Towards a Model-driven Approach for Reverse Engineering Design Patterns

Awny Alnusair and Tian Zhao

University of Wisconsin-Milwaukee, USA
{alnusair,tzhao}@uwm.edu

Abstract. The size and complexity of software systems is rapidly increasing. Meanwhile, the ability to understand and maintain such systems is decreasing almost as fast. Model Driven Engineering (MDE) promotes the notion of modeling to cope with software complexity; in this paper we report on our research that utilizes ontological modeling for understanding complex software systems. We focus the discussion on recovering design pattern information from source code. We thus argue that an effective recovery approach needs to utilize semantic reasoning to properly match an ontological representation of both: conceptual source code knowledge and design pattern descriptions. Since design patterns can take different forms when implemented in code, we argue that hard-coding their descriptions limits the flexibility and usability of a detection mechanism.

Key words: OWL, Semantic Web, Ontology, Semantic Reasoning, Design Patterns, Program Understanding, Reverse Engineering

1 Introduction

In Software Engineering practices, software understanding deals with the process of studying software to mentally conceptualizing its behavior and inner working structure. Unfortunately, understanding voluminous and complex software is widely regarded as the most time consuming and resource intensive activity during both reverse and forward engineering practices. In order to cope with software complexity and consequently enable better understanding of software, Model Driven Engineering (MDE) has emerged as a software engineering discipline. MDE emphasizes the systematic use of models and modeling principles throughout the software development lifecycle. In fact, software design and design best practices (i.e., design patterns) are of particular relevance to MDE due to the heavy reliance on modeling techniques and principles.

Design patterns describe reusable solutions to common recurrent object-oriented design problems. The classic book [1] of design patterns introduced many of these patterns. Ever since then, these patterns have been liberally used in building and documenting new object-oriented systems as well as re-engineering legacy software. It is thus evident why one would be interested in reverse engineering approaches for design pattern instance recovery.

Central to all reverse engineering activities is the accurate identification of interrelationships among the components of a subject system as well as a true representation of the system at a higher level of abstraction [2]. As a typical reverse engineering activity, design pattern recovery is no exception. However, building an effective model that promotes understanding and truly represents the internal working structure and organization of a software system is not a straight forward task; it requires a formal, explicit, and semantic-based representation of the conceptual knowledge of source code artifacts found in the subject system. The research presented in this paper explores this hypothesis by providing ontological representations of software assets.

In our further exploration of this hypothesis, we take into account modeling principles and techniques from Model Driven Engineering. MDE promises full support for reverse engineering activities through modeling and model transformations at different levels of abstractions. Due to the formal and built-in reasoning foundations of ontologies, our methodology is solely reliant on ontology-based modeling. Therefore, we provide an OWL[1] ontology model that includes a software representation ontology; this ontology is automatically populated through text-to-model transformation with ontological instances representing various program elements of a subject system. Furthermore, each design pattern's structure including its participants' behaviors and collaborations are represented by a separate OWL ontology; this ontology also encodes the rules needed to detect this particular pattern.

The rest of the paper is organized as follows: In Section 2, we describe our approach for structuring and populating the knowledge base. We discuss the details of design pattern detection in Section 3. Implementation and evaluation case studies are discussed in Section 4. Finally, a discussion of the current state-of-the-art followed by future work directions are discussed in Section 5 and 6, respectively.

2 Ontology Model

Due to their ability to enable automatic knowledge sharing and understanding, ontologies are being used as the knowledge representation component of the Semantic Web vision [3]. To this end, many ontology modeling languages and Semantic Web technologies have emerged and been used in various domain areas. In our work, we utilize the Web Ontology Language (OWL), the Resource Description Framework (RDF)[2], the Semantic Web Rule Language (SWRL)[3], and SPARQL[4] query language.

