ETS-RT-2011-001

FRAMEWORK FOR ASSESSING UNDERSTANDABILITY IMPACT OF DESIGN PATTERNS

SOUMARÉ H., CHRISTOPHER P. FUHRMAN

ETS-RT - 2011-001

FRAMEWORK FOR ASSESSING UNDERSTANDABILITY IMPACT OF DESIGN PATTERNS

CADRE POUR ÉVALUER LA COMPRÉHENSIBILITÉ D'UNE CONCEPTION QUI IMPLÉMENTE DES PATRONS DE CONCEPTION

ETS TECHNICAL REPORT

SOUMARÉ HADAMON,
CHRISTOPHER P. FUHRMAN

Département de génie logiciel et des technologies de l'information

ÉCOLE DE TECHNOLOGIE SUPERIEURE
UNIVERSITE DU QUEBEC

MONTRÉAL, 15 MAI 2011

ETS-RT - 2011-001

FRAMEWORK FOR ASSESSING UNDERSTANDABILITY IMPACT OF DESIGN PATTERNS

SOUMARÉ HADAMON,
CHRISTOPHER P. FUHRMAN
Département de génie logiciel et des technologies de l'information
ÉCOLE DE TECHNOLOGIE SUPERIEURE

ISBN 978-2-921145-76-3

Legal Deposit : Bibliothèque et Archives nationales du Québec, 2011
Legal Deposit : Library and Archives Canada, 2011

CADRE POUR ÉVALUER LA COMPRÉHENSIBILITÉ D'UNE CONCEPTION QUI IMPLÉMENTE DES PATRONS DE CONCEPTION

SOUMARÉ HADAMON,
CHRISTOPHER P. FUHRMAN

SOMMAIRE

La *maintenabilité* du logiciel est non seulement affectée par la facilité avec laquelle il peut être modifié, mais aussi par l'aisance avec laquelle il se laisse comprendre. Par la philosophie de la conception qui vise l'extensibilité, « *design for change* », beaucoup de patrons de conception orientée objet cherchent à améliorer la facilité de modification du logiciel. Cependant, ces patrons nuisent aussi à la compréhensibilité du logiciel. Nous proposons un cadre pour évaluer l'impact de l'aspect statique des patrons de conception orientée objet sur les qualités de compréhensibilité et de facilité de modification du logiciel. Notre cadre est basé sur l'évaluation de l'effet des principes GRASP, principes logiciels généraux d'assignation des responsabilités, sur les qualités de compréhensibilité et de facilité de modification du logiciel. Nous étendons par la suite cette évaluation aux patrons de conception orientée objet que nous montrons comme étant structurellement constitués de principes GRASP. Nous démontrons l'applicabilité de notre cadre en utilisant des graphes d'interconnexion de « *softgoal* » (SIGs). Dans ces graphes, les patrons de conception sont considérés comme étant des solutions techniques, encore appelées « *operationalizations* » dans le jargon des SIGs. Ainsi les impacts des patrons de conception sur les qualités de compréhensibilité et de facilité de modification pourront être estimés et documentés avant toute décision du concepteur de les appliquer.

FRAMEWORK FOR ASSESSING UNDERSTANDABILITY IMPACT OF DESIGN PATTERNS

SOUMARÉ HADAMON,
CHRISTOPHER P. FUHRMAN

ABSTRACT

Software maintainability is affected not only by its modifiability but also by its understandability. Many object-oriented design patterns are intended to improve a design's modifiability, per the "design for change" philosophy. However, patterns detract from a design's understandability for various reasons. We propose an assessment framework for design patterns to qualify the impact of their static structure on a design's modifiability and understandability. Our framework is based on an assessment of fundamental General Responsibility Assignment Software Patterns (GRASP) in terms of characteristics related to understandability as well as modifiability. We apply the GRASP assessment to several Gang of Four (GoF) patterns, which we show to be structural compositions of the GRASP principles. We demonstrate our framework's applicability with Softgoal Interdependency Graphs (SIG) where GoF patterns are operationalizations. Their impact on modifiability and understandability can be evaluated and recorded before a decision is made to apply them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Designs in object-oriented (OO) software are compromises among various forces, some of which are in opposition. OO design patterns, such as many proposed by the Gang of Four (GoF) (Gamma et al., 1995), seek to provide design structures that solve problems arising from trying to make a software design highly modifiable. This philosophy is known as “design for change” as it seeks to maximize reuse by proposing designs that anticipate new requirements in specific ways. However, because of their complexity, design patterns also have negative consequences on a design. Although the GoF documentation sometimes presents the negative consequences of patterns, the document gives little guidance to help decide whether the benefits outweigh the disadvantages. Ideally, a design pattern should be applied only when its impact is fully understood and can be judged as having a net positive effect on a design.

The GoF patterns are presented in a consistent format with different aspects: *intent*, *motivation*, *applicability*, *structure*, *collaborations*, *consequences*, *implementation*, etc. In the section on *consequences* for each pattern, the GoF authors explain the impact of applying the pattern, in terms of strengths and weaknesses to the design. Despite being part of the consistent format, the documented consequences are not always formulated in terms of an impact on non-functional requirements. Furthermore, in almost all cases, design patterns add complexity to a design, which can be seen as a negative consequence. Indeed, overzealous application of design patterns can have a negative impact on a system (Wendorff, 2001).

Extending software to support future requirements is part of software maintenance, but it also includes other activities. To evaluate software in terms of its maintainability, Boehm et al proposed (Boehm, Brown et Lipow, 1976) quality characteristics such as testability, understandability and modifiability. Using a relaxed and modern interpretation of these characteristics (which were initially developed for procedural languages popular in the 1970s), we submit that “design for change” seeks to

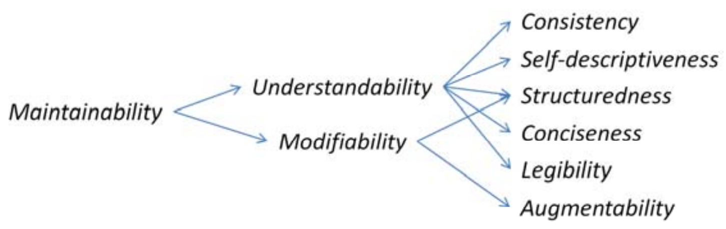

Fig. 1. Software quality characteristics relating to Maintainability (Boehm, Brown

maximize modifiability in a software design, while ignoring the impact on other characteristics such as testability or understandability. In this present work, we consider the problem of assessing in greater detail how GoF patterns affect a design's understandability.

Understandability is defined in (Boehm, Brown et Lipow, 1976) as "code [whose] purpose is clear to the inspector." As illustrated in Fig. 1, the authors show this characteristic is part of Maintainability, but that it is also dependent on simpler characteristics such as consistency, self-descriptiveness, structuredness, conciseness and legibility. These concepts still apply to modern, object-oriented software, even though their definitions in (Boehm, Brown et Lipow, 1976) were written at a time when polymorphism, dynamic memory allocation and class inheritance were mostly non-existent. In the present work, we use these characteristics with modern interpretations to assess the effect that GoF patterns have on understandability.

