Using Declarative Meta Programming for Design Flaws Detection in Object-Oriented Software

Sakorn Mekruksavanich and Pornsiri Muenchaisri
Center of Excellence in Software Engineering
Department of Computer Engineering, Faculty of Engineering
Chulalongkorn University, Bangkok 10330 Thailand
Sakorn.M@Student.chula.ac.th, Pornsiri.Mu@chula.ac.th

Abstract

Nowadays, many software developers and maintainers encounter with incomprehensible, unexpandable and unchangeable program structures that consequently reduce software quality. Such problems come from poor design and poor programming called design flaws. Design flaws are program properties that indicate a potentially deficient design of a software system. It can increase the software maintenance cost drastically. Therefore detection of these flaws is necessary. This paper proposes a declarative-based approach in which the design flaws of an object-oriented system can be detected at the meta-level in the declarative meta programming. We apply our approach to detect some well-known design flaws, and the results show that the proposed approach is able to detect those flaws.

1 Introduction

Object-oriented (OO) paradigm is proposed to be one of the most contemporary adaptable frameworks for developing systems. It promotes reusability, maintainability, extensibility and scalability of the systems. However, in practice, only experienced developers can fully achieve advantages of the object-oriented paradigm. A variety of programming pitfalls cause by inexperienced programmers might allow the system to perform desired functionalities, but it may directly reduce software quality. Design flaws are among these causes which are often occurring in developed software and artifacts.

Design flaws, introduced in early stages of software development or during evolution, are program properties that show a potential erroneous design of a software system. These flaws may result in low reuse, high complexity and low maintainability in developed systems. In the recent literature, design flaws are referred as Bad Smells [8] and AntiPatterns [2]. Many authors denote design flaws normally by using metaphors. They propose many approaches on how to recognize and correct such erroneous software flaws.

One famous and important technique to remove design flaws for improving software quality is manual software inspection [6, 7]. It incorporates sifting through source code, design and documentation. This technique plays an important role for quality assurance in modern software development processes like Extreme Programming. With such inspection technique, design flaws can be found before the testing stage. However, this technique is time consuming.

Another effective method of design flaws detection is using software metric. Many pieces of literature suggest the assistance of metrics in the detection of design flaws. Simon et al. [12] use object-oriented metrics to identify Bad Smells and also propose covered refactoring to improve the design of the code. Ducasse et al. [4] propose an approach to detect duplicated code in object-oriented software that should be avoided as it decreases maintainability. Emden and Moonen [5] integrate the detection of Bad Smells in Java with a visualization mechanism.

The goal of this paper is to introduce an approach to detect design flaws of an object-oriented system at the meta-level using the declarative meta programming. By using this declarative paradigm, design flaws can be detected in a simple way and proper results are obtained. This approach can help developers and maintainers to find design flaws in software systems easily in order to avoid program structures that cannot be extended and modified.

The remaining of this paper is organized as following. Section 2 describes the concepts of declarative meta programming and how to use this technique to transform the base program to be the input at the meta-level. An overview of the proposed detection and a case study are presented in Section 3. In Section 4, related works are discussed. Conclusions and future works are summarized in Section 5.

978-0-7695-3654-5/09 $25.00 © 2009 IEEE
DOI 10.1109/ICSPS.2009.143

2 Declarative Paradigm

Non-declarative programs, such as procedural programs and object-oriented programs, consist of two structures – data structure and control structure. Declarative programming languages differ from this scheme by expressing the logic of a computation without describing its control flow. It comprises not only of commands to be executed, but also the definitions and statements about the problem to be resolved.

2.1 Logic Programming [1]

Logic programming languages are declarative languages where the program represented by Horn clauses and where logic inference is the only control structure. Facts are used to specify static information which is always true in the application domain. Rules are used to derive new facts from existing ones. The premise of a rule specializes the conditions under which a new fact can be concluded. Queries are used to interrogate this information base.

2.2 Declarative Meta Programming

Declarative Meta Programming is defined as the use of a declarative programming language for writing meta programs. This meta programming has been investigated as a technique to support state-of-the-art software development [14].

The concept of declarative meta programming is very simple. A *meta program* must have access to a representation of its *object program* or *base program*. More precisely, the language of the object program must be represented in the language of the meta program.