OWL is used for capturing relationship semantics among domain concepts, OWL-DL is a subset of OWL based on Description Logic and has desirable computational properties for reasoning systems. OWL-DL's reasoning support

[1] http://www.w3.org/TR/owl-guide/
[2] http://www.w3.org/TR/rdf-primer
[3] http://www.w3.org/Submission/SWRL
[4] http://www.w3.org/TR/rdf-sparql-query

allows for inferring additional knowledge and computing the classification hierarchy (subsumption reasoning). RDF is used as a flexible data representation model. RDF is suitable for describing resources and provides a data model for representing machine-processable semantics of data. SWRL is used for writing rules that can be combined with OWL knowledge bases for reasoning and computing entailments. Finally, SPARQL is an RDF query language and protocol for ontological querying of RDF graphs.

In what follows, we describe how Semantic Web languages contribute to our methodology; in particular, we show how OWL-DL and RDF are used to obtain a precise formal representation of source code knowledge and design patterns. We also show how SWRL rules can be used to further extend the OWL representation of design patterns to handle features that cannot be expressed using OWL alone. Collectively, these representations form the base for semantic reasoning and inference of additional facts that can be retrieved using SPARQL semantic queries.

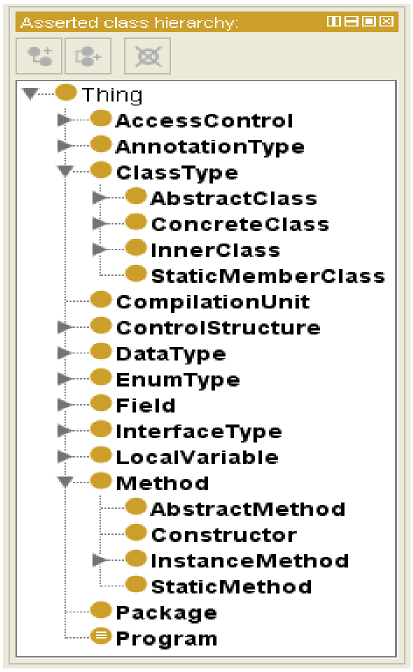
2.1 Structuring the Knowledge Base

At the core of the ontology model is a Source Code Representation Ontology (referred to afterwards as SCRO). This ontology is created to provide an explicit representation of the conceptual knowledge structure found in source code. SCRO captures major concepts of object-oriented programs and helps understand the relationships and dependencies among source code artifacts.

SCRO's knowledge is represented using the OWL-DL ontology language and verified using the Pellet OWL-DL reasoner [4]. It was designed with ontology reuse in mind, such that its vocabulary can be used not only for design pattern detection but also in any application that requires semantic source code information. This ontology is a work in progress, currently focused on the Java programming language; however, extensions for other object-oriented languages can be easily obtained. A fragment of the ontology's taxonomy using the Protégé [5] ontology editor is shown in Fig. 1 and the complete ontology can be found online [6].

Each of the concepts (classes) shown in Fig. 1 is interpreted as a set that defines a group of individuals (instances) sharing some properties. Disjoint subclasses are also modeled to define different sets of individuals. For example, all individuals that are members of class `InstanceMethod` are necessarily members of class `Method` but none of them can simultaneously be a member of another subclass of `Method`. Currently SCRO defines 56 OWL classes and subclasses. Those classes map directly to source code elements and collectively represent the most important concepts found in object-oriented programs such as method, class, nested class, and field.

Various object properties, sub-properties, and ontological axioms are defined within SCRO to represent relationships among concepts by linking individuals from different OWL classes. For example, `hasOutputType` is a functional property defined for the return type of a method and `hasSuperType` is a



```
Asserted class hierarchy:
Thing
    AccessControl
    AnnotationType
    ClassType
        AbstractClass
        ConcreteClass
        InnerClass
        StaticMemberClass
    CompilationUnit
    ControlStructure
    DataType
    EnumType
    Field
    InterfaceType
    LocalVariable
    Method
        AbstractMethod
        Constructor
        InstanceMethod
        StaticMethod
    Package
    Program
```

Fig. 1. Excerpt of SCRO class hierarchy

transitive object property defined with two transitive sub-properties for inheritance and interface implementation. Inverse properties are also used to define the inverse relation. For example, `isLocalVariableOf` is the inverse property of `hasLocalVariable`. This is in particular useful in many situations such as traversing the resulted RDF graph in both directions and making the ontology useful for applications that do not depend on a reasoner system. Currently, SCRO defines 73 object properties, sub-properties and data properties. Some of these properties are further discussed in Section 3.