After the success of the GoF patterns, many hundreds of other patterns have been documented. To help understand the overwhelming number of patterns, the author in (Larman, 2005) proposes just nine basic principles known as General Responsibility Assignment Software Patterns (GRASP) principles. Nearly all the GoF patterns can be viewed as some blend of the four GRASP principles known as *Indirection*, *Polymorphism*, *Pure Fabrication* and *Protected Variations*. To better understand the impact of applying the more complex GoF patterns to a design, we seek to model them as these fundamental GRASP principles. We consider modeling qualitatively the positive and negative consequences of design decisions at the GRASP level, because it is easier to understand the impact that a GRASP decision has on a particular non-functional characteristic. Once the impact of the various GRASP principles is understood, we consider modeling the impact at a higher level, such as in the GoF patterns that are comprised of these principles. Our work presents an assessment of three (3) of the GRASP principles in terms of understandability and modifiability. We extend our assessment of GRASP to construct a framework used to evaluate design patterns, based on the natural relationship between design patterns and the underlying GRASP principles.

To apply this framework, we propose using it with Softgoal Interdependency Graphs (SIG) (Chung, 1999) which have been used to model non-functional requirements (NFR). Design patterns can be viewed as *operationalizations* in this context. In effect, before applying a design pattern, by modeling it with a SIG the designer can assess its impact on various NFRs, including understandability.

This work has the following contributions: 1) an assessment of GRASP principles in terms of understandability and modifiability, 2) an identification of GRASP principles in several GoF design patterns, 3) an application of this information using SIG to assess the impact that design patterns have on both understandability and modifiability.

We present the work in this paper as follows. Section 2 describes the framework based on non-functional characteristics of design decisions. Section 3 presents the assessment of the GRASP principles in the framework, and its extension to several GoF patterns. Section 4 is an integration of our assessment within GoF patterns, to assess them in terms of the design characteristics that interest us. Section 5 illustrates the use of our framework with SIGs using example GoF patterns. Section 6 discusses the work related to our research. Section 7 discusses the results and future work of our research. Section 8 concludes the paper.

2 FRAMEWORK FOR ASSESSING MAINTAINABILITY

Our framework works with an assumption that OO design is responsibility-driven. That is, software elements have responsibilities which are an abstraction of what functions they fulfill or roles they play in the design. In this paradigm, GRASP principles are the elemental impetuses of object-oriented design decisions. For example, in the approach proposed by Larman (Larman, 2005), designers document decisions about collaborations and responsibilities between objects with annotations in UML interaction diagrams. These annotations are generally in the form of GRASP principles.

Another precondition for our framework is that software is being developed following a process whereby software objects generally model real-world (domain) objects, to achieve a *low representational gap* (Larman, 2005). For example, in the development method presented by Larman (Larman, 2005), a designer models the functional requirements in use-case diagrams (capturing behavior) and in UML class diagrams representing the so-called domain model. The objects in the domain model are intermediate metaphors, and their goal is to reduce the representational gap between the requirements and the design. As such, proposing a design object that corresponds to a domain object can be considered a design rationale in itself—it is an object that is easily traceable to the requirements and therefore helps the understandability of the design. This motivation is important, since some objects existing in a design do not easily trace to the requirements, making a design harder to understand.

Our framework provides an assessment of three of the nine GRASP principles with respect to the quality characteristics for understandability shown in Table 1. Since the initial definitions of understandability and modifiability (Boehm, Brown et Lipow, 1976) were intended for software that was not object-oriented, we add cohesion as a characteristic. We include Self-descriptiveness for consistency with (Boehm, Brown et Lipow, 1976), yet we were unable to easily find a way to map this definition to the GRASP principles. Similarly, we redefine augmentability, as its definition in

Table 1. Quality characteristics relating to modifiability and understandability

Characteristic	Adapted definition
Cohesion	"the extent to which the individual components of the module perform the same task. … a measure of relatedness in functionality." (Bansiya, 1997)
Conciseness	"the extent that excessive information is not present. This implies that [responsibilities] are not excessively fragmented into [classes], nor that the same [responsibility] is repeated *n* numerous places." (Boehm, Brown et Lipow, 1976)
External consistency	"the extent that the content is traceable to the requirements." (Boehm, Brown et Lipow, 1976) "OO designs and languages support Low Representational Gap (LRG) between software components and mental models of a domain. That improves comprehension." (Larman, 2005)
Internal consistency	"the extent that [a module] contains uniform notation, terminology, and symbology within itself." (Bansiya, 1997) "[A module's] elements contribute to and reinforce its overall intent or effect." (Dromey, 1995)
Legibility	"the extent that [a module's] function is easily discerned by reading the code [design]." (Boehm, Brown et Lipow, 1976)
Self-descriptiveness	"the extent that [modules] contain enough information for a reader to determine or verify [their] objectives, assumptions, constraints, inputs, outputs, components, and revision status. Commentary and traceability of previous changes by transforming previous versions of code into non-executable but present (or available by macro calls) code are some of the ways of providing this characteristic." (Boehm, Brown et Lipow, 1976)
Structuredness	"the extent that [modules] possess a definite pattern of organization among themselves. This implies that evolution of the program design has proceeded in an orderly and systematic manner, and that standard modular design structures have been followed." (Boehm, Brown et Lipow, 1976)
Augmentability	The extent that modules are organized to encapsulate responsibilities among objects, through *protected variation* (Larman, 2001).

(Boehm, Brown et Lipow, 1976) seems to apply to general class structures in OO software. We present the details of our assessment in the following section.

3 ASSESSEMENT OF GRASP PRINCIPLES

We assess the GRASP principles according to the quality characteristics relating to understandability. We consider the four GRASP principles known as Indirection, Polymorphism, Pure Fabrication and Protected Variations, although Protected Variations always takes the form of certain combinations of the other three. To understand them in terms of design choices, we describe them as refactorings, and present basic examples that we use for our assessment.

3.1 Indirection

According to Larman (Larman, 2005), Indirection is defined as: "Problem: Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to de-couple objects so that low coupling is supported and reuse potential remains higher? Solution: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. The intermediary creates an indirection between the other components."

Indirection is illustrated in Fig. 2, where a responsibility R in A that requires a coupling to B is transferred to an intermediate class C such that A is no longer coupled with B. The assessment for Indirection with respect to Understandability is shown in

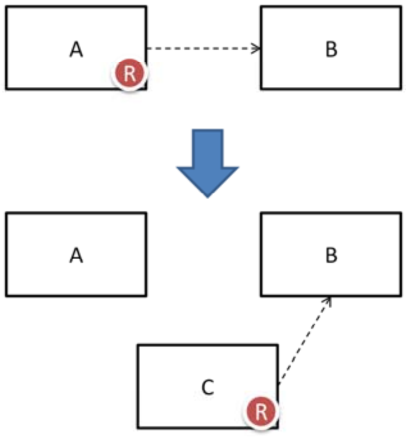

Fig. 2. Indirection is a transfer of a responsibility that caused undesired coupling

Table 2. The notation for this table is as follows: + = positive impact, (+) = potentially positive impact, − = negative impact, and (−) = potentially negative impact.