The representation is a mapping from the symbols of the base language to the symbols of the meta-language, but this mapping is not simply a translation from one language to another. The mapping must also enable the meta-language to make statements about structures of the base language and must also be a quotation mechanism.

3 Design Flaws Detection Approach

In this section, the architecture of the proposed detection approach as shown in Figure 1 is described. The proposed detection architecture consists of two levels: the base level and the meta level. In the base level, base program is represented. The meta level consists of a number of facts and rules which together describe the base program in the base level. This level constitutes main three sub-modules : the Base-level meta module, the Composition-level meta module and the Design flaw-level meta module.

The transformation module analyzes and transforms the syntactical and semantical information of the base program into the meta program between the base level layer and the meta level layer. The transformation uses a general parse tree representation, which enables the use of fine-grained static information. The transformation module allows both layers to be language-independent. A base program is represented indirectly by means of a set of logic propositions. These logic propositions are stored in a logic database and they help to describe the represented base program and its logic representation in meta program.

This paper introduces the use of a declarative programming language – the most widely known logic programming language, PROLOG [13] – for writing meta program that represents the base programs in the declarative meta programming.

3.1 Domain definition

In this subsection, sets and relation domain of essential design-level information of object program for giving meta program description are defined such as class, package, method of classes, attribute of classes and statements of the method invocation.

An object program is represented as a set of eight components defined as follows.

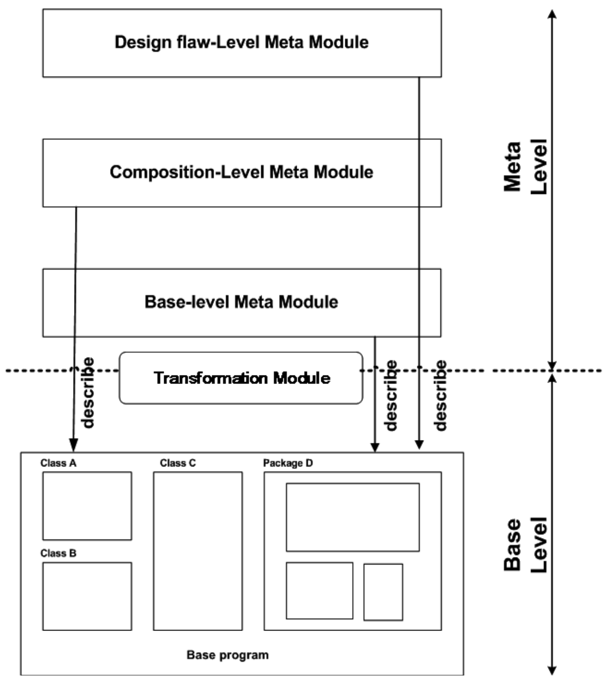


Figure 1. Architecture of the proposed detection approach

Object Program Sets = {P, C, A, M, S, R, λ, μ}
where :

P is a set of packages in the system.

C is a set of classes in packages of the system.

A is a set of attributes of classes where A $\subseteq$ C_0 x Name x visibility is the relation over the subsets of their Cartesian product of owner class C_0 of an attribute, an attribute name *Name* and its *visibility*, which consists of the set of {private, protected, public}.

M is a set of methods of classes where M $\subseteq$ C_0 x Name x m_r x m_{p1} x m_{p2} x ... x m_{pn} is the relation over the subsets of their Cartesian product of owner class C_0 of a method, a method name *Name*, a return type m_r = {object, primitiveType} and a list of parameter types $m_{p1}...m_{pn}$.

S is a set of statements of methods of the system.

The relation R over related sets is a set of subsets of the Cartesian product, where R $\subseteq$ ((CxC) $\cup$ (CxM) $\cup$ (CxA) $\cup$ (MxM) $\cup$ (MxA) $\cup$ (CxA) $\cup$ (MxC) $\cup$ (CxCxCxC) $\cup$ (CxCxC)).

λ : M $\rightarrow$ C_0 x Name x m_r x m_{p1} x m_{p2} x ... x m_{pn} is the function mapping any method *m* in M to its subset where $\lambda(m)_0$ denotes the owner class of method *m*, $\lambda(m)_r$ denotes the return type of method *m*, $\lambda(m)_n$ denotes the name of method *m* and $\lambda(m)_i$ denotes positioned parameter i of method *m* .