2.2 Design Pattern Ontology Sub-model

In order to properly represent a design pattern, a proper identification of its participating classes, their instances, roles, and collaborations is indeed essential. We thus reuse the vocabulary defined in SCRO and build definitions for each design pattern. The result is a modular extensible structure of OWL ontologies linked together via regular OWL reuse mechanisms. This structure is depicted in Fig. 2.

The design-pattern ontology represents knowledge common to all design patterns. An ontology is created for each design pattern describing its essential participants and their properties, collaborations, restrictions, and the corresponding SWRL rules needed to detect this pattern. This modular structure promotes on-

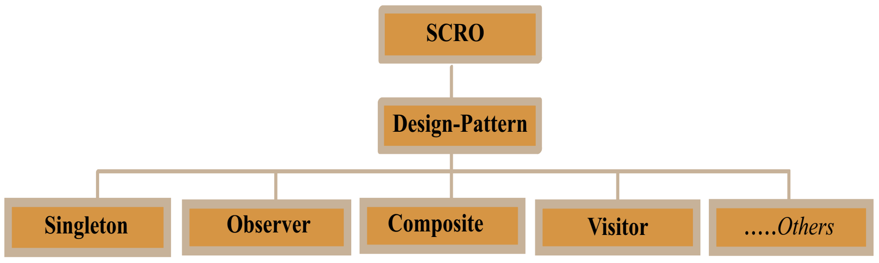


Fig. 2. Modular structure of design pattern ontologies linked via `owl:imports`

tology reuse, thus allowing SCRO to be used with various maintenance related activities and certainly with reasoners that do not support SWRL rules. Having the ontology structure created, the next natural step is to populate the knowledge base with ontological instances of various concepts in the ontology. This process is described next.

2.3 Automatic Knowledge Population

Populating knowledge repositories requires binding concrete resource information with the ontology to create instances of ontological concepts. These instances are the building blocks of the knowledge base that can be used for browsing and querying. When dealing with large source code repositories, a large number of instances is usually created, this is why automatic text-to-model transformation and knowledge population is essential.

Inspired by the Java RDFizer idea from the Simile[5] project, we have built a subsystem that automatically extracts knowledge from Java binaries. The original RDFizer was able to distinguish only class types, super types, and basic use relations represented in a class file. Currently, our subsystem performs a comprehensive parsing of the class file format specified by the Java Virtual Machine. It captures every ontology concept that represents a source code element and effectively generates instances of all ontological properties defined in our ontologies for those program elements. Our knowledge generator subsystem distinguishes itself as not being relied on the existence of source code to extract knowledge. This is in particular helpful for binary reverse engineers who must understand a software system in which the source code is unavailable.

The semantic instances generated by our subsystem are serialized using RDF triples; a separate RDF ontology in Notation3[6] syntax is generated for each application framework parsed, this amounts to instantiating an OWL knowledge base for that framework. This arrangement provides a clean separation of explicit OWL vocabulary definitions from the metadata represented by RDF. Since OWL is a vocabulary extension of RDF, the encoded metadata represented in RDF

[5] http://simile.mit.edu
[6] http://www.w3.org/DesignIssues/Notation3

can be naturally linked to the OWL design pattern ontologies via OWL reuse mechanisms. A sample output of our knowledge extractor subsystem for the JHotDraw[7] application framework can be examined online [6].

3 Design Pattern Recovery Approach

Due to their abstract nature and the varying responsibilities and interactions of their participants, design patterns can generally be implemented in different ways and using different techniques or language-specific constructs. Examples of such implementation variations are plenty. Consider object composition for example, some systems do not use a built-in collections framework component for maintaining an aggregate collection of objects. Instead, they use a user-defined data structure. Other systems favor some delegation techniques over the others when implementing design patterns such as Visitor, State, and Strategy. We thus believe that pattern detection tools should be flexible and easily extensible to improve usability and effectiveness, that is, the roles and interactions of participants should not be hard-coded. In fact, users of the detection tool often need to change those hard-coded role restrictions but find it extremely difficult because it requires code modification and rebuilding the entire tool.