Table 2. Quality characteristics of Indirection

Quality characteristic	Impact	Explanation
Cohesion	+	A has fewer responsibilities, R is cohesive with C
Conciseness	+	R is not repeated, but displaced to C
External consistency		Does not affect traceability to requirements
Internal consistency	+	Similar to cohesion
Legibility	−	Each level of indirection makes the flow harder to understand.
Self-descriptiveness		Does not appear to affect characteristic
Structuredness	+	Indirection leads to modular design
Augmentability	(+)	May lead to protected variation.

3.2 Polymorphism

Polymorphism is defined by Larman (Larman, 2005) as follows: “Problem: How to handle alternatives based on type? How to create pluggable software components? Solution: When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies.”

Polymorphism is illustrated in Fig. 3 as a refactoring (also defined by Fowler (Fowler et Beck, 1999)) where a condition-based responsibility R in method m() in class A is broken up and assigned instead to subclasses B, C, … through a polymorphic operation m(), according to the conditional variations.

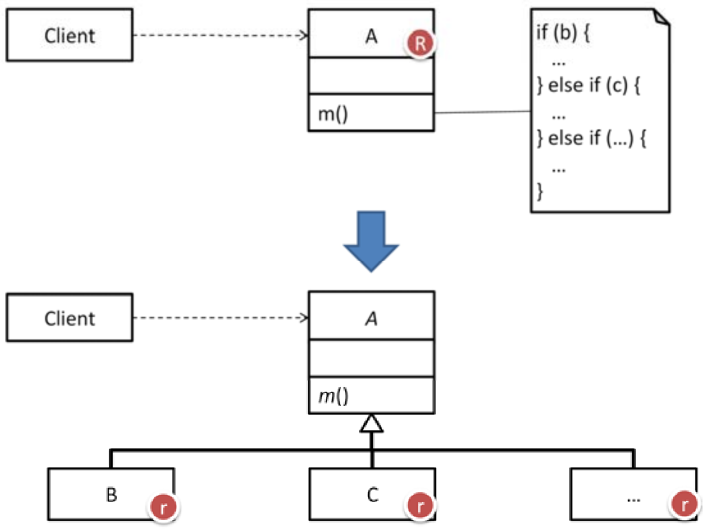

Fig. 3. Polymorphism transfers responsibilities to subclasses

The assessment for Polymorphism with respect to understandability is shown in Table 3.

Table 3. Quality characteristics of Polymorphism

Quality characteristic	Impact	Explanation
Cohesion		*A* is a cohesive theme across its subclasses, but they may not be individually cohesive
Conciseness	(−)	Possible excessive fragmentation of responsibilities, although not repetition
External consistency		Does not affect traceability to requirements
Internal consistency	+	All subclasses of *A* implement m()
Legibility	−	Subclasses are understood in a hierarchy
Self-descriptiveness		Does not appear to affect characteristic
Structuredness	+	Inheritance hierarchy imposes organization
Augmentability	+	Easy to add polymorphic variants (Protected Variation)

3.3 Pure Fabrication

Larman (Larman, 2005) defines Pure Fabrication as follows: “Problem: What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but [other] solutions […] are not appropriate? Solution: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept—something made up, to support high cohesion, low coupling, and reuse.”

Pure Fabrication is illustrated as a refactoring where, when a responsibility R in class A causes an undesirable coupling to class B in Fig. 4(a) or does not fit with the cohesive theme of class A in Fig. 4(b), the responsibility R (and the possible coupling to B) is displaced to class C. Two aspects are important: 1) class C is created to have a cohesive theme to fit the responsibility R, and 2) class C is not traceable to the domain model. In terms of the OO development process proposed by Larman, it means C does not represent a real-world set of objects and thereby increases the representational gap of the software design.

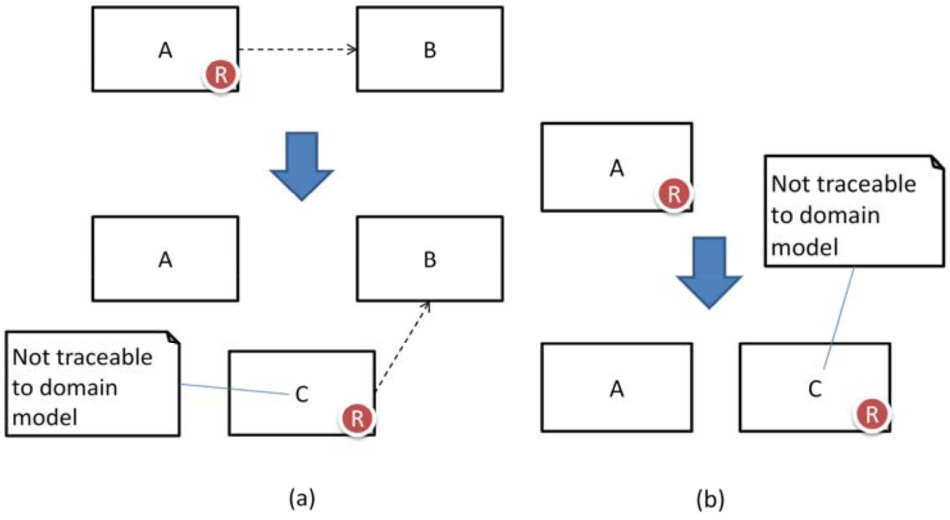

Fig. 4. Pure Fabrication with Indirection (a) and without (b)

Table 4 shows the assessment for Pure Fabrication without considering Indirection. That is, it corresponds to the refactoring in Fig. 4(b).

Table 4. Quality characteristics of Pure Fabrication (PF)

Quality characteristic	Impact	Explanation
Cohesion	+	A has fewer responsibilities; C is cohesive
Conciseness	(−)	Risk of fragmented responsibilities in each PF
External consistency	−	C does not trace to the domain model
Internal consistency	+	Similar to cohesion
Legibility		Does not appear to affect characteristic
Self-descriptiveness		Does not appear to affect characteristic
Structuredness	+	Modularity is enhanced, because C is cohesive and reusable.
Augmentability		Does not appear to affect characteristic

3.4 Protected Variations

According to Larman (Larman, 2005), Protected Variations is defined as follows: "Problem: How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements? Solution: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them." Protected Variations is viewed as a refactoring that potentially combines the three other GRASP principles, Indirection, Polymorphism and Pure Fabrication.

In the refactoring shown in Fig. 5, class A depends on a stable class C rather than variable (unstable) class(es) B1, (B2, B3, etc.) shown with V. This is a form of Indirection. A part of the responsibility R (relating to the coupling to the B classes) is transferred to C, shown by r. The responsibility R' indicates that A has simpler responsibilities after the Protect Variations refactoring is applied. If C is a class that does not map to the domain model, then it is a Pure Fabrication.

