STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVIII**, Number 1, 2013

A CONTENT ONTOLOGY DESIGN PATTERN FOR SOFTWARE METRICS

IONEL VIRGIL POP

Abstract. This paper presents a content ontology design pattern for the representation of software metrics, in software engineering ontologies, called OOPMetrics. This content ontology design pattern is designed to ease the detection of software design flaws based on the metrics that are defined in the ontology that uses it. We also present a case study that shows how an ontology that uses this pattern may be queried in order to detect these design flaws. In particular, we will focus on the God Class design flaw.

1. Introduction

In the field of ontology engineering, content ontology design patterns were introduced in [12]. They are a class of ontology design patterns and are useful in reusing concepts over many ontologies. In the field of software engineering, it was shown by Marinescu in [17, 18], that software metrics gathered through code analysis can help in detecting design flaws in software systems.

Sometimes we need to represent software metrics in a software engineering ontology, because software engineering ontologies often make use of software metrics in their content. For this purpose, we present in this paper a content ontology design pattern for software metrics, called OOPMetrics. The content ontology design pattern that is proposed in this paper is not extracted from a particular ontology, but is based on the best practices of using software metrics in the semantic web, mostly from the approach taken in [16] to query knowledge about software metrics. By not having such a content ontology design pattern it is clearly seen from the first stages of ontology development that the naming and relationships among different components that interact

Received by the editors: November 29, 2012.

2010 *Mathematics Subject Classification.* 68N30, 68Q55, 68T30, 97P20.

1998 *CR Categories and Descriptors.* D.2.8 [**Software**]: Software Engineering – *Metrics*; I.2.4 [**Computing Methodologies**]: Artificial Intelligence – *Knowledge Representation Formalisms and Methods*.

Key words and phrases. Content Ontology Design Patterns, Software Metrics, Reverse Engineering.

in a software metrics ontology is very difficult to decide. This is especially the case if one wants a unified system of components with that of other ontologies or wants to query the software metrics ontology to extract software design flaws based on the metrics.

Although OOPMetrics is described here for the first time in a scientific article, we also made available a short description of OOPMetrics online in the pattern collection from the ODP Portal at [23] with a link to it's implementation [22]. However the short description available there was mainly extracted automatically from the pattern's implementation based on it's annotations. And the ontological elements listed there were also extracted automatically from it's implementation.

This paper is structured as follows: after this introductory section, the next one deals with the definitions of the terms used throughout the paper and presents the related work that was done so far in this area. Section 3 describes our proposed content ontology design pattern. Section 4 presents a case study on behavioral god classes. In section 5 we draw some conclusions and present the work that we intend to do in the future.

2. Background and Related Work

The notion of ontology has many definitions in literature. One of the earliest definitions that were given, that is also suitable for the context in which ontologies are described in this paper, is the following:

Definition 2.1. *"An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary."*[20]

Knowledge patterns were described in [5], an updated version of [4]. Later, this notion was extended and ontology design patterns were described, based on [12, 24] as providing *"a modeling solution to solve a recurring ontology design problem"*[21].

There are six classes of Ontology design Patterns (OP), as they were classified in [13]: Structural OPs, Correspondence OPs, Content OPs (CPs) or Content (or conceptual) Ontology Design Patterns (CODePs), Reasoning OPs, Presentation OPs and finally Lexico-Syntactic OPs. We will focus on CPs in this paper as the other ontology design patterns are beyond the scope of this paper.

In Gangemi's article [12] the notion of a Conceptual (or Content) Ontology Design Pattern (CODeP) is described for the first time. In [13] the following definition is given:

Definition 2.2. *CODePs "encode conceptual, rather than logical design patterns. In other words, while Logical OPs solve design problems independently of a particular conceptualization, CPs propose patterns for solving design problems for the domain classes and properties that populate an ontology, therefore addressing content problems."*[13]

Software maintenance and the evolution of software systems were first addressed by Lehman in [14]. Software maintenance accounts for some 70 percent of the total expenditure required in the life cycle of a software system [15].

Software metrics ontologies, especially those that use an appropriate content ontology design pattern, may be queried to find design flaws in the software system. *"The presence of design flaws in a software system has a negative impact on the quality of the software, as they indicate violations of design practices and principles, which make a software system harder to understand, maintain, and evolve"*[9].

