Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Rule-based Detection of Design Patterns in Program Code

Awny Alnusair[1⋆], **Tian Zhao**[2], **Gongjun Yan**[3]

[1] Indiana University, Kokomo, IN 46904, USA, e-mail: alnusair@iuk.edu
[2] University of Wisconsin-Milwaukee, Milwaukee, WI 53201, USA, e-mail: tzhao@uwm.edu
[3] University of Southern Indiana, Evansville, IN 47712, USA, e-mail: gyan@usi.edu

Received: date / Revised version: date

Abstract. The process of understanding and reusing software is often time-consuming, especially in legacy code and open-source libraries. While some core code of open-source libraries may be well-documented, it is frequently the case that open-source libraries lack informative API documentation and reliable design information. As a result, the source code itself is often the sole reliable source of information for program understanding activities. In this article, we propose a reverse-engineering approach that can provide assistance during the process of understanding software through the automatic recovery of hidden design patterns in software libraries. Specifically, we use ontology formalism to represent the conceptual knowledge of the source code and semantic rules to capture the structures and behaviors of the design patterns in the libraries. Several software libraries were examined with this approach and the evaluation results show that effective and flexible detection of design patterns can be achieved without using hard-coded heuristics.

Key words: Design Patterns – Design Recovery – Software Maintenance – Ontology Formalisms – Knowledge Representation – Semantic Inference

1 Introduction

Design patterns provide reusable solutions to common object-oriented design problems. They are viewed as a natural way of decoupling the bindings between system components so that systems can be more understandable, scalable, and adaptable [7]. The concept of design patterns in the context of object-oriented design was popularized by the Gang-of-Four (GoF) [15]. Since then, patterns have been widely used for documenting and structuring new software libraries [1] as well as comprehending and re-engineering existing legacy software.

During development, design patterns can be used to understand the libraries being reused and to speed-up the development process by providing tested design templates that can be used to solve design dilemmas. On the other hand, during the process of refactoring and maintenance, design knowledge can provide a better glimpse into design intent and thus establishes a better foundation for high-level understanding of the structure, organization, and the interaction between the components of the system being maintained.

Unfortunately, for some legacy systems and large open-source libraries, such design knowledge is not richly documented, which leaves the source-code itself as the only mean for capturing design intent. Therefore, the software engineering community has been interested in providing effective reverse-engineering approaches to analyze source code and recover the lost design rationale that can be depicted in the form of design patterns.

To that end, we propose an effective and usable reverse-engineering approach that is based on semantic techniques to detect design pattern instances from source code. This approach provides a formal, explicit, and semantic-based representation of the conceptual knowledge of source code. It is solely reliant on ontology modeling and semantic-based techniques from the emerging field of Semantic Web [4]. The fundamental hypothesis we explore in this article is that representing software knowledge using ontology formalisms and semantic techniques is effective in improving the precision of recovering design pattern information.

⋆ *Corresponding author. Awny Alnusair was partially supported by the Indiana University fellowship research grant*

[1] Although they have a few distinguishing features, we use the terms 'library', 'framework', and 'software system' interchangeably

1.1 Semantics-enabled Design Recovery

Semantic Web technologies have been a great asset in providing solutions for various domain specific problems. Ontologies are the backbone technology for formal and explicit representation of knowledge in the Semantic Web. Due to their formal reasoning foundation, ontologies can play an important role in domain engineering. One can use ontologies to structure and build a source-code knowledge base that can be used by software agents and serve as an effective base for semantic queries [13,26].

This research contributes to the proper linking of the established field of program understanding with the emerging field of Semantic Web by developing ontology models for the problem of design pattern recovery and semantic knowledge-representation. Our ontology model includes a Source Code Representation Ontology (SCRO). This ontology is created to provide an explicit representation of the conceptual knowledge structure found in source code. SCRO provides an effective base model for understanding the relationships and dependencies among source-code artifacts in a software system. Therefore, SCRO serves as a basis for design pattern recovery and can be utilized by other applications that require semantic knowledge at the source code level. Since SCRO is primarily used for building a software knowledge base, a set of parsing tools have been developed to instantiate the ontology with ontology instances that represent source code artifacts.

