneously. Procedural programming

language like Java or AspectJ does not address those issues at the language level. It is not clear to us how to detect and resolve potential conflicts among multiple related aspects.

The benefit of this approach has been argued based on the well-established software engineering principle (i.e., the separations of concerns principle), and based on one case study. Wide-scope trial should be conducted by many more different programmers of different background and for different application domains. Statistics from the wide scope trial should be analyzed to make a conclusive evaluation of the proposed approach.

The distinction between Non-Functional Requirements and functional requirements is not a clear cut. This research work has been focusing on how to deal with NFRs at the design and implementation levels. However the results from this research work can be potentially applicable to functional features that crosscut many other functional features. It would be interesting to see case studies in this area.

We also would like to investigate in the future how this methodology impact the testing phase. We expect that the tracability among requirements, design artifacts, code, and test cases will be improved.

Traditionally development teams are organized surrounding features. Since NFRs crosscut many functional features, NFRs are typically distributed into all the feature

teams, and a prime coordinates all the activities related to NFRs. This incurs much additional overhead in communication and coordination among multiple teams. It would be interesting to find out how this methodology can impact the organization structure, e.g., whether it would be more effective to have a dedicated NFR team to implement all NFRs, by using the proposed methodology. The challenge will be in the areas of how to balance the power structure to ensure both the NFR team and the teams on functional features are motivated and willing to communicate with each other.

CHAPTER 7 APPENDIX: NFR ONTOLOGY

This is a list of terms from NFR Ontology [Chung00b, Babacci95] that are used in our case study. The signature of each term is presented.

Term from NFR Ontology	Signature of the term as used in PEOCL	Descriptions
encrypted	Boolean encrypted (String text);	'encrypted' is used as a predicate to indicate whether the text is encrypted or not.
timeStamped	Boolean timeStamped(String text);	'timeStamped' is used as a predicate to indicate whether the text has a time stamp in it or not.
timeInMessage	Date timeInMessage(String text);	'timeInMessage' returns the date and time encoded in the text.
now	Date now;	'Now' refers to the current date and time.
Log	Collection Log;	'Log' is a collection of logging messages

CHAPTER 8 REFERENCES

[Ahmed97] Amal Ahmed, “R++ User Manual for Release 1.1”, AT&T. May 15, 1997

[AOP01] ACM Communications of ACM, Vol44, issue 10, October 2001. Special issue on AOP

[AspectJ02] http://www.aspectj.org

[Babacci95] Mario Babacci, Mark H. Klein, Thomas A. Longstaff, Charles B. Winstock, “Quality Attributes”, CMU Technical Report, Document number: CMU/SEI-95-TR-021

[Barbuceanu98] Mihai Barbuceanu, Tom Gray, Serge Mankovski, "Coordinating with obligations", Proceedings of the second international conference on Autonomous agents 1998, ACM Press, pp. 62 – 69, 1998

[Bearden01] Mark Bearden, Sachin Garg, and Woei-jyh Lee, “Integrating Goal Specification in Policy-Based Management”, Proceedings of Policy 2001, LNCS 1995, pp. 153-170, 2001

[BlowFish02] http://www.counterpane.com/blowfish-download.html

[Bolognesi00] Tommaso Bolognesi, "Toward Constraint-Object-Oriented Development", IEEE Trans on SE Vol. 26, No. 7, July 2000

[Boutaba01] Raouf Boutaba, Andreas Polyrakis, “Towards Extensible Policy Enforcement Points”, Proceedings of Policy 2001, LNCS 1995, pp. 247-261, 2001

[Chung00a] L. Chung, D. Gross, E. Yu, "Architectural Design to Meet Stakeholder Requirements", in Software Architecture, Patrick Donohue, ed., Kluwer Academic Publishers, pp. 545-564, 1999

[Chung00b] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, "Non-functional requirements in software engineering", Boston: Kluwer Academic Publishers, 472 pp. ISBN 0-7923-8666-3, 2000

[Chung94] Lawrence Chung, Brian A. Nixon and Eric Yu , "Using Quality Requirements To Systematically Develop Quality Software", Fourth International Conference on Software Quality, McLean, VA, U.S.A. October 3–5, 1994

[Clocksin87] W.F. Clocksin and C.S. Mellish, "Programming in Prolog", Springer Verlag, 1987 ISBN 0-387-17539-3

[Cole01] James Cole, John Derrick, Zoran Milosevic, Kerry Raymond, "Policies in an Enterprise Specification", Policy 2001, LNCS 1995, pp. 1-17, 2001

[Corradi01] Antonio Corradi, naranker Dulay, Rebecca Montanari, Cesare Stefanelli, "Policy-Driven Management of Agent Systems", Policy 2001, LNCS 1995, pp. 214-229, 2001

[Damianou01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, Morris Sloman, "The Ponder Policy Specification Language", Policy 2001, LNCS 1995, pp. 18-38, 2001

[Dijkstra76] Edgar W. Dijkstra. "A Discipline of Programming", Prentice-Hall, Englewood, N.J., 1976.