Possible design flaws in object-oriented software design were presented by Selvarani et al. in [26]. According to them, they are:

- In the case of *improper coupling:* **Shotgun Surgery** (at class level) and **Wide Subsystem Interface** (at subsystem level);
- In the case of *low cohesion:* **Feature Envy** (at method level) and **Misplaced Class** (at subsystem level);
- In the case of *improper distribution of complexity:* **God Class** (at class level), **God Method** (at method level) and **God Package** (at subsystem level);
- In the case of *flaws related to data abstraction:* **Data Class, Refused Bequest** (both at class level).

Marinescu [17, 18] has done notable work in the area of re-engineering software systems and the current work is based on his papers. Particularly we use the method presented in [17] to show how design flaws may be detected.

Li et al. [16] have brought their contribution in integrating software metrics data using semantic web techniques.

In [7], Şerban has proposed a quantitative evaluation methodology for object-oriented design, based on static analysis of the source code, and described by a conceptual framework. Serban's conceptual framework has four layers of abstraction [7, 8]: Object-Oriented Design Meta-Model, Formal Definitions of Object-Oriented Design Metrics, Specifications of the Assessment Objectives and Measurement Results Analysis. In [8], a case study for her approach was presented, involving God Class design flaw detection.

A number of content ontology design patterns were designed recently. The ODP Portal [25] has the descriptions for a collection of such content ontology design patterns that were designed for many domains. At present we have however not found any other content ontology design pattern for the software engineering domain on this portal, besides our OOPMetrics [22, 23] content ontology design pattern, even though this is a large and perhaps the only comprehensive collection of content ontology design pattern descriptions on the web.

3. Description of the Proposed Content Ontology Design Pattern

The name of the pattern described in this section is Object-Oriented Programming Metrics Pattern (OOPMetrics). This is a content ontology design pattern for software metrics. Ontologies that use our content ontology design pattern may be queried to detect flaws in design, based on the value of the metrics.

In the description of our content ontology design pattern we will follow some of the steps of describing a pattern that were used in describing software design patterns in [11], such as: intent, motivation, applicability, structure, participants, collaboration, consequences and implementation.

3.1. **Intent.** The goal of this content ontology design pattern is to represent object-oriented software metrics especially for the purpose of detecting flaws in the design of software systems based on these metrics. This may be useful for re-engineering the software system.

3.2. **Motivation.** We consider a context where we have a properly designed ontology for object-oriented software metrics that is based on the OOPMetrics content ontology design pattern. Now let us consider the following scenario: find which class is a God Class based on it's metrics. By using a simple query over the ontology, we can find the God Class if it exists, because we have used a content ontology design pattern that facilitated this. Of course it would have been possible to query it without using a content ontology design pattern to design the ontology. However, in this situation, every software metrics ontology will have to define the concept of software metric in it's own way. Thus, software metrics ontologies would have to be queried in various, different ways that are not necessarily optimized to detect God Classes. In order to have a unified content for the software metrics ontologies, we need a content ontology design pattern like OOPMetrics.

3.3. **Applicability.** The domain of applicability for the OOPMetrics content ontology design pattern is software engineering, more particularly software metrics. This content ontology design pattern has very good applicability in detecting design flaws in software systems based on metrics. Mainly, we have identified the following competency questions:

- What are the software metrics for a certain project or package or class or method?
- Knowing the necessary software metrics, is there a design flaw in the software system?

3.4. **Structure, Participants and Collaboration.** Figure 1 represents the diagram of the OOPMetrics content ontology design pattern. Additionally, table 1. and table 2. provide some of the elements that do not appear on the diagram.