In the context of design pattern recovery, we have also designed and developed a design pattern ontology sub-model. This sub-model extends SCRO's vocabulary and includes an upper design-pattern ontology that is further extended with a specific ontology for each individual design pattern. This setup provides flexibility and transparency during the pattern detection process. The current state-of-the-art tools in design pattern recovery hardcode the descriptions and roles of individual design-pattern participants. This limits the usability and flexibility of such tools since users have no control over these descriptions. However, using ontology representation, design patterns can be specified externally within their respective ontologies and their participants are depicted using semantic rules that can be easily relaxed or fortified by users according to their needs. Therefore, we further hypothesize that ontology representation of software knowledge enables a flexible mechanism of design recovery through rule relaxation, which may detect more pattern instances.

Our earlier work [1] explores the notion of using semantic techniques for design pattern recovery. The work presented in this article delves into the details and extends the previous work in the following primary directions:

- We provide a refined description of the proposed approach that further investigates the issue of ontology modeling and how it can be properly used to detect design patterns.
- We present updated ontology models that include an enhanced source-code ontology as well as new design pattern ontologies.
- We include additional support for detecting a wider range of design patterns.
- We enhance the process of knowledge population with a new parser that captures additional aspects of source-code that allows higher levels of source-code analysis.
- We present a more detailed performance evaluation and empirical case studies using a new tool implemented as a plugin for the Eclipse platform.

In the following section, we provide the necessary background that is needed for understanding the proposed methodology. In particular, we introduce the concepts, languages, and Semantic Web technologies that are employed in our approach to program understanding and design recovery. Furthermore, this section provides a detailed discussion of our source-code ontology model and the process we use to instantiate this model and generate the knowledge base.

2 Ontology Modeling for Design Pattern Recovery

The research presented in this article seeks to leverage program understanding and design recovery using ontological modeling with the help of various Semantic Web techniques. Central to our approach is an ontology model that captures heterogeneous sources of conceptual knowledge found in source code by modeling the most important aspects of its internal structure. This kind of modeling serves as the basis for providing efficient and common access to relevant information resources. Once the ontology model is formally defined and represented using an appropriate taxonomy, the model needs to be automatically populated with ontological instances. These instances are the basic building blocks for constructing and populating a knowledge base that can be accessed through semantic queries. The following subsections provide an overview of ontology modeling and discuss the process of structuring and building a semantic knowledge base for source-code artifacts.

2.1 Ontologies and the Semantic Web

Ontology is a term originally used in Philosophy. It describes the nature of being or the kinds of existence and their basic categories and relationships. Computer scientists have borrowed this term and used it in many different areas, including Knowledge Engineering, Database Theory, Artificial Intelligence, Software Engineering, and Information Retrieval and Extraction. More recently, this

term has been exploited by the Semantic Web community. Scientists have provided many definitions for this term. The most widely-used definition is from Gruber [16]: "*An ontology is an explicit specification of a conceptualization*". In other words, an ontology provides a working strategy and a framework of general concepts, objects, properties, and other entities in a domain of discourse and the relationships that hold among the various domain concepts.

In technical terms, an ontology is a data model for a particular domain that consists of machine-interpretable definitions of classes that describe a set of concepts, interrelationships among these classes, structural properties of classes, and constraints expressed as axioms. Individuals (or instances) of classes are the basic concrete objects of an ontology. The individuals may or may not be part of an ontology but the ontology must provide a way for classifying these individuals.

Ontologies are the backbone of the Semantic Web. The vision of the Semantic Web [4] relies on structuring and organizing data in a way that makes it easy for people to understand and for machines, software programs, and agents to process. Due to their ability to enable automated knowledge sharing and understanding, ontologies are used as the knowledge representation component of the Semantic Web vision. To this end, various ontology modeling languages and Semantic Web technologies have emerged and are currently used in various domains. In this work, we utilize the Web Ontology Language (OWL) [25], the Resource Description Framework (RDF) [23], the Semantic Web Rule Language (SWRL) [27], and the SPARQL protocol and query language [29].