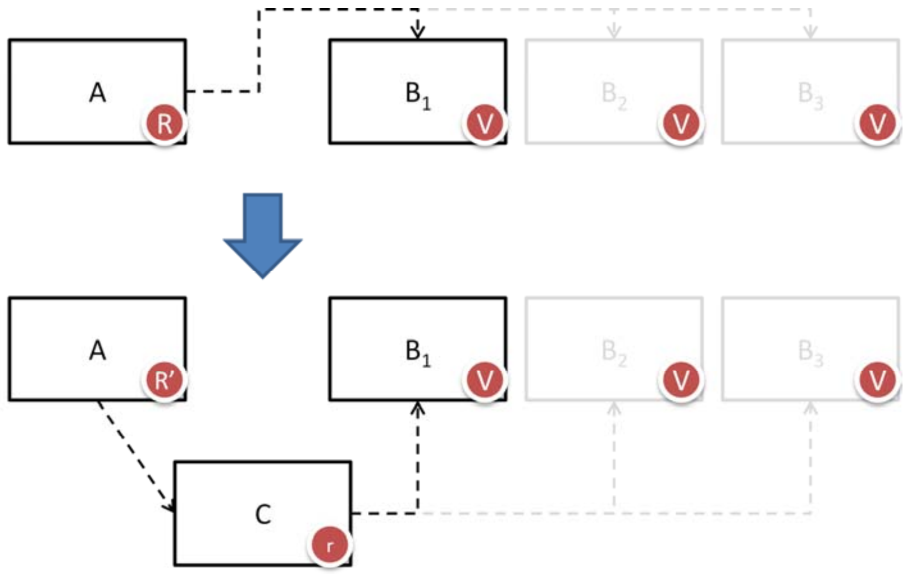

Fig. 5. Protected Variation with aggregation

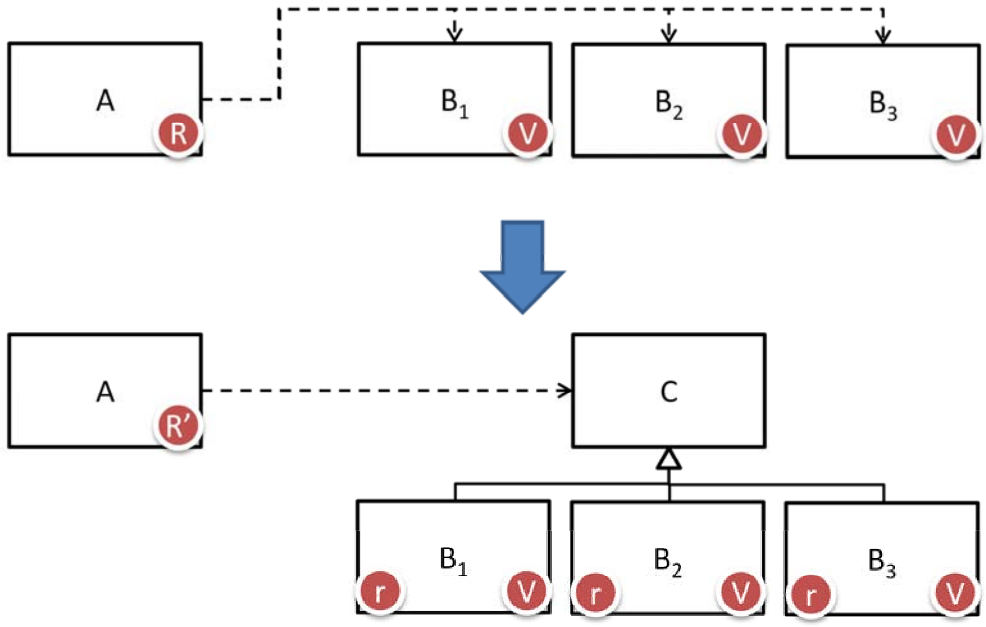

Fig. 6. Protected Variation with polymorphism

In the refactoring shown in Fig. 6, class A depends on a stable class C, rather than on the variable (unstable) classes B1, B2, B3, indicated with V, thanks to both Indirection and Polymorphism. Indirection is evidenced by A's lowered responsibility R' since it no longer is coupled directly to the classes B1, B2, B3, which have all taken part of that responsibility, shown by r. Polymorphism is evidenced by C having a polymorphic method (not shown in the figure) to accomplish those responsibilities that vary by type. As with the previous refactoring, if C does not trace to the domain model, it would be a Pure Fabrication.

Table 5 presents the assessment for Protected Variations, which as shown above can have different forms. We conclude its impact on quality characteristics depends on the existence of the other three GRASP principles that can be present in Protected

Table 5. Quality characteristics of Protected Variations

Quality characteristic	Impact	Explanation
Augmentability	+	Makes design easier to modify in the dimension provided by the protected variation

Variations. Therefore, we only include augmentability in the table, as Protected Variations has a definite impact on this regardless of its form. We explain the relationship between the GRASP principles in the following section.

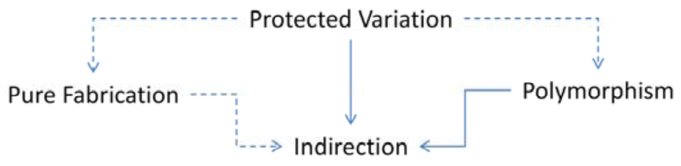

Fig. 7. Relationship between GRASP principles (dashed lines = possible relation)

3.5 Inter-GRASP relationships and summary

The above assessments provide insight to the relationship of GRASP principles among themselves. We present them in the general case in Fig. 7. Protected Variations can be seen as always an application of Indirection, with possibly Polymorphism and Pure Fabrication. Pure Fabrication can sometimes be an application of Indirection. Polymorphism is always an Indirection.

Table 6 presents the combined results of our assessment of GRASP principles with respect to understandability. Again, because Protected Variations is a special case depending on the other GRASP principles that comprise it, we do not include it in this table.

Table 6. Impact of GRASP refactorings on quality characteristics

Understandability/Modifiability	Indirection	Polymorphism	Pure Fabrication
Cohesion	+		+
Conciseness	+	(−)	(−)
External consistency			−
Internal consistency	+	+	+
Legibility	−	−	
Self-descriptiveness			
Structuredness	+	+	+
Augmentability	(+)	+	

4 UNDERSTANDABILITY/MODIFIABILITY IMPACT OF GOF

To better understand how GoF patterns affect not only modifiability but also understandability, we analyze the patterns within our framework. First, we identify which GRASP principles exist in each GoF pattern. Second, we re-use the impact assessment of the GRASP principles in the context of the application of a GoF pattern.

