A Formal Criterion for Comprehensibility of Design

Nadya Kuzmina, John Paul, Ruben Gamboa, and James Caldwell[*]

University of Wyoming, Laramie WY 82070, USA,
{nadya, jpaul, ruben, jlc}@cs.uwyo.edu

Abstract. We believe that *comprehensibility* can be used as a unifying application-oriented criterion for evaluating the quality of structural design. This paper describes a formal apparatus for measuring comprehensibility of refactored design. The main contribution of this paper is a precise and formal definition for three refactoring categories, which are useful, neutral, and obscure. The categories reflect the effect of a refactoring on the comprehensibility of the resulting design. The formalisms in this paper provide the foundation for the development of a tool for measuring comprehensibility of structural design.

1 Introduction

The software community has been concerned about the quality of software design even before the object-oriented paradigm came to life. A lot of different methodologies and guidelines have been proposed on the subject of good design. For example, Kernighan and Pike [6] suggest *simplicity*, *clarity*, and *generality* as the guiding criteria for developing programs. On the other hand, design patterns [4] and refactoring [3] provide "recipes" that developers can use to produce good designs. However, once we have a design, is it possible to measure its quality in an objective way? Is it possible to compare designs for being "better" or "worse" automatically? We address these questions in more detail in our other paper [7], while this paper concentrates on the formalisms which allow us to evaluate designs resulting from a refactoring.

Our main thesis, called the Comprehensibility Thesis, proposes *comprehensibility* as a unifying application-oriented criterion for evaluating the quality of structural design [7]. By structural design we mean the structure of a program's source code. We defined an (almost) objective measure for comprehensibility based on automatically detected program constraints which is best suited for designs that result from refactorings.

The suggested approach compares the *initial* specification according to which the application was developed and the *recovered* specification generated by an automated constraint detector via program analysis of the application. The amount of the initial specification which can be inferred from the recovered specification

[*] This material is based upon work supported by the National Science Foundation under Grant No. NSF CNS-0613919.

serves as the measure of comprehensibility of the application's design[1]. If there are two designs implementing the same specification, then the design with higher measure for comprehensibility is considered "better".

This paper concentrates on the Refactoring Thesis which states how different categories of refactorings affect the comprehensibility of the resulting design. The main contribution of this paper is a precise and formal definition for three refactoring categories: *useful*, *neutral*, and *obscure*. According to the Refactoring Thesis, useful, neutral, or obscure refactorings either respectively increase, do not change, or decrease the comprehensibility of the resulting design. The formalisms developed in this paper provide a foundation for developing a semi-automatic tool which may be used to evaluate refactorings [7].

We formalize *constraint languages* and develop *labeled theories* to define the effect of a refactoring on the recovered specification in Sect. 5. Our formalisms are based on graph transformations proposed by Mens et al. [9] for formalizing refactorings.

The rest of this paper is organized as follows. The next section provides a short introduction to the Comprehensibility Thesis. Section 3 presents a working example which is used throughout the paper. The necessary background on graph transformations is described in Sect. 4. Section 5 first introduces the formalisms that the Refactoring Thesis builds upon and then states the Refactoring Thesis. Tool support for the Refactoring Thesis is discussed in Sect. 6. Section 7 works through a detailed example of an application of the Refactoring Thesis. An overview of the related work is given in Sect. 8. Finally, Sect. 9 concludes and briefly discusses our future work.

2 Our Motivation

In this section we explain that our motivation for formalizing refactorings is to be able to define the comprehensibility of a design in an objective way and to apply this measure to the task of comparing refactored designs.

We propose comprehensibility as a single criterion that characterizes good structural design under the assumption that the program at hand is correct with respect to its specification. A new developer should be able to understand a comprehensible program. Comprehensibility, as it is commonly understood, is a subjective notion and has been studied empirically in the psychology of programming [11]. The Comprehensibility Thesis allows us to express comprehensibility in terms of an automatically recovered specification and to define a more objective quantitative measure of comprehensibility.

Comprehensibility Thesis. *A good design is comprehensible to an automated tool, which means that an automated tool is able to recover nearly complete specifications from good designs.*

Constraint detectors are tools that attempt to recover a program's specification in the form of constraints by analyzing a program or its behavior. How

[1] We refer to *structural design* as simply design when not explicitly stated.

much a tool is able to learn about the underlying program in comparison to the initial specification comprises the objective measure of comprehensibility. A *Recovered specification* consists of the facts that a constraint detector learns about the underlying program, while the *initial specification* is the specification used to develop the program in the first place. Informally, we claim that the more facts a constraint detector is able to learn about the underlying program, the better is its structural design. In essence, the objective notion of comprehensibility is modeled by how much of the initial specification a constraint detector is able to recover.

The *closer* the recovered specification is to the initial specification, the more comprehensible the corresponding design is. To define the notion of closeness of one specification to another formally we can use logical implication: the more of the initial specification that is logically implied by the recovered specification, the closer the recovered specification is to the initial one. Thus when two designs exist for the same specification, this notion of comprehensibility can be used to compare them to each other. One context in which multiple designs for the same specification come into play is that of software refactoring.