[DMTFSLA02] The DMTF (Distrib uted Management Task Force) Service Level Agreement (SLA) Working Group, http://www.dmtf.org/info/sla.html

[FU01] Zhi Fu, S. Felix Wu, He Huang, Kung Loh, Fengming Gong, Ilia Baldine, and Chong Xu, “IPSec/VPN Security Policy: Correctness, Conflict Detection, and Resolution”, Policy 2001, LNCS1995, pp. 39-56, 2001.

[GAMMA97] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns -- Elements of Reusable Object-Oriented Software", Addison Welsley, 1997 (ISBN 0-201-63442-2)

[Gross00] Daniel Gross, Eric Yu, "From Non-Functional Requirements to Design through Patterns", Proceedings of the 5th Mitel workshop "Innovation in Technology & Applications", August 24th-25th, 2000

[Hitchens01] Michael Hitchens and Vijay Varadharajan, “Tower: A Language for Role Baesd Access Control”, Proceedings of Policy 2001, LNCS 1995, pp. 88-106, 2001

[Java00] Java Language Specification, Second Edition, 2000 Sun Microsystems, Inc.

[JLOGREF02] ILOG JRules Language Reference, version 3.0 (http://www.ilog.com/)

[JLOGUSER02] ILOG JRules User’s Manual, version 3.0

[Kanada01] Yasusi Kanada, “Taxonomy and Description of Policy Combination Methods”, Proceedings of Policy 2001, LNCS 1995, pp. 171-184, 2001

[Kazman00] Rick Kazman, Mark Klein, Paul Clements, “ATAM: Method for

Architecture Evaluation", CMU/SEI-2000-TR-004

[Kazman97] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, Jeomy Carriere, "The Architecture Tradeoff Analysis Method", TR, SEI, Carnegie Mellon University, Pittsburgh, PA 15213

[Litman97] Dianne Litman, Peter F. Patel-Schneider, Anil Mishra, "Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules", OOPSLA 1997

[Log4J02] Log4J: http://jakarta.apache.org

[Lutfiyya01] Hanan Lutfiyya, Gary Molenkamp, Michael Katchabaw, and Michael Bauer, "Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework", Policy 2001, LNCS 1995, pp. 185-201, 2001

[Martin98] James Martin, James J. Odell, "Object-oriented Methods – A Foundation: A UML Edition", Pentice Hall, 1998, ISBN 0-13-905597-5

[OCL97] "Object Constraint Language Specification", version 1.1, 1997, ad/97-08-08, Rational Software, Microsoft, HP, Oracle, IBM, etc. (The document can be found within this link: http://www.rational.com/uml, the exact link to the document is subject to change)

[Oracle99] "Java Stored Procedure Developer's Guide", Oracle 8i, release 2 (8.1.6), December 1999.

[PIB00] Policy Information Base (PIB) for Differentiated Service QoS Internet Draft, Network Working Group, "Differentiated Services Quality of Service Policy Information Base", November 24, 2000 (

http://ring.htcn.ne.jp/pub/doc/internet-drafts/draft-ietf-diffserv-pib-09.txt (the revision number is subject to change as newer revisions coming out), conform to Section 10 of RFC2026)

[RUP00] Software Process Engineering Management -- The Unified Process Model (UPM), Initial Submission, OMG document number ad/2000-05-05, May 12, 2000, Submitted by IBM, Rational Software, SofTeam, Unisys, Nihon Unisys Ltd., Alcatel, Q-Labs (ex-objectif technologies), Supported by Valtech, Toshiba

[Scott99] Michael L. Scott, “Programming Language Pragmatics”, October 1999, ISBN 1-55860-442-1

[Together02] http://www.togethersoft.com/

[UML00] “Unified Modeling Language Specification”, version 1.3, 2000, Rational Software

[UMLMeta97] UML Meta Model Specification, version 1.1, 1997, Rational Software

[Waldbusser00] Steve Waldbusser, Jon Saperia, Thippanna Hongal, Internet Draft "Policy-Based Management MIB", Nov 22, 2000 (http://ring.htcn.ne.jp/pub/doc/internet-drafts/draft-ietf-snmpconf-pm-04.txt, the revision number is subject to change)

[Weiss01] Weiss, M., Araujo, I., "Patterns and Non-Functional Requirements: An Interim Report", Proceedings of MICON 2001, "Patterns and NFR section", 2001

[XML00] Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, October 2000.

[XMLWriter02] XMLwriter release 1.21, http://XMLwriter.net/