For each GRASP principle, we considered whether the structural aspects of the design pattern include it as a refactoring. For example, in the Abstract Factory pattern shown in Fig. 8, the Client class has delegated responsibilities to Concrete Factories as a way to avoid direct coupling with the Product classes. This is an example of Indirection, Polymorphism and Pure Fabrication. For most patterns, it is easy to identify Indirection and Polymorphism, as the structure is clear. Presence of the Pure Fabrication principle is less clear. For our analysis, we only consider Pure Fabrication to be present if a class is likely not analogous to a real-world set of objects. In this example, Product classes are likely objects inspired by real-world objects. However, the AbstractFactory hierarchy is most definitely a Pure Fabrication.

The results of our interpretation of the GRASP principles within the GoF patterns are shown in Table 7. Some notable cases include Singleton (which has no GRASP principles), Facade (which has no Polymorphism, yet is a Protected Variation) and Flyweight (which we considered to have Protected Variation because instantiation of

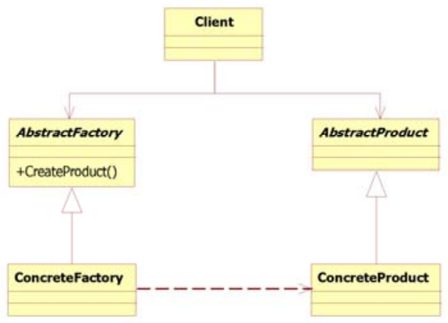

Fig. 8. Abstract Factory structure

Table 7. GRASP principles in each of the GoF Design Patterns

GoF Design Pattern	Indirection	Polymorphism	Pure Fabrication	Protected Variation
Abstract factory	d	d	d	d
Builder	d	d	d	d
Factory method	d	d		d
Prototype	d	d		d
Singleton				
Adapter	d	d	d	d
Bridge	d	d	d	d
Composite	d	d		d
Decorator	d	d		d
Facade	d		d	d
Flyweight	d	d	d	d
Proxy	d	d	d	d
Chain of responsibility	d	d	d	d
Command	d	d	d	d
Interpreter	d	d	d	d
Iterator	d	d	d	d
Mediator	d	d	d	d
Memento	d		d	d
Observer	d	d	d	d
State	d	d		d
Strategy	d	d		d
Template method	d	d		d
Visitor	d	d	d	d

Flyweight objects is encapsulated in the FlyweightFactory, even though the Client is directly coupled to Flyweight subclasses).

Since patterns are always applied in a specific context, the results in our table are generalized and should be used as a guide. One would have to examine a pattern's application to be sure of which GRASP principles are really being used. For example, in the case Chain of responsibility GoF pattern, the Handler hierarchy was considered a Pure Fabrication. In a real application, however, the Handler objects could actually be analogous to real-world objects. In such a specific case, the effects of the Pure Fabrication would not apply directly.

In the next section, we demonstrate how to apply our framework to evaluate the understandability and maintainability characteristics of a GoF design pattern.

5 APPLICATION WITH SIG

In this section, we demonstrate how a GoF pattern can be assessed in terms of understandability through the use of our framework applied with Softgoal Interdependency Graphs (SIG) (Chung, 1999). A SIG allows a designer to trace high-level non-functional goals, such as the design qualities described in Table 1, through lower-level scenarios to the actual design choices, known as operationalizations. In our method, we consider that a GoF design pattern is an operationalization composed of various GRASP principles (also operationalizations), which each have an impact on the design qualities (softgoals).

In the examples in this section, we make the following assumptions. First, if a GRASP has been shown in Table 6 to have a positive impact (potential or otherwise), we consider it to be a '+' helps contribution to satisficing the corresponding softgoal (design quality). Similarly, if a GRASP shows any form of a negative impact, we consider it to be a '−' hurts contribution to satisficing the corresponding softgoal. When there is no impact, we make no contribution, e.g., Self-descriptiveness is not affected by any GRASP principles according to Table 6.

Second, because of the inter-GRASP relationship defined in Fig. 7, we simplify our framework by considering that only an instance of Protected Variation contributes positively to Augmentability, despite the entries for the other GRASP principles for this characteristic in Table 6. Similarly, we map the GRASP principle contributing to a Protected Variation in a GoF pattern using the 'and' notation in SIG. For simplicity of modeling, we use 'eql' equals contribution between a GoF operationalization and the GRASP operationalizations that comprise it.

Finally, because certain GoF patterns contain more than one instance of a given GRASP principle, we can repeat this GRASP in the SIG to reflect the added impact of the principle in the operationalization. For example, the Visitor pattern makes use of Polymorphism two times in its structure.

In the example shown in Fig. 9 we present the Observer pattern as a SIG showing its impact on the non-functional requirements. From Table 7 we see that all of the GRASP principles are involved in the Observer pattern. For simplicity, we assume that the Pure Fabrication occurs only once—the Observer role. We do not consider the Subject role as an instance of a Pure Fabrication, even though in certain variants of this pattern it certainly could be. Similarly, we do not consider the polymorphism used by the Subject hierarchy as part of this SIG. Therefore, Polymorphism is only applied as a GRASP one time in the SIG representing the Observer hierarchy. The Protected Variation is an operationalization deriving from Indirection, Polymorphism and Pure