In our approach, we aim at providing flexibility and transparency such that design patterns are specified externally using ontology formalisms and participant's responsibilities are depicted using ontology rules that can be easily understood and modified. We thus use the expressivity of OWL-DL and SWRL rules to formally represent each design pattern using a set of intent-preserving conditions. When these conditions are collectively met, an instance of the design pattern is detected. This approach relies solely on the definitions found in our ontologies and on an OWL-DL reasoner that is capable of computing inferences from the set of facts and SWRL rules defined in the ontologies. To test the validity of our approach, we authored SWRL rules for five well-known design patterns, namely, Visitor, Observer, Composite, Template-Method, and Singleton. In the next subsections, we illustrate our detection mechanism for Visitor and Observer. Since SWRL rules are part of the OWL ontology representing a particular design pattern, the reader can examine the rules for the other three patterns by downloading the corresponding ontology [6] and view it in an ontology editor such as Protégé. Note that our approach is not limited to these five design patterns, others can be easily formalized and thus detected using the same procedure.

3.1 Detecting the Visitor Design Pattern

Visitor is a behavioral design pattern that facilitates flexibility by allowing external operations to be performed on the elements of an object structure and hence not modifying the classes of the elements [1]. The idea is to keep the object

[7] http://www.jhotdraw.org

structure intact by defining new structures of visitors representing the new behaviors of interest. This pattern defines two separate class hierarchies: `Host` and `Visitor`. The following are three sample conditions for detecting this pattern:

1. At the root of the Visitor hierarchy is a `Visitor` interface common to all concrete visitors; this interface declares the invariant abstract behaviors (`visit` methods) that should be implemented by each `ConcreteVisitor`, since each `visit` method is designed for a `ConcreteHost`, the concrete host must be present as an argument (input type) in the corresponding `visit` method.
2. A `ConcreteVisitor` is a concrete class type that implements the `Visitor` interface and overrides each `visit` method to implement visitor-specific behavior for the corresponding `ConcreteHost`.
3. A `ConcreteHost` must define the `accept` instance method that takes `Visitor` as an argument. This method implements the double dispatching technique by calling the matching `visit` method and passing the host object into the visitor.

Intuitively, we should take advantage of OWL-DL expressivity to formalize the above restrictions. However, these restrictions require more expressive power than what Description Logic provides. OWL can only handle descriptions of infinite number of unstructured objects connected in a tree-like manner [7]. We thus use SWRL to handle all non-tree-like situations and property chaining for design pattern restrictions. SWRL extends OWL-DL with First Order Horn-like rules; a rule in SWRL has two parts: the antecedent and the consequent, each of these parts contain only positive conjunctions of either unary or binary atoms. In its simple form, a unary atom represents an OWL class predicate of the form `C(var1)` and a binary atom represents an OWL property predicate of the form `P(var1, var2)`, both `var1` and `var2` are variables over OWL individuals. The reasoner will carry out the actions specified in the consequent only if all the atoms in the antecedent are known to be true.

Listing 1 shows a sample SWRL rule that depicts the above conditions for the visitor design pattern. Please refer to SCRO and the visitor ontology found online [6] for the definitions of OWL properties used in this rule. Every restriction was formalized using three different atoms in the rule; the `hasInputType` OWL object property is defined in SCRO to represent a method's formal parameter and `methodOverrides` works for both method overriding and interface method implementation. Upon classifying the ontology, a reasoner with a rule engine such as Pellet would infer and thus create instances for the different participants of this design pattern as described in the consequent part of the rule.

```
scro:InterfaceType(?visitor) ∧
scro:hasAbstractMethod(?visitor, ?visit) ∧
scro:hasInputType(?visit, ?concrete-host) ∧

scro:hasSuperType(?concrete-visitor, ?visitor) ∧
scro:hasInstanceMethod(?concrete-visitor, ?c-visit) ∧
scro:methodOverrides(?c-visit, ?visit) ∧
```

```
scro:hasInstanceMethod(?concrete-host, ?accept) ∧
scro:hasInputType(?accept, ?visitor) ∧
scro:invokesMethod(?accept, ?visit)
   ⟹
visitor:Visitor(?visitor) ∧
visitor:hasConcreteVisitor(?visitor, ?concrete-visitor) ∧
visitor:hasConcreteHost(?visitor, ?concrete-host) ∧
visitor:hasVisitMethod(?concrete-visitor, ?c-visit) ∧
visitor:hasAcceptMethod(?concrete-host, ?accept)
```