μ : A $\rightarrow$ C_0 x Name x visibility is the function mapping any attribute *a* in A to its subset where $\mu(a)_0$ denotes the owner class of attribute *a*, $\mu(a)_n$ denotes the name of attribute *a* and $\mu(a)_v$ denotes the visibility of attribute *a*.

3.2 Base-Level meta module

Meta programs in the declarative meta programming are represented in this level. In this paper, logic programming is used for the declarative programming language. In the base-level, basic rules and facts are generated for representing syntactical information elements of the object-oriented systems (e.g. classes, methods, attributes and statements) in the logic meta-language. The basic logic representation for rules and facts in formal definition of the predicates is shown in Table 1.

3.3 Composition-level meta module

In the second level, composition rules and facts are generated from the base-level meta module for representing relations of the object program such as method invocations or attribute access. It provides relations of defined elements in the base-level meta module to relate with the structure and behavior of the object-oriented systems in the base level. Some composition-level rules and facts representation are shown in terms of predicate logics as follows.

Table 1. Basic Logics Representation

Basic rules	Description
package(*p*)	*p* is a package where *p* $\in$ P
class(*c*)	*c* is a class where *c* $\in$ C
statement(*s*)	*s* is a statement where *s* $\in$ S
method(*m*)	*m* is a method where *m* $\in$ M
attribute(*a*)	*a* is an attribute where *a* $\in$ A

new(method(*m*), class(*c*)) is true when (*m, c*) $\in$ R and there exists a statement within method *m* which creates an object of class *c* by using *new* keyword.

new(class(c_1), class(c_2)) is true when (c_1, c_2) $\in$ R and $\exists m \in$ M[($\lambda(m)_0 = c_1$) $\wedge$ new(m, c_2)].

own(class(*c*), method(*m*)) is true when (*c, m*) $\in$ R and $\exists m \in$ M[$\lambda(m)_0 = c$].

call(method(m_1), method(m_2)) is true when (m_1,m_2) $\in$ R and there exists a statement within method m_1 of any class that calls method m_2 of any class.

call(class(*c*), method(*m*)) is true when (*c ,m*) $\in$ R and $\exists m' \in$ M[$\lambda(m')_0 = c \wedge$ call(m, m')].

call(class(c_1), class(c_2)) is true when (c_1,c_2) $\in$ R and $\exists m$ $\in$ M[$\lambda(m)_0 = c_1 \wedge$ call(m, c_2)].

use(method(*m*), attribute(*a*)) is true when (*m, a*) $\in$ R and there exists a statement within method *m* of any class that use attribute *a* of any class.

3.4 Design flaw-Level Meta Module

In the third level, design flaw inference rules that represent the existence of design flaws in object-oriented software are defined. Predicate logics from the lower layer are used to define these inference rules which consist of boolean expression and predicate logic rules. In next subsection, the detail of the proposed detection approach is presented.

3.5 Detecting design flaws

The proposed approach for design flaw detection consists of two phases: Representation phase and Detection phase. The Representation phase translates structural and behavioral object-oriented information into logic rules in Base-Level meta module and Composition-level meta module. Inference rules are defined from the design flaw concepts by using previous defined logic rules and are stored in Design flaw-Level Meta Module. The set of inference rules are manually developed based on heuristic and knowledge of human experts. After that, the prototype for detecting design flaws of object-oriented software is implemented based on the defined inference rules. This prototype is validated before it is used to detect design flaws in Detection phase. If results from prototype are incorrect, inference rules are redefined.

At Detection phase, the input of this phase is Java programs. By consolidating programs, the design information of programs is extracted and represented by formal design information in the same manner like logic rules in Representation phase. After that, the formal design information is transformed to a set of PROLOG logic facts. We use executable program prototype from Representation phase to find design flaws by logic query.

In next subsection, some inference rules used for design flaw detection is presented. These rules use to detect high-coupling [8] flaws such as Message Chains, Inappropriate Intimacy and Middle Man.