OWL is the standard web ontology language. It is used to create machine-understandable definitions of the precise meanings of domain concepts and to capture the relationship of the concepts. OWL-DL is a sub-language of OWL based on Description Logic (DL). Due to its decidability and completeness, OWL-DL has desirable computational properties for reasoning systems. OWL-DL's expressive power and reasoning support enable inferring additional knowledge and computing the classification hierarchy (subsumption reasoning). RDF is used as flexible and expressive data representation model suitable for describing resources and for formally representing machine-processable semantics of data. SWRL extends OWL expressive power with Horn rules that are similar to the Prolog or DATALOG rules. These rules can be combined with OWL knowledge bases for reasoning and computing entailments. Finally, SPARQL is a SQL-like query language and protocol for ontological querying of RDF metadata.

In what follows, we describe how these Semantic Web languages and technologies contribute to our approach. In particular, the next two subsections show how OWL-DL and RDF are used to obtain a precise formal representation of various source-code artifacts. These representations form the basis for semantic reasoning and inference of additional facts which can be retrieved using SPARQL semantic queries. Sect. 3 explores the SWRL language in more details and provides an overview of how semantic rules can be used to further extend OWL representations to handle features that cannot be expressed using OWL alone.

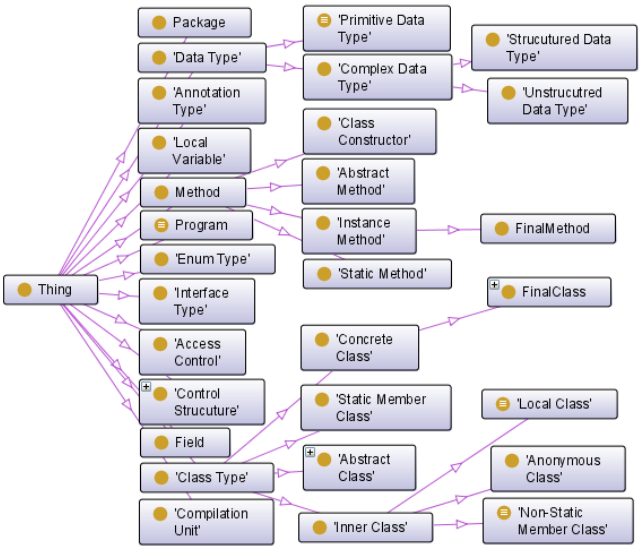

Fig. 1. An excerpt of the SCRO's taxonomy

2.2 Structuring the Knowledge Base

We have developed a Source Code Representation Ontology (SCRO) using OWL-DL constructs. This ontology is created to support major program understanding tasks by explicitly representing the conceptual-knowledge structures found in source code. It serves as a base model for understanding the relationships and dependencies among source-code artifacts. SCRO captures major concepts and features of object-oriented programs including:

- encapsulation mechanisms,
- class inheritance and interface implementation,
- aggregation relationship between objects
- type and subtype information including nested and anonymous classes,
- method overloading and overriding,
- method signatures and invocations,
- access control mechanisms,
- language-specific features such as enumeration and annotation, and
- control structures (repetition, selection, and sequence controls).

A partially expanded fragment of the ontology's taxonomy is shown in Fig. 1 and the complete ontology can be found online [2].

SCRO's knowledge is represented using the OWL-DL ontology language. Each of the ontology *classes* shown

in Fig. 1 corresponds to a *concept* in the object-oriented programming domain. Collectively, these classes form the taxonomy tree of the ontology where *owl:Thing* is the root (superclass) of this subsumption hierarchy. An ontology class is interpreted as a set that defines a group of individuals sharing some properties. Disjoint subclasses are also modeled in SCRO to define different sets of individuals. For example, all individuals that are members of the class `InstanceMethod` are necessarily members of class `Method` since `Method` subsumes `InstanceMethod` but none of these individuals can be simultaneously a member of another subclass of `Method`.

SCRO is precise, accurate, well-documented, and carefully designed with ontology reuse [35] in mind. SCRO is available online, which allows researchers to reuse or extend its representational components to support any semantic-based application that requires source-code knowledge. SCRO was created using Protégé [2] ontology editor and verified using Pellet [33] OWL-DL reasoner. In addition to performing different forms of ontology checking such as consistency and subsumption checking, the reasoner is primarily used to infer additional information that is not explicitly stated within the ontology using the set of asserted ontology facts and axioms.