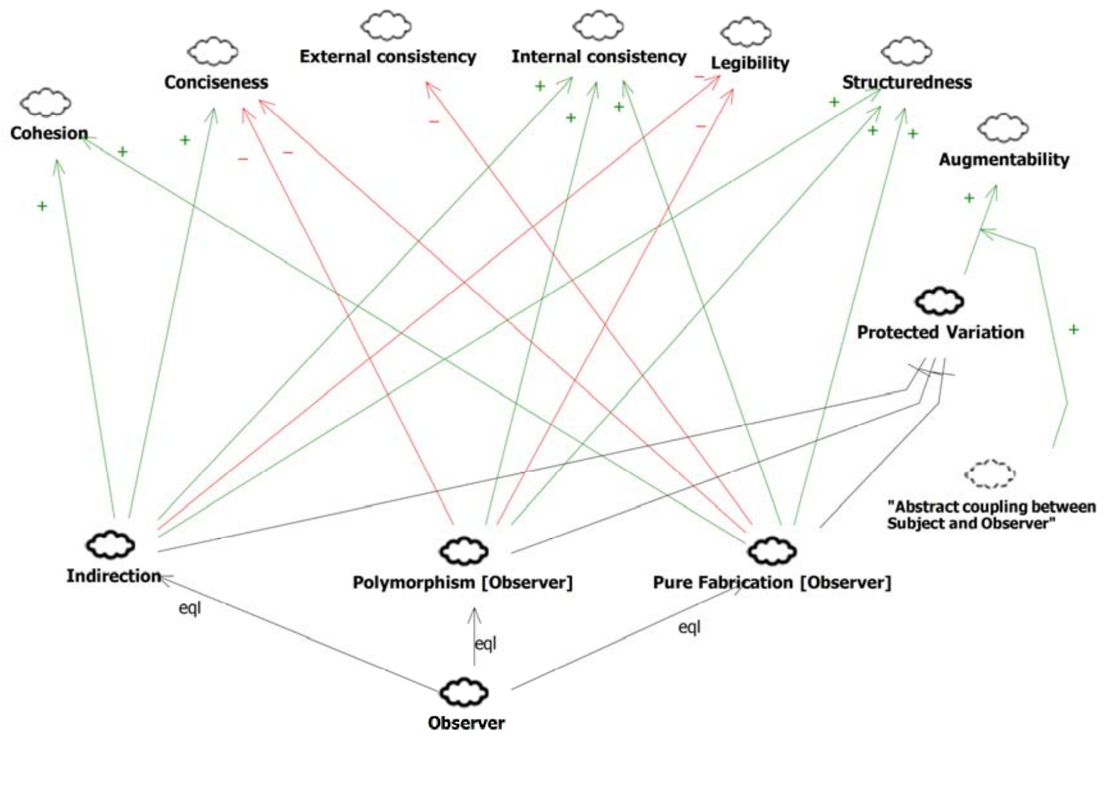

Fig. 9. Observer pattern with GRASP principles placed into a SIG

Fabrication. In the Consequences section for the Observer pattern in (Gamma et al., 1995) it is cited that "Abstract coupling between Subject and Observer" is a benefit. We illustrate this in the SIG as a claim softgoal, contributing positively to the Augmentability provided by the Protected Variation.

Understandability and Modifiability are related to the highest-level softgoals as illustrated in Fig. 1, but we did not include them in our example SIG to keep things simple. It becomes clear that several aspects of the Observer pattern contribute positively to Modifiability, via Structuredness and Augmentability which are strengthened by the pattern. In terms of Understandability, however, we see that things are not as clear. Cohesion and Internal Consistency benefit from the pattern, but Conciseness receives a mostly negative impact with some benefit, and External

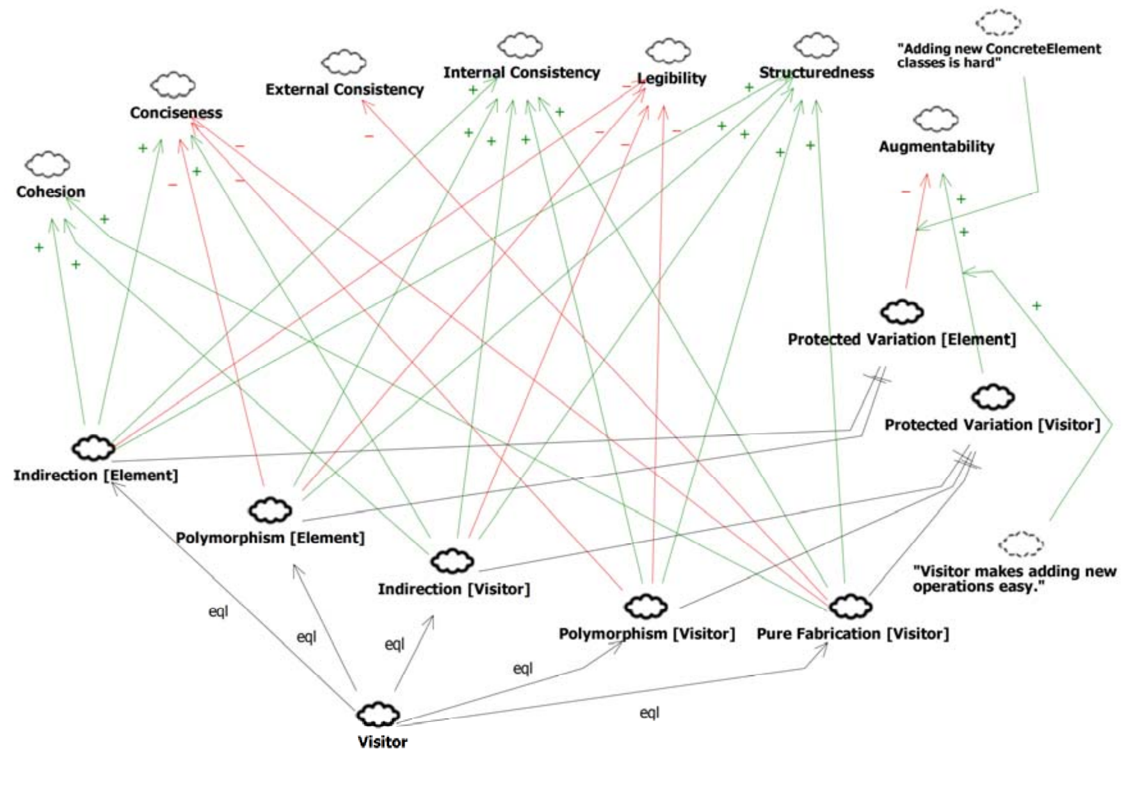

Fig. 10. Visitor pattern with GRASP principles placed into a SIG

Consistency and Legibility are clearly negatively affected.

Fig. 10 illustrates the Visitor pattern applied to our framework with a SIG. This example is used because it illustrates multiple instances of a Protected Variation through Polymorphism. We consider that the Element hierarchy is not a Pure Fabrication, whereas the Visitor hierarchy most definitely is. In terms of impact on the softgoals, we see that Cohesion, Internal Consistency and Structuredness benefit from the Visitor pattern. Because this pattern has two dimensions of variability (both the Polymorphism and the Element hierarchies), it has a strong negative impact on Legibility and to some degree Conciseness.

It would seem that both Protected Variations should benefit Augmentability. However, we chose to alter this aspect in the SIG, namely the Protected Variation of Element on Augmentability. The direct coupling of the Visitors to the ConcreteElements makes this pattern's structure difficult to add new ConcreteElements. The Protected Variation of Element is beneficial in this pattern, but within a limited scope of the ObjectStructure aggregate proposed by the pattern, which is why we chose to alter this GRASP principle's contribution to Augmentability, using a softgoal claim (repeated from the Consequence section of the Visitor pattern).