Listing 1. A sample SWRL rule for Visitor

Relaxing or adding more restrictions to the requirements is relatively simple. For the sake of argument, one might want to retrieve only pattern instances that declare a super type for all concrete hosts in the `Host` hierarchy. This can be accomplished by modifying the third condition and introducing a fourth condition as shown below. The result is a new rule depicted in Listing 2.

3. A `ConcreteHost` is a subtype of `Host`. It defines the `accept` instance method that overrides the `hook` method found in `Host`. This method implements double dispatching by calling the matching `visit` method and passes the host in to the visitor.
4. At the root of the `Host` hierarchy is an interface or abstract class type. It represents the super type of all concrete hosts. It declares the abstract `hook` method; this method takes `Visitor` as an argument.

```
scro:InterfaceType(?visitor) ∧
scro:hasAbstractMethod(?visitor, ?visit) ∧
scro:hasInputType(?visit, ?concrete-host) ∧

scro:hasSuperType(?concrete-visitor, ?visitor) ∧
scro:hasInstanceMethod(?concrete-visitor, ?c-visit) ∧
scro:methodOverrides(?c-visit, ?visit) ∧

scro:hasSuperType(?concrete-host, ?host) ∧
scro:hasAbstractMethod(?host, ?hook) ∧
scro:hasInputType(?hook, ?visitor) ∧

scro:hasInstanceMethod(?concrete-host, ?accept) ∧
scro:methodOverrides(?accept, ?hook) ∧
scro:invokesMethod(?accept, ?visit)
   ⟹
visitor:Visitor(?visitor) ∧
visitor:hasHost(?visitor, ?host)
    ....................
```

Listing 2. A modified SWRL rule for Visitor

3.2 Detecting the Observer Design Pattern

Observer represents a one-to-many dependency between communicating objects such that when the subject object changes its state, it sends a notification message to all its listeners to be updated accordingly. Unlike the Composite pattern, Observer is implemented in two separate hierarchies of participants. An interface for all listeners sits at the root of the listener's hierarchy; it identifies a common behavior for all listeners interested in observing a particular subject such that each concrete listener maintains a reference to that particular subject. On the other hand, the subject knows its listeners, it provides means to establishing the relationship with them, and it is responsible for communicating the behavior change to all those registered listeners.

Listing 3 shows sample rule representations of this pattern. The first rule identifies candidates for potential listeners, their concrete listeners and the corresponding update methods.

```
scro:hasSuperType(?c-listener, ?listener) ∧
scro:hasMethod(?listener, ?update) ∧
scro:methodOverriddenBy(?update, ?c-update) ∧
scro:isMethodOf(?c-update, ?c-listener)
   ⟹
observer:hasCListenerCandidate(?listener, ?c-listener) ∧
observer:hasCUpdate(?c-listener, ?c-update) ∧
observer:hasUpdate(?listener, ?update)
-------------------------------------------------------------
scro:hasField (?c-subject, ?container) ∧
scro:hasStructuredDataType (?container, ?containerDT) ∧
scro:hasMethod (?containerDT, ?insert) ∧
scro:methodInvokedBy (?insert, ?add-listener) ∧
scro:isMethodOf (?add-listener, ?c-subject)
   ⟹
observer:hasAddListener(?c-subject, ?add-listener)
-------------------------------------------------------------
observer:hasCListenerCandidate(?listener, ?c-listener) ∧
observer:hasUpdate(?listener, ?update) ∧
observer:hasAddListener(?c-subject, ?add-listener) ∧
scro:hasInputType(?add-listener, ?listener) ∧
scro:hasPart(?c-listener, ?c-subject) ∧
scro:hasMethod(?c-subject, ?notify) ∧
scro:invokesMethod(?notify, ?update)
   ⟹
observer:hasConcreteListener(?listener, ?c-listener) ∧
observer:listensTo(?c-listener, ?c-subject) ∧
observer:Observer(?notify)
```