3.5.1 Message Chains

Message Chains is a flaw that occurs when objects communicate to each other with improper paths. Figure 2 shows the typical sequence of the Message Chains design flaw. Message Chains exists when class A needs data from class D. To access this data, class A needs to retrieve object C from object B. When class A gets object C it then asks C to get object D. When class A finally has a reference to class D, A asks D for the data it needs. The problem here is that A becomes unnecessarily coupled to classes B, C, and D, when it only needs some piece of data from class D. An inference rule that involves with 3-class relation of an candidate of Message Chains design flaw detection is proposed as following.

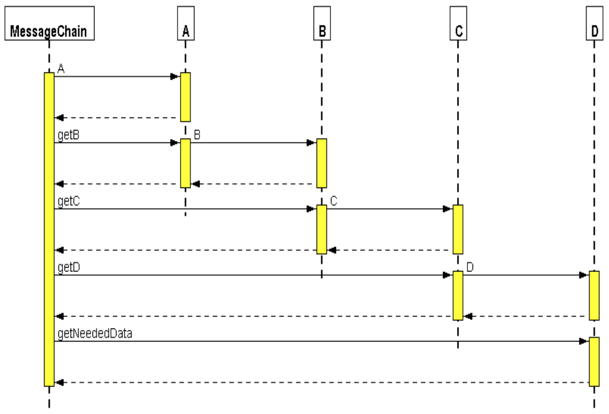


Figure 2. Message Chains Design Flaw

MessageChainsCandidate = (c_a, c_b, c_c) $\in$ R $\wedge$ own(class(c_a), method(m_a)) $\wedge$ $\exists c \in$ C[return(method(m_a), class(c))] $\wedge$ new(method(m_a), class(c_b)) $\wedge$ $\lambda(m_a)_r$ = "object" $\wedge$ own(class(c_b), method(m_b)) $\wedge$ return(method(m_b), class(c_a))) $\wedge$ new(method(m_b), class(c_c)) $\wedge$ $\lambda(m_b)_r$ = "object" $\wedge$ own(class(c_c), method(m_c)) $\wedge$ return(method(m_c), class(c_b)) $\wedge$ $\lambda(m_c)_r$ = "primitiveType".

The 3-class candidate Message Chains occurs when class c_a need primitive data from class c_c. Method m_a in class c_a creates object of class c_b and the return type of this method is defined to be an object. Similarly to class c_a, method m_b in class c_b creates object of class c_c and gets desired data from class c_c.

Message Chains for more than three classes can be constructed in the same way. Note that Message Chains with more than four classes is not considered in this paper.

3.5.2 Inappropriate Intimacy

Because of bad design, Inappropriate Intimacy occurs when one class accesses to the internal parts that should be private of another class. This situation can be taken place in two forms in object-oriented paradigm. One situation called general form happens between any two classes and another one called subclass form occurs between superclass and subclass. Two inference rules for detecting each flaw are as following.

InappropriateIntimacyCandidate (General form) = (c_1, c_2) $\in$ R $\wedge$ $\exists m \in$ M, $\exists a \in$ A[own(class(c_1), method(m)) $\wedge$ own(class(c_2), attribute(a)) $\wedge$ call(method(m), attribute(a)) $\wedge$ $\neg\exists \mathrm{m} \exists a \lambda(m)_0 \neq \mu(a)_0$].

This inference rule can be described that Inappropriate Intimacy in general form exists when method m of class c_1 try to access directly attribute a of class c_2 and class c_1, c_2 must not be the same class.

InappropriateIntimacyCandidate (Subclass form) = (c_1, c_2) $\in$ R $\wedge$ inherit(class(c_1), class(c_2)) $\wedge$ $\exists m \in$ M, $\exists a \in$ A[own(class(c_1), method(m)) $\wedge$ own(class(c_2), attribute(a)) $\wedge$ call(method(m), attribute(a)) $\wedge$ $\neg\exists \mathrm{m} \exists a \lambda(m)_0 \neq \mu(a)_0$].

Inappropriate Intimacy in subclass form exists in the same nature like general form, but method that wants to access data from the child class is the parent class.

3.5.3 Middle Man

Middle Man occurs when most methods of a class call the same or a similar method on another class. It involves with three classes which result in unnecessary method invocation and should be removed. An inference rule for detecting this flaw consists of two invocation among three classes.