According to Fowler [3], a *refactoring* is a process of changing a software system that does not alter its external behavior yet improves its internal structure. Since a refactoring necessarily results in an altered design which conforms to the same specification as the initial one, this allows us to directly compare their respective comprehensibility measures. For the remainder of this paper we will be studying comprehensibility as the comparative measure of two designs belonging to the same sequence of refactorings.

3 A Motivating Example

The *CartEntry* example from the *Replace Temp with Query* refactoring [3] will serve as an illustration for our approach throughout this paper. While keeping the codebase manageable this example will allow us to illustrate the effects of refactoring on comprehensibility in Sect. 7.

The initial design consists of a class *CartEntry* with method `getPrice` which calculates the price for an entry in a shopping cart as shown in the left pane of Fig. 1. The *Replace Temp with Query* refactoring extracts the computations for the base price and the discount factor into separate methods. This design of the class is presented in the right pane of Fig. 1.

4 Background: Formalizing Refactorings with Graph Transformations

Mens et al. [9] describe a lightweight formal method for considering program refactoring. The basis of this approach is the representation of object oriented programs as labeled graphs and program refactorings as sequences of graph rewrites. This section explains briefly how the graph formalism works. In Sect. 5 we will adopt this formalism to state our Refactoring Thesis.

```
public class CartEntry{
int quantity;
int itemPrice;
double getPrice(){
  int basePrice = quantity * itemPrice;
  double discountFactor;
  if (basePrice > 1000)
    discountFactor = 0.95;
  else
    discountFactor = 0.98;
  return basePrice * discountFactor;
} }
```

Initial Design D_1

```
public class CartEntry{
int quantity;
int itemPrice;
int basePrice(){
  return quntity * itemPrice;
}
double discountFactor(){
  if (basePrice() > 1000)
    return 0.95;
  else
    return 0.98;
}
double getPrice(){
  return basePrice() * discountFactor();
} }
```

Refactored Design D_2

Fig. 1. Initial design D_1 and refactored design D_2 of the *CartEntry* class.

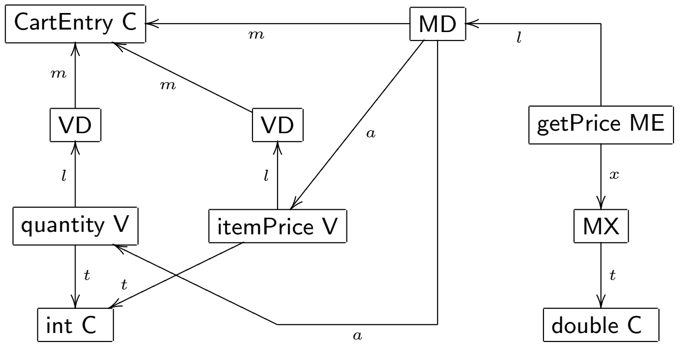

Fig. 2. Method and Field Definitions in the initial class *CartEntry*.

Definitions. *Let Σ be a set of node labels and Δ a set of edge labels. A program graph over Σ and Δ is a 3-tuple $G = (V_G, E_G, nlab_G)$, where V_G is the set of nodes, $nlab_G : V_G \to \Sigma$ is the node labeling function and $E_G \subseteq V_G \times \Delta \times V_G$ is the set of edges.*

In a program graph, language entities such as classes, variables, methods, method parameters, etc., are represented by nodes whose label is a pair consisting of a name and a node type.

For example, the *CartEntry* class may be represented by a node u with name *CartEntry* and type C (i.e., class). This is denoted as $name(u) = CartEntry$ and $u \in C$. Figure 2 presents the program graph for the initial design of the *CartEntry* class, and Fig. 3 depicts the graph for the refactored design. The set $\Sigma = \{C, ME, MX, MD, V, VD, P, E\}$ of all possible node types is described in Fig. 4. Intuitively, each type in this collection corresponds to a particular type of syntax fragment in the program. For instance, nodes $u \in C$ represent classes,

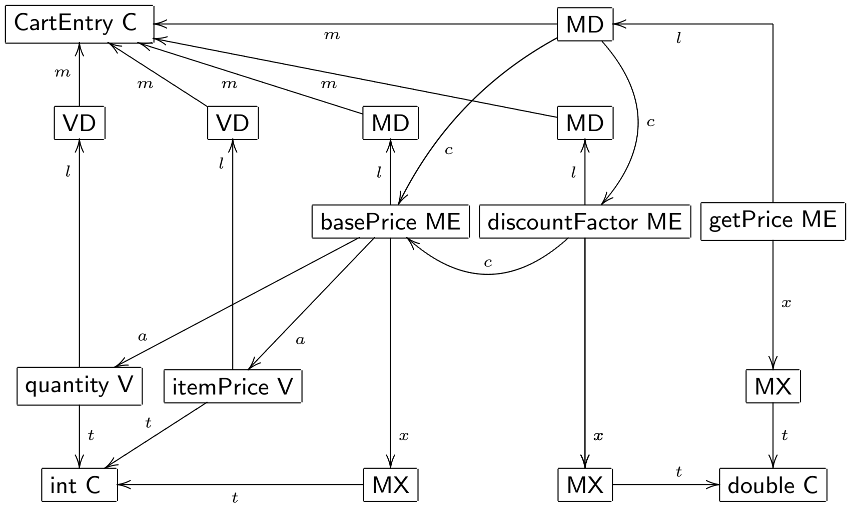

Fig. 3. Method and Field Definitions in class *CartEntry* after Refactoring.