Listing 3. Sample SWRL rules for the Observer pattern

The OWL object property hasSuperType is made transitive so that the reasoner can infer all direct or indirect super types of a given class. The update method specified in the listener interface should be implemented by all concrete listeners. Recall that the object property methodOverrides works for both classes and interfaces.

The second rule effectively identifies potential candidates for concrete subjects and the method used for establishing the relationship between this subject and its listeners. The subject class maintains a collection of its listeners, typically stored in a field that has a structured data type. In SCRO, StructuredDataType represent arrays or collections of elements; it is a sub-class of ComplexDataType which also includes UnstructuredDataType such as user-defined data structures. Therefore, the rule in Listing 3 can be modified to detect containers of any kind.

The third rule builds on the other two rules and effectively ensures the conditions needed for detecting this pattern. The hasPart property ensures that a concrete listener must maintain a reference to the observable; the hasInputType property ensures that the candidate add-listener method accepts only listener objects, and finally the notification behavior is specified using method invocation.

4 Implementation and Evaluation

We are currently developing a program understanding tool that utilizes reasoning services over the ontological representation of source code elements. This tool currently accepts the Java byte code and the ontologies (Sections 2.1 and 2.2) as input; the knowledge generator module parses the byte code and generates the RDF repository (Section 2.3). This repository is stored and managed by Jena [8], an open source Java framework for building Semantic Web applications by providing programmatic support and handling for RDF, OWL, and SPARQL. Our tool currently supports SPARQL ontological queries against the knowledge base. Several SPARQL queries are embedded within the tool for retrieving the detected design pattern details. This process is simple and fully automated, no manual interaction is required by the reverse engineer.

4.1 Case Study: Design Pattern Detection

We have conducted a preliminary study on multiple open source frameworks including JUnit[8], JHotDraw, and Java AWT. The chosen frameworks vary in size, they represent different domains, and most importantly, they were built with design patterns in mind; this makes them a good fit for evaluating our approach. Instance detection is shown in Table 1.

The interpretation of a pattern instance can be readily obtained from the corresponding rule for that pattern. For example, a Composite instance represents the child element's container, a Singleton instance represents the singleton

[8] http://www.junit.org

Table 1. Inferred design pattern instances

	Visitor			Observer			Composite			T-Method			Singleton			
	T[1]	TP[2]	Pr[3]	T	TP	Pr	T	TP	Pr	T	TP	Pr	T	TP	Pr	Pr:all
JUnit 3.7	0	0	-	4	4	100%	1	1	100%	1	1	100%	0	0	-	100%
JHotDraw 6.0	1	1	100%	9	9	100%	2	1	50%	11	11	100%	1	0	0%	91%
Java AWT 1.6	1	0	0%	13	8	61%	6	3	50%	6	5	83%	9	7	77%	65%

[1] Total number of instances
[2] Number of identified True Positives
[3] Precision value

class, and a Template-method instance represents the template method itself. In our manual evaluation of the obtained results, we adopted precision measures – The number of correctly inferred pattern instances (True Positives) over the total number of inferred instances. In order to accept an instance as truly positive, this instance needs to be explicitly stated in the framework's documentation or there have to be strong indications found by inspecting source code and available comments. In all the tests performed, none of the inferred pattern instances violates any of the structural or behavioral restrictions put forward for that pattern in the axioms and rules; this means that the knowledge base accurately represents the source code and the reasoner acted accordingly. However, as noted in Table 1, there are a few cases in which precision suffers since the identified instances were considered by our standards as false positives – falsely inferred instances that were not designed as such. In all these cases, neither documentation nor source code comments strongly certified those instances as true positives.