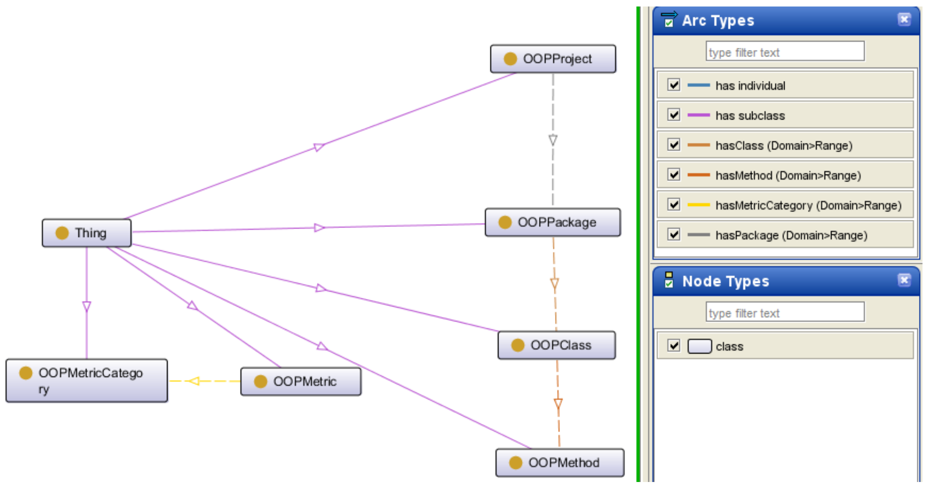

FIGURE 1. The OOPMetrics Diagram

The conceptual elements behind the OOPMetrics content ontology design pattern are classes, data properties and object properties. The following classes are defined:

- **OOPProject**: This class represents a software project;
- **OOPPackage**: This class represents the "package" concept found in object-oriented programming;
- **OOPClass**: This class represents the "class" concept found in object-oriented programming;
- **OOPMethod**: This class represents the "method" concept found in object-oriented programming;
- **OOPMetricCategory**: A (computed) software metric belongs in an OOPMetricCategory. Therefore this class represents the category in

which it belongs to (or what kind of software metric it is). A software metric is therefore represented separately in two distinct classes. These are: OOPMetricCategory (for defining a software metric) and OOPMetric (for computing a software metric) for flexibility reasons;
- **OOPMetric**: This class represents the concept of a (computed) software metric found in object-oriented programming.

In table 1, we show the data properties that are specific to OOPMetrics.

Data Property	**Domain**	**Range**
hasFloatValue	OOPMetric	float
hasIntegerValue	OOPMetric	integer
hasLongName	OOPMetricCategory	string
hasName	OOPMetricCategory	string
hasTag	OOPMetricCategory	integer

Table 1. Data Properties of the OOPMetrics pattern

In table 2, we show the object properties that characterize OOPMetrics.

Object Property	**Domain**	**Range**
hasPackage	OOPProject	OOPPackage
hasClass	OOPPackage	OOPClass
hasMethod	OOPClass	OOPMethod
hasMetricCategory	OOPMetric	OOPMetricCategory
hasMetric	UnionOf: OOPProject, OOPPackage, OOPClass, OOPMethod	OOPMetric

Table 2. Object Properties of the OOPMetrics pattern

3.5. **Consequences.** The OOPMetrics content ontology design pattern allows ontology engineers and software engineers to design software metrics ontologies easier and in a more unified way. Also, this content ontology design pattern was designed in such a way that software design flaws can be easily detected based on software metrics by using it.

3.6. **Implementation.** It is essential for a content ontology design pattern to be implemented in a particular ontology language in order for the content to be imported later when an ontology is created. This content ontology design pattern was implemented in the Web Ontology Language (OWL) [1], using the ontology editor and knowledge acquisition system Protégé [10]. The implementation is available at [22]. This implementation of the OOPMetrics content ontology design pattern can be imported in Protégé and used in designing software metrics ontologies.

4. Case Study

In this section, a case study will be presented regarding how a software metrics ontology that uses the OOPMetrics pattern may be queried, in order to detect software design flaws. The focus in this case study will be on the God Class design flaw.

In [17] three metrics were used to detect God Classes: Weighted Methods Per Class (WMC) that is defined in [3] and may use various complexity measures such as McCabe's cyclomatic complexity [19], Tight Class Cohesion (TCC), defined in [2] and Access to Foreign Data (ATFD), described in [17]. In [17], it was also explained that high values of WMC and ATFD and low values of TCC may lead to God Classes.