nodes $v \in ME$ represent method invocations[2], and nodes $w \in MD$ represent method definitions. Methods are represented by a combination of method invocations (*ME*-nodes), method definitions (*MD*-nodes) and method exits (*MX*-nodes). The reason that *MD*-nodes are separate from *ME*-nodes is to allow the graph to capture those situations when the same method invocation may refer to different method definitions due to late binding and dynamic method lookup.

In what follows we will simply treat Σ as a partition of the nodes belonging to a design D into disjoint node sets, some of which may be empty.

Relationships such as membership, inheritance, method lookup, variable accesses, method calls, etc., that hold between software entities are represented by *edges* between the corresponding nodes. For example, the dynamic lookup for the definition of method `getPrice` is represented by an edge of type l between the *getPrice* *ME*-node and the corresponding *MD*-node in Fig. 2. If a method takes n parameters, the corresponding *ME*-node has n outgoing edges with labels $1.p$ through $n.p$. The set $\Delta = \{l, i, m, t, e, c, a, u, x\} \cup \{1.p, 2.p, \ldots\}$ consists of all possible edge-types, which are described in figure 4.

In what follows we are simply going to treat Δ as a collection of binary predicates defined over the particular node sets Σ of a design D. For instance, when $u \in C$ and $v \in MD$, $\langle u, v\rangle \in m$ means that class u defines method v. Also, for each $j > 0$, predicate $j.p$ captures the relationship that each method invocation $u \in ME$ has to the type $v \in C$ of its jth parameter. So, there are only finitely many nonempty relations of this kind.

[2] We introduce one modification to the the theory of Mens et al. [9]. We replaced the single method node type M with two nodes types: method entry *ME* and method exit *MX* and link them with an x (i.e., exit) edge.

Node Type	*Desription*	*Examples*
C	**C**lass	*CartEntry*
ME	**M**ethod **E**ntry	entry to *getPrice*
MX	**M**ethod e**X**it	exit from *getPrice*
MD	**M**ethod **D**efinition	
V	**V**ariable	*quantity, itemPrice*
VD	**V**ariable **D**efinition	`int quantity;`
P	**P**arameter of method definition	
E	**E**xpression in method definition	
Edge Type	*Desription*	*Examples*
l: $ME \rightarrow MD$	dynamic method **l**ookup	
$V \rightarrow VD$	variable **l**ookup	`int quantity;`
i: $C \rightarrow C$	**i**nheritance	
m: $VD \rightarrow C$	variable **m**embership	variable `quantity` is in *CartEntry*
$MD \rightarrow C$	method **m**embership	method *getPrice* is in *CartEntry*
t: $V \rightarrow C$	variable **t**ype	**`int`** `quantity`
$MX \rightarrow C$	method return **t**ype	**`double`** `getPrice`
p: $ME \rightarrow C$	**p**arameter definition and type	
e: $MD \rightarrow E$	**e**xpression in method definition	
$E \rightarrow E$	sub**e**xpression in method definition	
c: $E \rightarrow ME$	(dynamic) method **c**all	`basePrice()`
a: $E \rightarrow \{V—P\}$	variable or parameter **a**ccess	`quantity`
u: $E \rightarrow \{V—P\}$	variable or parameter **u**pdate	
x: $ME \rightarrow MX$	method e**x**it	

Fig. 4. Set $\Sigma = \{C, ME, MX, MD, V, VD, P, E\}$ of all possible node types and set $\Delta = \{l, i, m, t, e, c, a, u, x\} \cup \{1.p, 2.p, \ldots\}$ of all possible edge types.

An entire system involving multiple classes can be represented as a single program graph using this formalism. Although it currently does not account for Java modifiers such as static, abstract, protected, final, etc., one can introduce these modifiers into the graph formalism by attaching the appropriate attributes to nodes of type C, MD and VD.

Refactorings are then formalized as graph transformation rules with the use of graph rewriting systems [9]. For instance, the *Replace Temp with Query* refactoring would be formalized as a system of rewrite rules that will transform the graph shown in Fig. 2 into the graph in Fig. 3.

5 Introducing the Refactoring Thesis

In this section we introduce the formalism which will be used to compare the effects that different refactorings have on determining the comprehensibility of a recovered specification. Based on their effects, we categorize refactorings into three groups: *useful*, *neutral* and *obscure*. The Refactoring Thesis relates the refactorings in each group to the changes in comprehensibility of the resulting design. Whereas a neutral refactoring leaves comprehensibility unchanged, and a useful refactoring actually improves the overall comprehensibility, an obscure refactoring leads to a loss of comprehensibility.