Recall statistics, however, are usually harder to come up with since they require the identification of all false negatives – occurrences that were intended as actual pattern instances but were not inferred by the reasoner. Most framework documentations do not explicitly report on all actual number of pattern instances, thus manual identification through source code inspection is subject to one's interpretations of what constitutes an actual instance. Nevertheless, we found some evidence of false negatives. Some of the instances were missed due to our parser's inability to capture the complete picture of the running code; others are due to the restrictions specified in OWL axioms and SWRL rules. For example, the current rule for Template Method requires the primitive operation that is called by the template method to be declared *protected*, however, this is not a requirement; we found evidence of that on both JHotDraw and JUnit. It is obvious that such relaxation of the Template Method rule allows the reasoner to detect the missed instances. Generally speaking, relaxation reduces false negatives; it does, however, increase the risk of false positives.

It was evident in some cases that better capturing of behavior specified in method bodies through more effective parsing is indeed helpful, an appropriate parsing provides more flexibility when relaxing the requirements and certainly improves both precision and recall. Nevertheless, our approach to pattern detection distinguishes itself as being flexible and usable such that users can relax or restrict pattern descriptions as they wish. In Singleton, for example, accounting for another form of lazy instantiation or perhaps disputing the way the static reference is being declared, can be readily obtained by slightly modifying the rule.

A more comprehensive case study is currently being conducted. In this study we are investigating the effect of rule relaxation, applying our approach to other frameworks, comparing our results to other non-model driven approaches, and finally attempting to come up with recall statistics for selective well documented frameworks. Preliminary results are extremely encouraging and show an improvement However, a systematic study can uncover the overlap between different approaches and most fundamentally show the value of ontology-based modeling in software engineering.

Table 2 shows statistics related to the software frameworks used in our study as well as running time analysis.

Table 2. Framework statistics and time analysis. Time unit: Second

		JUnit 3.7	JHotDraw 6.0	Java AWT 1.6
Statistics	No. of Classes	99	377	549
	No. of OWL Individuals	1411	6254	11853
Processing Time	Parsing + Processing	3	6.5	13
Reasoner Time	Visitor	1.6	18	49.7
	Observer	2.9	33.3	95.2
	Composite	2.5	29.1	76
	Template Method	1.6	14.7	46
	Singleton	1.7	14	48

It is evident that the time required for parsing the code, loading the ontologies, executing the queries, etc. is in most cases minimal when compared to the time required by the reasoner to classify the ontologies and execute SWRL rules. Furthermore, since rules allow for reasoning with all available individuals, rule execution time is susceptible to increase as the number of OWL individuals in the ABox (assertions that represent ground facts for individual descriptions) as well as the complexity of the rules increase. In fact, Pellet, our experimentation reasoner, does not claim optimal performance. However, it does have an integrated rule engine that provides a complete and sound support for SWRL; that definitely adds to the cost of memory consumption and computational speed.

5 Related Work

On surveying the state of the art in design pattern recovery, we found numerous non-semantic based approaches. Most existing approaches differ in the detection technique utilized as well as the mechanism used for obtaining source code knowledge and formalizing pattern descriptions. PINOT [9] for example, represents design patterns using control-flow graphs which have some scalability issues. This tool relies on processing the Abstract Syntax Tree (AST) of method bodies to build a CFG for program elements. This CFG is then examined to verify the existence of restrictions related to a particular design pattern. Tsantalis et al. [10] proposed an approach that relies on graph and matrix representation of both, the system under study and design patterns. The ASM framework is used to parse the Java bytecode and populate the matrices for a particular system; using a well known similarity score algorithm, system's representation is matched with pattern descriptions that are hard-coded within their tool. Other approaches utilize information obtained through dynamic analysis of the running code [11]. In general, runtime analysis can be effective in detecting behavioral patterns only when the software is backed by a suitable and complete test data.

Whether the representation mechanism used is CFG, ASG, matrices, or DFG, it is our belief, however, that using a semantic-based formalism ensures consistent and accurate functional representation of design patterns, their participants, and the system under study. Furthermore, semantic representation allow for computing inferences of knowledge that was not explicitly stated, yielding an inference-based detection of design pattern instances.