MiddleManCandidate = firstInvocation $\wedge$ secondInvocation

firstInvocation = $(c_1, c_2) \in$ R $\wedge$ $\exists m_1 \exists m_2 \in$ M[$\lambda(m_1)_0 = c_1 \wedge \lambda(m_2)_0 = c_2 \wedge$ call(method(m_1), method(m_2)) $\wedge$ $\lambda(m_1)_n \neq \lambda(m_2)_n$]

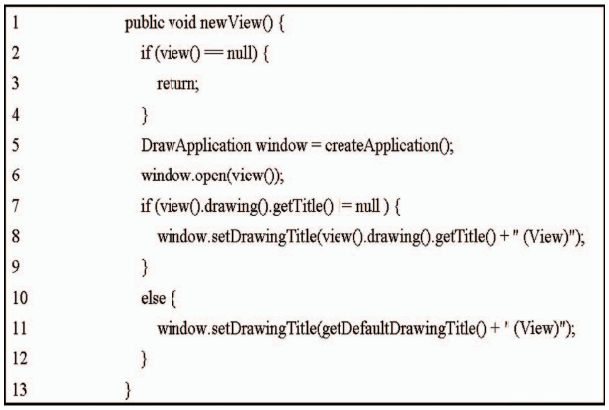
secondInvocation = $(c_2, c_3) \in$ R $\wedge$ $\exists m_2 \exists m_3 \in$ M[$\lambda(m_2)_0 = c_2 \wedge \lambda(m_3)_0 = c_3 \wedge$ call(m_2, m_3) $\wedge$ $\lambda(m_2)_n \neq \lambda(m_3)_n$]

3.6 Case study

A prototype for supporting this concepts of detection is implemented by using Eclipse IDE tool and SWI-Prolog program. We use the prototype to detect a program called JHotDraw. This program is small-sized to medium-sized which consists of 20 classes with around 30,000 lines of code. To obtain preliminary result, some packages in JHotDraw are used for detecting design flaws such as *Application*, *Figures* and *Standard* package.

3.6.1 *Step 1 - Representing rules and facts*

Java Compiler Compiler version 1.5 is used for parsing Java source code and constructing general representation trees. In Representation phase, design information and design flaws are extracted from representation trees and PROLOG inference rules for detecting design flaws are manually defined. In Detection phase, JHotdraw source code is translated to logic facts according to defined rules in Base-level meta module and Composition-level meta module.



```
1   public void newView() {
2       if (view() == null) {
3           return;
4       }
5       DrawApplication window = createApplication();
6       window.open(view());
7       if (view().drawing().getTitle() != null ) {
8           window.setDrawingTitle(view().drawing().getTitle() + " (View)");
9       }
10      else {
11          window.setDrawingTitle(getDefaultDrawingTitle() + " (View)");
12      }
13  }
```

Figure 3. Method *newView* **in class** *DrawApplication* **of** *Application* **package**

3.6.2 *Step 2 - Flaw detection*

After inference rules and facts have been defined, design flaw detection is performed by logic query in SWI-Prolog. Figure 3 illustrates a fragment of code of method *newView* in class *DrawApplication* of *Application* package. Method *newView* opens a new view for drawing at the current activated window. At line number 8, the program sets the new title of drawing. By performing this operation, class *DrawApplication* in package *org.jhotdraw.application* needs to set title data from class *StandardDrawing* in package *org.jhotdraw.standard*. However, class *DrawApplication* need to retrieve object *StandardDrawing* from object *DrawingView*. Finally, when class *DrawApplication* has a reference to class *StandardDrawing*, *DrawApplication* asks *StandardDrawing* for performing data it needs. The problem here is that class *DrawApplication* becomes unnecessarily coupled to classes *DrawingView*. Therefore, method *newView* is detected to have Message Chains design flaws according to the 3-class relation inference rule. The number of existences of design flaw in some packages of JHotDraw software are summarized in Table 2.

As it can be seen from the case study, design flaws can be detected by the defined rules. After investigating those candidates in program code and comparing them with the characteristics of the design flaws, the proposed technique can well detect design flaws in Java source code. Moreover, in the meta level, we can construct simple and straightforward query rules for searching systems that have high complexity and a large combination of components using backtracking and unification mechanism of declarative programming language.