A recovered specification is a set of program points and a mapping which associates each program point to a set of constraints. Each constraint in one of these sets is stated in terms of the program variables visible at the program point where the constraint is originally specified. First we introduce a many-sorted first-order language to formalize the constraint language at a program point in

Sect. 5.1. Then we develop labeled theories and state precisely the Refactoring Thesis in Sect. 5.2.

5.1 Labeled Constraint Languages

The set of *program points* or *labels* for a design D consists of all its method entry, method exit, and class nodes as identified in its program graph according to the previous section. Constraints that are collected at these three types of points correspond to object invariants, method preconditions and method postconditions, respectively. Formally we define the labels to be the set $\mathcal{A}(D) = ME \cup MX \cup C$. In this section we introduce a many-sorted first-order language [5] at a label $u \in \mathcal{A}(D)$ called the *constraint language* for label u.

This step is necessary since the type of a program variable determines the kinds of constraints in which it can participate. For example, $x > 0$ may be a constraint formed for an integer-valued variable x, while $x \neq \mathit{null}$ may only be formed when x is array-valued. Each class $u \in C$ in a design D can be thought of as a design-specific type. Together with the primitive types used in the program (which are also depicted as C nodes) the design-specific types define a set of *sorts*.

This set of sorts S is defined by $S = \{\mathit{sort}(u) | u \in C\}$ where *sort* maps each class $u \in C$ to its unique sort.

Constraints are generally formed on the fields of a class $u \in C$ whenever these fields are visible at a particular program point $v \in \mathcal{A}(D)$. The fields of a class $u \in C$ consist not only of the fields defined in u but potentially the fields defined in any class from which u inherits as well. To formalize this fact we first introduce the transitive closure i^* of the inheritance relationship $i \in \Delta$. The function $\mathcal{I} : C \rightarrow 2^C$ is defined for each class $u \in C$ to be the set consisting of u and all its ancestors. That is, $\mathcal{I}(u) = \{u\} \cup \{v | \langle u, v\rangle \in i^*\}$. Next we introduce all the fields of a particular type $u' \in C$ that are accessible within a class $u \in C$ as a set of constant symbols of the corresponding sort $\mathit{sort}(u') \in S$.

$$\begin{aligned}\mathcal{F}_s(u) = \bigcup_{u'' \in \mathcal{I}(u)} \{w |\ w \in V \ \wedge\ & (\exists v \in VD)(\exists u' \in C) \\ & (\langle v, u''\rangle \in m \ \wedge\ \langle w, v\rangle \in l \ \wedge \\ & \ \langle w, u'\rangle \in t \ \wedge\ \mathit{sort}(u') = s)\}\end{aligned}$$

So, for instance, if u is the *CartEntry* node in Fig. 2, then $\mathcal{F}_{\underline{int}}(u) = \{\mathit{quantity}, \mathit{itemPrice}\}$ and $\mathcal{F}_{\underline{bool}}(u) = \{\}$ [3].

As a preliminary to capturing all the fields that are possibly accessible at a method entry $u \in ME$ we specify the classes containing these fields. This is done by first identifying all the potential method definitions for the invocation and then including the containing classes and their ancestors. Similarly the compile-time type of each parameter and its ancestors must also be included.

$$\begin{aligned}\mathcal{C}(u) = \{v' |\ & (\exists w \in MD)(\exists v \in C)(\langle u, w\rangle \in l \ \wedge\ \langle w, v\rangle \in m \ \wedge\ v' \in \mathcal{I}(v)) \ \vee \\ & (\exists v \in C)(\exists n \in I\!N)(\langle u, v\rangle \in n.p \ \wedge\ v' \in \mathcal{I}(v))\}\end{aligned}$$

[3] Concrete sorts are underlined.

It is now possible to specify the program variables that are visible at each type of program point. These will be introduced as S-sorted constant symbols. For $u \in C$ only the fields visible in the class are considered. For $u \in ME$ we consider not only the class fields that are visible in either defining classes *or* parameter classes, but also the parameters themselves. For $u \in MX$ two tagged copies of the variables available at the corresponding method entry are considered along with a special *return* symbol [4] if the method's return type is non-void. The copy tagged *pre* represents the program state at method entry, while the copy tagged *post* represents the same variables at method exit.

The following equations specify the set of program variables $\mathcal{V}_s(u)$ of sort $s \in S$ that are possibly accessible at a program point $u \in \mathcal{A}(D)$ in a design D.

$$\begin{aligned}
\mathcal{V}_s(u) &= \mathcal{F}_s(u), \text{ if } u \in C \\
\mathcal{V}_s(u) &= \bigcup_{v \in \mathcal{C}(u)} \mathcal{F}_s(v) \ \cup \ \{p_n \mid (\exists n \in I\!N)(\exists v \in C) \\
&\qquad\qquad\qquad (\langle u, v\rangle \in n.p \ \wedge \ \mathit{sort}(v) = s)\}, \text{ if } u \in ME \\
\mathcal{V}_s(u) &= \{c \mid \ (\exists v \in ME)(\langle v, u\rangle \in x \ \wedge \ c \in \mathcal{V}_s(v) \times \{\mathit{pre}, \ \mathit{post}\}) \ \vee \\
&\qquad (\exists v \in C)(\langle v, u\rangle \in t \ \wedge \ \mathit{sort}(v) = s \ \wedge \ c = \mathit{return}(u))\}, \text{ if } u \in MX
\end{aligned}$$

So, for instance, if $u \in MX$ in Fig. 2, then $\mathcal{V}_{\underline{int}}(u) = \{\mathit{quantity}, \mathit{itemPrice}\}$, while $\mathcal{V}_{\underline{double}}(u) = \{\mathit{getPrice}\}$.