Closer to our approach is the work of Kramer and Prechelt [12] and Wuyts [13] in which declarative meta-programming techniques were utilized. In particular, design patterns are described using variations of Prolog predicates and limited information about program elements and their roles are represented as Prolog facts. Most recently, the work presented in [14] and [15] explored the notion of utilizing OWL ontologies to structure the source code knowledge. However, these approaches provide template ontology descriptions or at best rudimentary ontologies that must be extended to become more expressive. The full potential of semantic-based modeling was yet to be explored. The primary focus of the work presented in [14] is basically to facilitate exchanging and sharing knowledge about patterns, anti-patterns, and refactoring. Our aim, however, is to provide a true semantic-based approach for design pattern detection and provide an aid for understanding, reasoning, and conceptualizing source code.

6 Conclusion and Future Work

In this paper, we proposed a semantic-based approach for software understanding. In particular, we illustrated our approach for recovering design patterns from source code. The proposed approach is fully automatic and distinguishes itself as being extensible and extremely usable. Moreover, the proposed approach is purely semantic-based that describes knowledge using a set of ontologies. SCRO, our main ontology, provides a consistent and complete functional description of program elements and allows for semantic inference of knowledge that was not explicitly stated.

The tool described in Section 4 is currently a work in progress. Our ultimate goal is to provide a more comprehensive program comprehension environment for conceptualizing, understanding, and recovering software knowledge to aid both reverse and forward engineering activities. Notably the generated knowledge by the bytecode parser may not provide the full picture of the running code; in particular, method bodies need to be effectively parsed to capture more detailed behavioral aspects of programs. We are currently investigating the use of other means to augment the current knowledge, the more knowledge captured, the more flexibility is given to the user in authoring the rules for detecting design patterns. We also investigating our approach on other design patterns found in literature and conducting more case studies as described in Section 4.1.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
2. Chikofsky, E.J., Cross, J.H.: Reverse engineering and design recovery: A taxonomy. IEEE Software. 7(1), 13–17 (1990)
3. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American. 284(5), pp. 34–43, (2001)
4. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Kartz, Y.: Pellet: A Practical OWL-DL Reasoner. Web Semantics: Science, Services and agents on the World Wide Web. 5(2) (2007)
5. Knublauch, H., Fergerson, R.W, Noy, N.F., Musen, M. A.: The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In: Third International Semantic Web Conference (ISWC 2004), pp. 229–243. (2004)
6. Source Code Representation Ontology (SCRO); Design Pattern Ontolgies; and an automatically generated ontology from parsing the JHotDraw application framework, `http://www.cs.uwm.edu/~alnusair/ontologies`
7. Motik, B., Grau, B.C., Sattler, U.: Structured Objects in OWL: Representation and Reasoning. In: 17th International Conference on World Wide Web, pp. 555–564. (2008)
8. McBride, B.: Jena: A Semantic Web Toolkit. In: IEEE Internet Computing, 6(6), 55-59 (2002)
9. Shi, N., Olsson, R.A.: Reverse engineering of design patterns from Java Source Code. In: 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 123–132. (2006)
10. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.: Design Pattern Detection Using Similarity Scoring. IEEE Transactions on Software Engineering. 32(11), 896–909 (2006)
11. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Behavioral Pattern Identification Through Visual Language Parsing and Code Instrumentation. In: European Conference on Software Maintenance and Reengineerig (CSMR'09), pp. 99–108. (2009)

12. Kramer, C., Prechelt, L.: Design Recovery by Automated Search for Structural Design Patterns in Object Oriented Software. In: 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 123–132. (2006)
13. Wuyts, R.: Declarative reasoning about the structure of object-oriented systems. In: Proceedings of TOOLS USA '98, pp. 112–124. (1998)
14. Dietrich, J., Elgar, C.: A Formal Description of Design Patterns Using OWL. In: Proceedings of the 2005 Australian Software Engineering Conference, pp. 243–250. (2005)
15. Kirasić, D., Basch, D.: Ontology-Based Design Pattern Recognition. In: 12th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES'08), pp. 384-393. (2008)