4 Related works

There are several works to present the use of software metrics for design flaws detection. Miceli1 et al. [9] propose a technique for automatical detection based on analyzing the impact of various transformations on software metrics using quality estimation models. The metric-based

Table 2. Design flaw in packages of JHotDraw software

Package	Number of candidate design flaws		
	Message Chains	Inappropriate Intimacy	Middle Man
Application	5	0	4
Figures	4	0	4
Standard	5	1	2

heuristic framework proposed by Salehie et al [11] is to detect and locate object-oriented design flaws at the class level from the source code. Design flaws are detected and located systematically in two phases using a generic object-oriented design knowledge-base. Improving accuracy detection of metric-based techniques in design flaws detection are supported by the Tuning Machine method [10], based on a genetic algorithm, which tries to find automatically the proper threshold values.

Using logic rules for improving software quality in software development and maintenance stage is not new issue. Consens et. al. [3] proposes the graphical visualization techniques that can help to understand system and represent structural design information. Wuyts [16] proposes a declarative framework that allows reasoning about the structure of object-oriented programs. As similar to our work, Tourwé and Mens [15] use a semiautomated approach based on logic meta programming to detect bad smells and proposing refactoring opportunities to remove these smells. The difference between this method and our proposed method is on how to define logic rules. Their research works focus on objective-based defined rules that make these derived rules clearly, but we propose the method that begins with defining components of object-oriented system and construct rules from these components that enable the rules to be defined easily and complexity is reduced.

5 Conclusions

In this paper, we propose an automated approach to detect design flaws in Java source code. We use the technique of the declarative meta programming that consists of two main layers for detection: the base level and the meta level. In our approach, how to identify the candidate design flaws for a particular design flaws is formulated as inference rules. First, we define the domain definition that clarifies the scope in the base program in Java language. After that, the logic rules and the inference rules that are used for flaws detection are proposed respectively. Logic facts from Java source code are translated in the same way as logic rules translation. We show a design flaws detection example and the result shows that the proposed approach can detect design flaws. The proposed idea can reveal design flaws in object-oriented software and reduce possible maintenance cost.

At this time, our approach supports only some coupling-related flaws. We plan to extend our approach to detect more design flaws such as ones in [8, 2] and to cover some flaws generated from design patterns by developing more inference rules and strategies. In addition, we will investigate further to apply artificial intelligence techniques with our approach for more accuracy design flaw detection.

References

[1] M. Bramer. *Logic Programming with Prolog*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[2] W. J. Brown, R. C. Malveau, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, March 1998.

[3] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 138–156, New York, NY, USA, 1992. ACM.

[4] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings ICSM99 (International Conference on Software Maintenance*, pages 109–118. IEEE, 1999.

[5] E. V. Emden and L. Moonen. Java quality assurance by detecting code smells. In *in Proceedings of the 9th Working Conference on Reverse Engineering. IEEE Computer*, pages 97–107. Society Press, 2002.

[6] M. E. Fagan. Advances in software inspections. pages 609–630, 2002.

[7] M. E. Fagan. Design and code inspections to reduce errors in program development. pages 575–607, 2002.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[9] T. Miceli, H. Sahraoui, and R. Godin. A metric based technique for design flaws detection and correction. *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 307–310, Oct 1999.

[10] P. Mihancea and R. Marinescu. Towards the optimization of automatic detection of design flaws in object-oriented software systems. *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 92–101, March 2005.

[11] M. Salehie, S. Li, and L. Tahvildari. A metric-based heuristic framework to detect object-oriented design flaws. *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 159–168, 0-0 2006.

[12] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *CSMR*, pages 30–38, 2001.

[13] L. Sterling and E. Shapiro. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.

[14] T. Tourwé and T. Mens. A declarative meta-programming approach to framework documentation. In *In Proceedings of the Workshop on Declarative Meta Programming to Support Software Development (ASE02)*, 2002.

[15] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 91, Washington, DC, USA, 2003. IEEE Computer Society.

[16] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA*, pages 112–124. IEEE Computer Society Press, 1998.