Having introduced sorted constants for the program variables visible at a label $u \in \mathcal{A}(D)$ it is now possible to define the *constraint language* $\mathcal{L}(u)$. Informally, a constraint language is a set of boolean formulas over the program variables visible at a label.

The set of sorts $\overline{S} = S \cup \{\underline{int}, \underline{bool}, \ldots\}$ used to define the language consists of the sorts obtained from the design as well as a handful of primitive sorts. As mentioned before, the type of a program variable determines the constraints that it can participate in. Types are represented by sorts in our formalism and constraints are represented by $\underline{bool}$ sorted ground terms.

The constant symbols used in constraint construction at a label $u \in \mathcal{A}(D)$ will include all the visible program variables of each sort as well as some predefined constants. For $s \in \overline{S}$, the set of constant symbols $\mathbf{CS}_s(u)$ of sort s consists of $\mathcal{V}_s(u)$ as well as the symbol *null* if s corresponds to a reference type in D. The sorts $\underline{int}$ and $\underline{bool}$ additionally have constants $\{\ldots -1, 0, 1, \ldots\}$, *true*, *false*, etc., defined for them.

Similarly, sets of $\overline{S}$-sorted function symbols define the operations used in constraint construction. For this purpose, a set of function symbols $\mathbf{FS}_{\theta,s}$ is defined for each non-empty signature $\theta \in \overline{S}^+$ and sort $s \in \overline{S}$. So, for instance, $+ \in \mathbf{FS}_{\underline{int}\,\underline{int},\underline{int}}$ and $= \in \mathbf{FS}_{s\,s,\underline{bool}}$ for any $s \in \overline{S}$. Also, $\{\leq, \geq\} \subset \mathbf{FS}_{\underline{int}\,\underline{int},\underline{bool}}$.

In addition to the constant and function symbols mentioned so far, new symbols may also be introduced that correspond to some of the methods defined in the design itself. The only restriction on the correctness of the resulting interpretation is that the methods thus included are indeed side-effect free. Thus, the

[4] For an n-place method f, this return symbol consists of the term $f(p_1, \ldots, p_n)$, or simply f if $n = 0$.

constants[5] *basePrice* and *discountFactor* can be included in the constraint language defined at method exit of *getPrice* in the refactored design of *CartEntry* (Fig. 3).

Together the constant and function symbols generate a set of well-sorted terms $\mathbf{T}_s$ for each $s \in \overline{S}$. The constraint language $\mathcal{L}(u)$ is then defined inductively in the following way.

$$\mathcal{L}(u) ::= t \in \mathbf{T}_{\underline{bool}} | \neg\phi | \phi \wedge \psi | \phi \rightarrow \psi | \ldots$$

In other words, $\mathcal{L}(u)$ consists of all terms of sort $\underline{bool}$ as well as more complex formulas built from these using the logical connectives.

5.2 Formalizing Recovered Specifications as Labeled Theories

The previous section formalized the constraint language at a program point. Here we define *labeled theories* to formalize recovered specifications as a whole.

In this section we assume the existence of a *domain theory* that contains axioms describing arithmetic and other basic operations used to specify constraints. In addition the domain theory should contain object invariants for classes in the design as well as axioms describing the semantics of design specific symbols.

A *labeled theory*, $T_u(D)$, for label $u \in \mathcal{A}(D)$ of a design D is a set of constraints at program point u closed under deduction with respect to the domain theory. That is, $T_u(D) = \{C \,|\, C\, \textit{is a constraint in the language}\; \mathcal{L}_D(u)\}^*$, where the symbol '*' denotes deductive closure with respect to the domain theory. Although the ultimate choice of domain theory is design specific, it may contain decidable subtheories like Presburger arithmetic.

We introduce the notion of a *theory extension* to represent the transformation of a recovered specification induced by a refactoring. A theory $T(D_2)$ *extends* a theory $T(D_1)$ if and only if for each label $u \in \mathcal{A}(D_1) \cap \mathcal{A}(D_2)$, $T_u(D_1) \subseteq T_u(D_2)$. For example, let $\mathcal{A}(D_1) = \mathcal{A}(D_2) = \{v\}$, $T_v(D_1) = \{x \geq 0\}^*$, and $T_v(D_2) = \{x == y^2\}^*$. Then $T(D_2)$ extends $T(D_1)$ since the constraints in $T_v(D_1)$ follow from the constraints in $T_v(D_2)$.