In [26] a formula was presented to detect God Classes. The following formula is an example of how to detect suspects, that is based on [26], but with absolute values:

$$\mathbf{GodClass(C) = (WMC(C) > 100)\ and\ (ATFD(C) > 1)\ and\ (TCC(C) < 0.5)}.$$

We must emphasize the fact that it is beyond the scope of this paper to prove that a detection strategy, such as the one based on the metrics used here to detect God Classes, actually works, because this has already been shown in [17, 18]. Therefore, this paper does not attempt to restate Marinescu's approach by presenting a case study for large software projects with hundreds or thousands of classes to show how such a detection strategy may work. Instead, the point of this case study is to show how the concepts and the relationships defined in the OOPMetrics pattern can be used in practice, in a SPARQL query. Thus, the query in figure 2 shows how an ontology that uses the OOPMetrics pattern may be queried by users, if the ontology was designed using this pattern.

In the query example from figure 2, a hypothetical ontology that uses the OOPMetrics content ontology design pattern is queried. The reason why a hypothetical ontology was chosen here and not a real world ontology is to show that any ontology that uses this content ontology design pattern may be queried in a similar manner in order to detect which classes are God Classes. However, in order to make sure that this query is correct, we have tested this query in Protégé with an ontology that we have designed using the OOPMetrics pattern.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX oopmetrics: <http://www.cs.ubbcluj.ro/~ivpop/ontologies/
oopmetrics.owl#>

SELECT DISTINCT ?class
WHERE {
    ?class rdf:type oopmetrics:OOPClass .

    ?class oopmetrics:hasMetric ?metric1 .
    ?metric1 rdf:type oopmetrics:OOPMetric .
    ?metric1 oopmetrics:hasMetricCategory ?wmc .
    ?wmc rdf:type oopmetrics:OOPMetricCategory .
    ?wmc oopmetrics:hasTag 1 .
    ?metric1 oopmetrics:hasIntegerValue ?v1 . FILTER (?v1 > 100)

    ?class oopmetrics:hasMetric ?metric2 .
    ?metric2 rdf:type oopmetrics:OOPMetric .
    ?metric2 oopmetrics:hasMetricCategory ?atfd .
    ?atfd rdf:type oopmetrics:OOPMetricCategory .
    ?atfd oopmetrics:hasTag 2 .
    ?metric2 oopmetrics:hasIntegerValue ?v2 . FILTER (?v2 > 1)