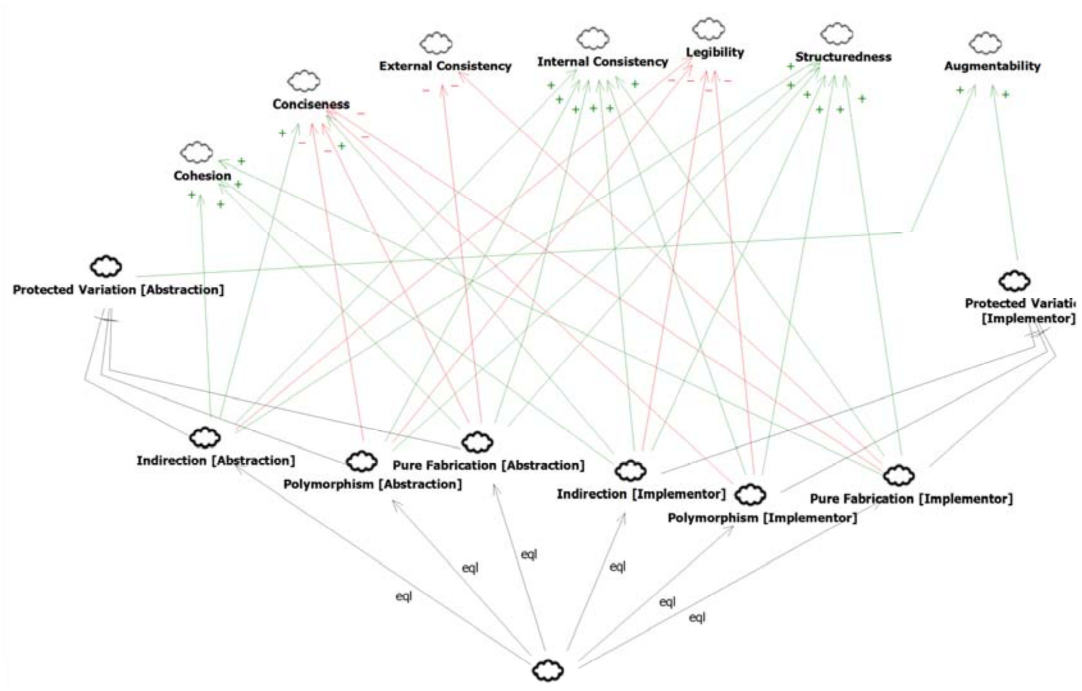
External Consistency
Internal Consistency
Legibility
Structuredness
Augmentability
Conciseness
Cohesion
Protected Variation [Abstraction]
Protected Variati
[Implementor]
Indirection [Abstraction]
Pure Fabrication [Abstraction]
Polymorphism [Abstraction]
Indirection [Implementor]
Pure Fabrication [Implementor]
Polymorphism [Implementor]
eql

Fig. 11. Bridge pattern with GRASP principles placed into a SIG

The next example we show in Fig. 11 is the Bridge pattern. This pattern has similar structure to the Visitor pattern, because it has two dimensions of inheritance. However, as shown in the SIG, we chose to consider both dimensions as contributing to the Augmentability characteristic. The SIG shows that Cohesion, Internal Consistency, Structuredness and Augmentability benefit strongly from this pattern, whereas Conciseness, External Consistency and Legibility suffer.

The last example is the Facade pattern shown in Fig. 12. This pattern is interesting because it provides a Protected Variation mechanism without using Polymorphism. The figure shows again mixed results for the various characteristics. We also point out in this example (because a Facade is a kind of flexible Protected Variation) another aspect of modeling patterns in our framework that shows the limitation of using GRASP principles. The GoF document states that the Facade pattern “doesn’t prevent applications from using subsystem classes if they need to.” We felt it was therefore

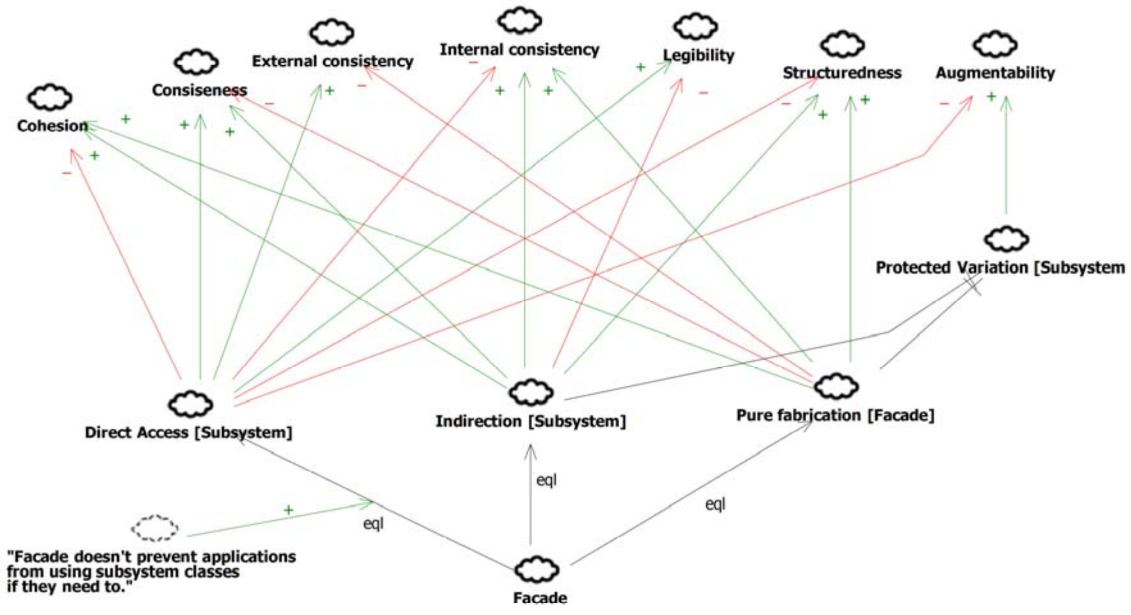
External consistency
Internal consistency
Legibility
Consiseness
Structuredness
Augmentability
Cohesion
Protected Variation [Subsystem
Direct Access [Subsystem]
Indirection [Subsystem]
Pure fabrication [Facade]
eql
eql
eql
"Facade doesn't prevent applications from using subsystem classes if they need to."
Facade

Fig. 12. Facade pattern with GRASP principles placed into a SIG

necessary to include as part of the pattern an operationalization called "Direct Access." We qualify in our SIG this aspect of the pattern as detracting from Augmentability, Cohesion, Internal Consistency and Structuredness. On the other hand, by allowing direct access to clients, the pattern also improves Conciseness, External Consistency and Legibility. This is an example of how applying a pattern to a SIG requires additional interpretation, since GRASP principles do not capture all nuances.

6 RELATED WORK

Chung et al (Chung et Supakkul, 2006) propose a pattern-based approach, using UML to capture and reuse functional requirements and a goal-oriented NFR Framework to capture non-functional requirement knowledge. They consider patterns as a collection of model refinement methods, each defining a step in the evolution of elements in the model. Patterns can be composed to form higher-level patterns, or specialized. Although the application of GoF patterns is mentioned briefly, no discussion is dedicated to modeling their impact on NFRs. Our work is related to their approach in that we model GoF patterns as compositions of finer-grained (GRASP) elements.

Tracing non-functional requirements to design patterns was done with SIGs in an approach proposed by Cleland-Huang et al (Cleland-Huang et Schmelzer, 2003). Their work is similar in that they represent design patterns as operationalizations for the purpose of evaluating design decisions. They establish links between non-functional requirements and operationalizations through SIGs. Their work is different from ours, however, in that no guidance is given on how to assess the impact a design pattern has on non-functional requirements. The work proposed by Gross et al (Gross et Yu, 2001) is also related to ours for the same reasons.