D_2 *non-conservatively extends* D_1 if $T(D_2)$ extends $T(D_1)$ and there exist a label $u \in \mathcal{A}(D_1) \cap \mathcal{A}(D_2)$ and a formula $\phi \in T_u(D_2) - T_u(D_1)$ such that ϕ is in the language of D_1 (i.e., $\phi \in \mathcal{L}_{D_1}(u)$). A non-conservative extension ensures that the theory of design D_2 contains new facts about the initial variables in D_1. On the other hand D_2 is a *neutral extension* of D_1 if $T(D_2)$ extends $T(D_1)$ and for every label $u \in \mathcal{A}(D_1) \cap \mathcal{A}(D_2)$ and every formula $\phi \in \mathcal{L}_{D_1}(u)$, $\phi \in T_u(D_1) \Longrightarrow \phi \in T_u(D_2)$. Neutral extensions represent refactorings that do not result in the loss of any facts over initial variables. The possible relationships that may exist between an initial and a refactored design determine three kinds of refactorings:

Definitions. *Let design D_{i+1} be the result of the refactoring R_i applied to design D_i. The refactoring R_i is useful if D_{i+1} is a non-conservative extension*

[5] Zero place methods are introduced as suitably sorted constant symbols.

of D_i. The refactoring R_i is neutral if D_{i+1} is a neutral extension of D_i. The refactoring R_i is obscure if D_{i+1} is not an extension of D_i.

Intuitively, a useful refactoring exposes some previously unknown facts about variable relationships in the initial design and, therefore, increases the comprehensibility of the resulting design. Neutral refactorings preserve the constraints known beforehand and require further analysis. An obscure refactoring results in the loss of constraints known prior to the refactoring. Obscure refactorings decrease the comprehensibility of the resulting design.

Refactoring Thesis. *Useful refactorings improve the comprehensibility of design with respect to its specification. Neutral refactorings do not affect the comprehensibility of design with respect to its specification. Obscure refactorings reduce the comprehensibility of design with respect to its specification.*

The Refactoring Thesis is an application of the Comprehensibility Thesis to refactorings. It directly implies the clarity principle of Kernighan and Pike [6]. The simplicity criterion may be used to judge the effect of neutral refactorings on the overall structure of the resulting design.

A refactoring may be useful, neutral or obscure independently of whether it expands or contracts the structure of a design. If a structure-expanding refactoring, such as *Extract Method*, *Replace Method with Method Object*, and *Replace Array with Object*, elicits new facts in the resulting design, then it increases the comprehensibility and will be classified as useful by our definition. Other structure-expanding refactorings, such as extract a `print` method for example, may preserve the relations known in the initial design, in which case they are neutral. If a structure-contracting refactoring, such as *Inline Method*, for example, preserves all the facts known in the initial design, it is also neutral. However, a neutral structure-contracting refactoring simplifies the structure of a design without reducing its comprehensibility. In this case we can use simplicity to identify such a refactoring as desirable from a practical standpoint. If any of the refactorings described above obscures a constraint, then the comprehensibility of the design is diminished, and the refactoring is not recommended. We provide a detailed example of the application of the Refactoring Thesis in Sect. 7. As a final remark, the Refactoring Thesis does not consider the transformations induced by a refactoring on the *structure* of the design, rather it is concerned only with the effects a refactoring may have on knowledge about the original design at preserved program points.

5.3 Sequences of Primitive Refactorings

Refactorings are commonly carried out as sequences of primitive refactorings where primitive refactorings are the ones catalogued by Fowler et al. [3]. The Refactoring Thesis refers to primitive refactorings, while this section addresses the comprehensibility of sequences of primitive refactorings.

A *composite* refactoring is a sequence of primitive refactorings. Each primitive refactoring in this sequence can be classified as useful, neutral or obscure. Neutral refactorings may be used in preparation for a useful refactoring. Therefore, a subsequence of neutral refactorings followed by a useful refactoring increases the comprehensibility of the resulting design and is desirable from a practical standpoint. It is possible for a subsequence of neutral refactorings to occur at the end of a composite refactoring sequence. These neutral refactorings do not increase the comprehensibility and should be well-justified by some other criteria, such as simplicity for example. We consider any sequence containing obscure refactorings to be obscure.

6 Tool Support for the Refactoring Thesis

The previous section is purely theoretical and abstracted from any particular constraint detector tool which may be used to infer recovered specifications. From a practical standpoint we need to consider an adequate automatic tool which can be used to produce recovered specifications.

From the perspective of a constraint detector tool the Refactoring Thesis means the following. Informally the thesis says that if a refactoring results in a design that enables the constraint detector to infer new facts about the variables in the initial design, then this refactoring increases the comprehensibility of the code. New facts refine the recovered specification, making it more precise and closer to the initial specification for the system, which increases the comprehensibility measure. Neutral refactorings do not affect the comprehensibility with respect to the constraint detector, and therefore require further analysis. An obscure refactoring prevents the tool from inferring relationships that it was able to discover in the initial design and therefore decreases comprehensibility.