    ?class oopmetrics:hasMetric ?metric3 .
    ?metric3 rdf:type oopmetrics:OOPMetric .
    ?metric3 oopmetrics:hasMetricCategory ?tcc .
    ?tcc rdf:type oopmetrics:OOPMetricCategory .
    ?tcc oopmetrics:hasTag 3 .
    ?metric3 oopmetrics:hasFloatValue ?v3 . FILTER (?v3 < 0.5)
}
```

FIGURE 2. SPARQL Query

In the query from figure 2 it is considered that an individual WMC, exists in the ontology, of type OOPMetricCategory for which the value of the hasTag data property is 1, an individual ATFD, exists in the ontology, of type OOP-MetricCategory for which the value of the hasTag data property is 2, and an individual TCC, exists in the ontology, of type OOPMetricCategory for which the value of the hasTag data property is 3. They could have been referred to by using the hasName or hasLongName properties in a similar manner.

5. Conclusions and Future Work

Content ontology design patterns provide a very convenient way of reusing components in an ontology. This paper has described a content ontology design pattern for software metrics. Then a case study was presented that showed how an ontology that uses such a content ontology design pattern may be queried.

Besides quering the ontology that uses the OOPMetrics content ontology design pattern to extract knowledge like in the case study presented in this paper, it is also possible, through rules, to inference knowledge using reasoners such as Pellet [6]. This allows for further exploitation of the OOPMetrics content ontology design pattern for even more advanced purposes. In the future we intend to exploit the advantages of using reasoners to infer knowledge over ontologies that use this content ontology design pattern by adding rules to it.

References

[1] Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A., OWL Web Ontology Language Reference, W3C Recommendation 10 february 2004, Dean, M. and Schreiber, G., eds., http://www.w3.org/TR/owl-ref/.

[2] Bieman, J. M., and Kang, B. K., Cohesion and reuse in object-oriented systems, In *Proceedings of the 1995 Symposium on Software Reusability*, ACM, USA, 1995, pp. 259–262.

[3] Chidamber, S., and Kemerer, C., A metrics suite for object-oriented design, *IEEE Trans. Softw. Eng. 20*, 6 (1994), 476–493.

[4] Clark, P., Thompson, J., and Porter, B., Knowledge patterns, In *KR'2000 (Proc 7th Int Conf)*, A. Cohn, F. Giunchiglia, and B. Selman, Eds. Kaufmann, 2000, pp. 591–600.

[5] Clark, P., Thompson, J., and Porter, B., Knowledge patterns, In *Handbook on Ontologies*, s. Staab and R. Studer, Eds. Springer, 2003, pp. 191–207.

[6] Clark&Parsia, Pellet: OWL 2 reasoner for java, http://clarkparsia.com/pellet, 2004.

[7] Şerban, C., A conceptual framework for object-oriented design assessment, In *Fourth UKSim European Modelling Symposium on Computer Modelling and Simulation (EMS)*, 2010, pp. 90–95.

[8] Şerban, C., God class design flaw detection in object oriented design. a case study, *Studia Univ. Babeş-Bolyai, Informatica LVI*, 4 (2011), 33–38.

[9] D'Ambros, M., Bacchelli, A., and Lanza, M., On the impact of design flaws on software defects, In *Proceedings of the 10th International Conference on Quality Software*, IEEE Computer Society, USA, 2010, pp. 23–31.

[10] Stanford Center for Biomedical Informatics Research (BMIR), The Protégé ontology editor and knowledge acquisition system. http://protege.stanford.edu/.

[11] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, USA, 1994.

[12] Gangemi, A., Ontology design patterns for semantic web content, In *Proceedings of the Fourth International Semantic Web Conference*. Springer, 2005, pp. 262–276.

[13] Gangemi, A., and Presutti, V., Ontology design patterns, In *Handbook on Ontologies, Second Edition*, International Handbooks on Information Systems. Springer, 2009, pp. 221–243.

[14] Lehman, M. M., The programming process, IBM Res. Rep. RC 2722, 1969.

[15] Lehman, M. M., Programs, life cycles, and laws of software evolution, *Proc. IEEE 68*, 9 (1980), 1060–1076.

[16] Li, Y., F., and Zhang, H., Integrating software engineering data using semantic web technologies, In *Proceedings of 8th Working Conference on Mining Software Repositories (MSR 2011)*, ACM, USA, 2011, pp. 211–214.

[17] Marinescu, R., Detecting design flaws via metrics in object-oriented systems, In *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS 39)*, IEEE Computer Society, USA, 2001, pp. 173–182.

[18] Marinescu, R., Detection strategies: Metrics-based rules for detecting design flaws, In *In Proc. IEEE International Conference on Software Maintenance* (2004).

[19] McCabe, T. J., A complexity measure, *IEEE Trans. Softw. Eng. 2*, 4 (1976), 308–320.

[20] Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W., Enabling technology for knowledge sharing, *AI Magazine 12*, 3 (1991), 36–56.

[21] Noppens, O., and Liebig, T., Ontology patterns and beyond - towards a universal pattern language, In *Proceedings of the Workshop on Ontology Patterns (WOP 2009)*. 2009, pp. 179–186.

[22] Pop, I. V., Oopmetrics (owl file), http://www.cs.ubbcluj.ro/~ivpop/ontologies/oopmetrics.owl.

[23] Pop, I. V., Submissions:oopmetrics, http://ontologydesignpatterns.org/wiki/Submissions:OOPMetrics.

[24] Presutti, V., and Gangemi, A., Content ontology design patterns as practical building blocks for web ontologies, In *Proc. of the 27th Int. Conf. on Conceptual Modeling (ER)*. 2008, pp. 128–141.

[25] NeOn Project, Ontology design patterns . org (odp), http://ontologydesignpatterns.org.

[26] Selvarani, R., Banu, W., and Prasad, K., Quantifying the design quality of object oriented system the metric based rules and heuristic, In *National Conference on Advanced Software Engineering*. Bangalore, 2008, pp. 54–62.

E-mail address: popionelvirgil@yahoo.com

Department of Computer Science, Babeş-Bolyai University, 1 M. Kogălniceanu St., 400084 Cluj-Napoca, Romania