Bansiya (Bansiya et Davis, 2002) presented a hierarchical model for assessing design qualities in object-oriented software. It relates design properties such as encapsulation, modularity, coupling, etc., which can be measured to high-level quality attributes such as reusability, flexibility, complexity, etc. The relationships between the properties and quality attributes are weighted, and validated empirically. This contribution is related to our framework in that we seek to model the impact of design-level characteristics on higher-level quality attributes. The obvious difference is that our approach is used to evaluate and record design choices. Our model does not quantify the impact, but relies on the soft-goal approach of the NFR Framework to allow designers to decide priorities.

Bowman et al (Bowman, Briand et Labiche, 2010) propose a multi-objective genetic algorithm (MOGA) that uses class coupling and cohesion measurement to assess the best way to assign responsibilities to classes. This work illustrates that the class responsibility assignment problem is difficult for humans to do, which is why a genetic algorithm is proposed. The algorithm is oriented at coupling and cohesion as primary design qualities. It does not consider the impact on understandability that a responsibility assignment might have.

Our work is related to the softgoal traceability patterns proposed by Fletcher et al (Fletcher et Cleland-Huang, 2006), in that the authors propose a framework to model design patterns as operationalizations. Their work differs from ours in that it seeks to model specific design pattern structures and characteristics, as opposed to general structures we model with GRASP principles. Our framework leverages the simplicity of GRASP to facilitate the mapping.

7 DISCUSSION AND FUTURE WORK

The motivation for developing this framework was to help the designer decide whether the potential benefits of adding a pattern outweigh its drawbacks. Over-use of patterns could lead to designs that are difficult to understand. When used in a SIG, our framework puts into perspective the impact of a given application of a pattern. As we show in the example with Visitor, the understandability of the design suffers as a consequence of the pattern. However, this emphasizes the need for designers to be familiar with GoF and other popular patterns.

Our assessment does not take into consideration design complexity caused by the dynamic aspects of patterns, since GRASP principles do not model these aspects. For example, the participants of the Observer pattern must follow a convention for interacting. The subjects must maintain a list of Observers, and notify them in the proper order whenever certain events occur. GRASP principles do not capture the understandability characteristics of these interactions. Future work seeks to include the dynamics of patterns modeled in UML interaction diagrams, for example.

Because GRASP principles are basic tenets of modular design, our framework can be extended to other patterns beyond those of the GoF. Essentially one needs to identify the GRASP principles present in any pattern to be able to evaluate it. Similarly, if other quality characteristics are important, our framework can be altered to include them. For example, Polymorphism and Indirection definitely have an impact on Performance.

Also, because GRASP principles are used potentially anywhere in a design, our framework could be used to assess any general design decisions that have complexity in the form of the GRASP principles we assessed.

Despite the fundamental nature of GRASP principles, applying them to a SIG can be to some degree arbitrary, as demonstrated in the Visitor example. The challenge seems to be related to scope—some GRASP principles contribute positively to characteristics within a given scope, but detract from them outside. A partial solution might be to include general weights for the impacts, derived and validated using empirical data.

8 CONCLUSIONS

We presented a framework based on assessing important design characteristics of four GRASP principles that are present in complex design structures, namely the design patterns of the GoF. We provide a mapping between GRASP principles and the GoF patterns, to facilitate the use of the framework in common scenarios where a GoF pattern would be applied.

We applied the framework using a SIG to several design patterns, showing the impact these patterns have on the dimension of understandability as well as modifiability of a design. We discussed the flexible nature of the framework, as well as its limitations and ways it could be extended in the future. We believe the results show how our framework could help a designer better understand the impact of complex design.

REFERENCES

Bansiya, J., et C.G. Davis. 2002. « A hierarchical model for object-oriented design quality assessment ». *IEEE Trans on Software Engineering,* vol. 28, n^{o} 1, p. 4-17.

Bansiya, Jagdish. 1997. « A hierarchical model for quality assessment of object oriented designs: a dissertation ». University of Alabama in Huntsville. <http://worldcat.org/oclc/39099325>.

Boehm, B.W., J.R. Brown et M. Lipow. 1976. « Quantitative evaluation of software quality ». In *Proceedings of the 2nd international conference on Software engineering*. p. 592-605. San Francisco, California, United States: IEEE Computer Society Press.

Bowman, Michael, Lionel C. Briand et Yvan Labiche. 2010. « Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms ». *Software Engineering, IEEE Transactions on,* vol. 36, n^{o} 6, p. 817-837.

Chung, L., et S. Supakkul. 2006. « Capturing and Reusing Functional and Non-functional Requirements Knowledge: A Goal-Object Pattern Approach ». In *Information Reuse and Integration, 2006 IEEE International Conference on* (16-18 Sept. 2006). p. 539-544.

Chung, Lawrence. 1999. *Non-functional requirements in software engineering*. Coll. « The Kluwer international series in software engineering ». Boston: Kluwer Academic.

Cleland-Huang, J, et D Schmelzer. 2003. « Dynamically tracing non-functional requirements through design pattern invariants ». In *Workshop on Traceability in Emerging Forms of Software Engineering, in conjunction with IEEE International Conference on Automated Software Engineering*. <http://www.soi.city.ac.uk/~gespan/paper1.pdf>.

Dromey, R. Geoff. 1995. « A Model for Software Product Quality ». *IEEE Trans. Softw. Eng.*, vol. 21, n^{o} 2, p. 146-162.

Fletcher, J., et J. Cleland-Huang. 2006. « Softgoal Traceability Patterns ». In *17th International Symposium on Software Reliability Engineering (ISSRE '06)* (7-10 Nov. 2006). p. 363-374. <http://dx.doi.org/10.1109/ISSRE.2006.42>.

Fowler, Martin, et Kent Beck. 1999. *Refactoring: improving the design of existing code*. Coll. « The Addison-Wesley object technology series ». Reading, MA: Addison-Wesley, xx1, 431 p. p.

Gamma, Erich, Richard Helm, Ralph Johnson et John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Coll. « Addison-Wesley professional computing series ». Reading, Mass.: Addison-Wesley, xv, 395 p. p.

Gross, Daniel, et Eric Yu. 2001. « From Non-Functional Requirements to Design through Patterns ». *Requirements Engineering,* vol. 6, n° 1 (2001/02//), p. 18-36.

Larman, C. 2001. « Protected variation: the importance of being closed ». *Software, IEEE,* vol. 18, n° 3, p. 89-91.

Larman, Craig. 2005. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*, 3rd. Upper Saddle River, N.J.: Prentice Hall PTR, xxv, 702 p. p.

Wendorff, P. 2001. « Assessment of design patterns during software reengineering: lessons learned from a large commercial project ». In *Proceedings of 5th IEEE European Conference on Software Maintenance and Reengineering, 14-16 March 2001* (2001). p. 77-84. Coll. « Proceedings Fifth European Conference on Software Maintenance and Reengineering ». Los Alamitos, CA, USA: IEEE Comput. Soc. <http://dx.doi.org/10.1109/CSMR.2001.914971>.