Since we model human comprehension with that of a constraint detector tool our future work is concerned with the development of such a tool. We used Daikon [1, 12] as a constraint detector in our preliminary experiments [7]. Daikon is based on dynamic analysis which infers likely constraints from variable values observed during program executions. A likely constraint is a constraint that has not been falsified on any of the observed program runs. A fixed grammar of properties determines the considered relationships between program variables.

Some of the problems we encountered are the limitations of dynamic constraint detection, such as "noisy" recovered specifications and overly generalized constraints that were reported. We also discovered that some of the constraints were missing because Daikon does not support the propagation of constraints from a method to the callers of this method.

"Noisy" recovered specifications refer to the output which is cluttered with the relationships which held during the examined program runs but are not interesting to a human developer. Such "noise" is due to the algorithm which considers all relationships defined in the grammar of properties to be true at the beginning. Overly generalized constraints, such as $x > 0$ instead of $x = y^2 + 2$, result from the fact that the grammar of properties is missing more precise properties used in the program. A possible fix is to add the precise property templates into the grammar of properties. However, it is very tedious to do so for every new desirable property. Also, such a fix will result in “noisier” output since the newly added property may be reported in other places in the program where a human developer may not be interested in it.

The propagation of constraints from a method to its callers can be vital to defining a precise constraint on the caller. For example, if a method `double f(int x, int t)` is implemented as `0.5 * power(x, t) + t` we need to propagate the constraint $power(x, t) = x^t$, which holds on the return of the `power` method, in order to infer that the constraint on the return of `f` is $f(x, t) = 0.5 * x^t + t$.

We believe that the drawbacks of dynamic analysis for constraint detection can be fixed by combining static and dynamic analyses. The tool we are developing will attempt to infer constraints in the Object Constraint Language [15] using such a hybrid approach. The static analysis will extract the relationships between variables from source code while the dynamic component will examine the values of program variables during execution to infer constraints in places where the static analysis does not perform well, such as method exits where the implementation uses loops. Static analysis will reduce the amount of “noisy” constraints and overly generalized constraints reported by using the relationships defined in the source code. The tool will also be capable of propagating constraints from a method to its callers.

In practice, a human developer should be able to evaluate a refactoring by comparing the recovered specifications for the initial and resulting designs and classifying the refactoring. If the refactoring is useful, the developer can be assured to keep it, if it is obscure, a roll back to the initial design would be preferable. If it is neutral, the developer may be guided by some other criteria, such as simplicity or extensibility, which is not taken into account by the comprehensibility of the recovered specification. In our future work we are going to investigate the possibilities of (semi)automatically comparing constraint-based specifications using an inference engine.

The limitations of our approach in practice are largely due to the sensitivity of particular constraint detectors and are described in more detail in our other paper [7].

7 A Complete Example

In this section we compare the comprehensibility of the original design D_1 of the *CartEntry* to that of the design D_2 that results after applying the *Replace Temp with Query* refactoring to D_1. The sourcecode for both designs is given in Fig. 1.

The original specification of the `getPrice` method's postcondition is captured by the following expression.

$$\begin{array}{l} \text{if } \mathit{quantity} * \mathit{itemPrice} > 1000 \\ \text{then } \mathit{getPrice} = \mathit{quantity} * \mathit{itemPrice} * 0.95 \\ \text{else } \mathit{getPrice} = \mathit{quantity} * \mathit{itemPrice} * 0.98 \end{array}$$

A symbolic execution of D_1 may reveal this directly, but a dynamic constraint detector that conjectures postconditions by inspecting execution traces most likely will not. Symbolic execution is an expensive operation for humans and machines alike, and although much may be learned in the endeavor, its drawback is that it does not differentiate finely enough between well and poorly designed programs. In fact, when it is undertaken by humans it is often a last resort in attempting to understand a poorly documented or designed program [8]. Therefore, what may be learned about a program by a dynamic constraint detector may actually be a better indicator of the quality of its design.

Thus we will limit ourselves to what a dynamic detector can predict about the *getPrice* postcondition in both designs. In the case of D_1 this is precious little; in fact all we are likely to know at method exit is that the return value is greater than or equal to zero. Using the formal apparatus of Sect. 5.2 this may be written as follows.

$$T_{D_1}(\mathit{getPrice\_exit}) = \{\mathit{getPrice} \geq 0\}^*$$

As before, the '*' operator represents the deductive closure of the enclosed constraint(s) in the presence of some suitably defined domain theory. Even with closure, however, the facts in this labeled theory are far from capturing the original specification.

Turning our attention to D_2, we list the labeled theories for the three method exits of that design.

$$\begin{aligned} T_{D_2}(\mathit{getPrice\_exit}) &= \{\mathit{getPrice} = \mathit{basePrice} * \mathit{discountFactor}\}^* \\ T_{D_2}(\mathit{basePrice\_exit}) &= \{\mathit{basePrice} = \mathit{quantity} * \mathit{itemPrice}\}^* \\ T_{D_2}(\mathit{discountFactor\_exit}) &= \{\mathit{discountFactor} = 0.95 \ \vee \ \mathit{discountFactor} = 0.98\}^* \end{aligned}$$

Since it contains the new symbols *basePrice* and *discountFactor*, the main constraint of $T_{D_2}(\mathit{getPrice\_exit})$ is not entirely in the language of the original design. Deductive closure of this formula is, however, taken in the presence of the following two formulas, which relate the preconditions[6] of *basePrice* and *discountFactor* to their respective postconditions.

$$\begin{aligned} &\mathit{true} \Longrightarrow \mathit{basePrice} = \mathit{quantity} * \mathit{itemPrice} \\ &\mathit{true} \Longrightarrow \ \mathit{discountFactor} = 0.95 \ \vee \ \mathit{discountFactor} = 0.98 \end{aligned}$$

Therefore $T_{D_2}(\mathit{getPrice\_exit})$ actually contains the following formula, which *is* in the language of the original design.

$$(\mathit{getPrice} = \mathit{quantity} * \mathit{itemPrice} * 0.95) \ \vee \ (\mathit{getPrice} = \mathit{quantity} * \mathit{itemPrice} * 0.98)$$

Since this is a new fact, the application of the *Replace Temp with Query* refactoring to D_1 is classified *useful*. Since the refactoring also *is* useful in the commonly understood way, the present example appears to bolster the Refactoring Thesis.

[6] Since the methods in D_1 and D_2 don't take any parameters, their preconditions are simply *true*.

8 Related Work

Other researchers have considered various approaches to formalizing object-oriented programs, design patterns and refactorings.

A number of techniques have concentrated on identifying common "building blocks" of object-oriented systems or patterns which were formalized in different ways. Mikkonen [10] formalized temporal behavior of design patterns in terms of high-level abstractions of communication based on Temporal Logic of Actions. Eden [2] developed a higher-order formal language for describing and reasoning about object-oriented systems called LePUS (Language for Patterns Uniform Specification). Smith and Stotts [14] define Elemental Design Patterns (EDPs), the "building blocks" for design patterns which are then formalized with the *rho calculus*, a version of the sigma calculus augmented with the ability to encode relationships. Primitive refactorings defined by Fowler [3] serve as the "building blocks" for refactorings in the graph rewriting formalism of Mens et al. [9]. We chose the last formalism as the basis for our work because the graph formalism provides a good foundation for the constraint languages.

A variety of tools has been developed to support different levels of refactoring. For example, Rajesh and Janakiram [13] propose a tool, called JIAD (Java based Intent-Aspects Detector) for refactoring using design patterns. JIAD automatically infers the code that needs to be refactored and the appropriate design patterns that need to be applied. The goals of our work are different from those of performing refactorings. We are using refactorings as a fruitful ground for evaluating the quality of structural design of a program. One of the lines of our future research is to integrate our constraint detector tool with an automated refactoring tool to evaluate the quality of designs produced by the refactoring tool.

9 Conclusions and Future Work

We presented a formal approach to measuring the comprehensibility of refactored design which relies on automatically recovered specifications. We formally defined three categories which correspond to the effect of a refactoring on the comprehensibility of the resulting design. The three categories are useful, neutral, and obscure, where the refactorings in each category respectively increase, do not change, and decrease the comprehensibility of the refactored design. The formal abstractions described in this paper provide the basis for the development of a tool capable of recovering adequate specifications for object-oriented programs.

In practice our approach relies on a constraint detector tool which can be used to automatically infer recovered specifications. The limitations of dynamic constraint detection determined our choice of hybrid program analysis. Our tool will use a combination of static source code analysis and runtime behavioral analysis to infer constraints in OCL [15]. Our future work also includes investigating the possibilities of automatically measuring the quality of the recovered specification by comparing it to the initial specification in the form of constraints.

References

[1] Daikon invariant detector. *http://pag.csail.mit.edu/daikon*.

[2] A. H. Eden. Formal specification of object oriented design. In *Int'l Conf. on Multidisciplinary Design in Engineering CSME-MDE 2001*, 2001.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[5] Jean H. Gallier. *Logic for Computer Science*, chapter 10. Harper & Row, Publishers, Inc., 1986.

[6] B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[7] N. Kuzmina, R. Gamboa, J. Caldwell, and J. Paul. Comprehensibility as a measure for structural design. Submitted to FASE'07: Fundamental Approaches to Software Engineering (under review), available at *http://www.cs.uwyo.edu/∼nadya*, 2007.

[8] S. Letovsky. Cognitive models in program comprehension. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*, pages 58–79. First Workshop on Empirical Studies of Programmers, 1986.

[9] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, July/August 2005.

[10] T. Mikkonen. Formalizing design patterns. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 115–124, Washington, DC, USA, 1998. IEEE Computer Society.

[11] A. Murray and T. C. Lethbridge. On generating cognitive patterns of software comprehension. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 200–211. IBM Press, 2005.

[12] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, Newport Beach, CA, USA, November 2–4, 2004.

[13] J. Rajesh and D. Janakiram. JIAD: a tool to infer design patterns in refactoring. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 227–237, 2004.

[14] J. McC. Smith and D. Stotts. Elemental design patterns: A formal semantics for composition of oo software architecture. *27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, page 183, 2002.

[15] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.