Various object properties, sub-properties, data properties, and ontological axioms are defined within SCRO to represent relationships among concepts by linking individuals from different OWL classes. A subset of SCRO's object properties is shown in Fig. 2.

For example, the object property `hasOutputType` is a functional property defined for the return type of a method and `hasSuperType` is a transitive object property defined with two transitive sub-properties for class inheritance and interface implementation. OWL restrictions are defined to represent single inheritance in Java by limiting the maximum cardinality to one while allowing multiple inheritance for Java Interfaces.

Inverse properties are used to define the inverse relation. For example, the `isLocalVariableOf` is the inverse property of `hasLocalVariable`. This feature is particularly 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 reasoner systems.

In order to properly describe a programming language class, we need to describe the various fields defined in the class. We achieve this by using the property hierarchy of `hasField` and its inverse property `isFieldOf`. Method overriding is also defined using the functional object-property `methodOverrides` and its inverse property `methodOverriddenBy`.

In order to represent aggregation (Whole-Part) relationships between objects, we include the object property `hasPart` and its inverse `isPartOf`. This setup cleanly describes those composite objects that are built from other constituent objects. The `hasPart` property is a sub-property of `use`, which describes the 'use' relationship for class objects that need to be aware of other objects to carry out their operations.

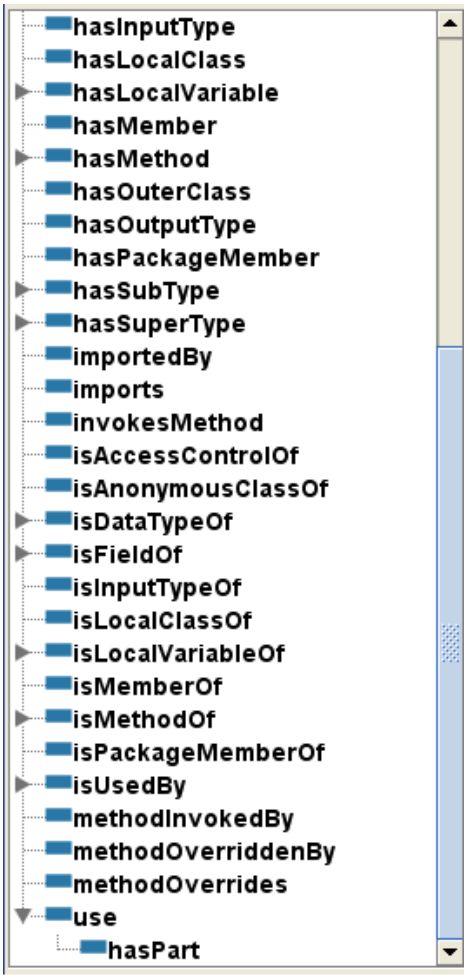

Fig. 2. Excerpt of SCRO object properties

2.3 Ontology instantiation and knowledge population

After having the ontology structure created with ontology concepts and their interrelationships specified, we need to create a knowledge base that facilitates inferencing and contains ontological instances (OWL individuals) that represent various concepts in the ontology. These instances are the building blocks of a knowledge base that can be used for browsing and querying. An instantiated concept (instance) is a concrete piece of knowledge. Thus, knowledge population is equivalent to instance generation that can be achieved by annotating the raw data source using predefined ontologies [6].

When dealing with a large source-code repository such as a complex framework, a large number of instances are usually created. Therefore, *automatic* text-to-model transformation and knowledge population are essential. Inspired by the Java RDFizer idea from the

[2] http://protege.stanford.edu/

Simile project [3], we have built a subsystem that automatically extracts knowledge from Java bytecode. The original RDFizer was capable of distinguishing only class types, super types, and basic *use* relations represented in a Java class file. Currently, our subsystem performs a comprehensive parsing of the Java class files specified by the Java Virtual Machine. It captures every SCRO concept that represents a source-code element and effectively generates instances of all ontological properties defined in SCRO for those program elements. Our knowledge generator sub-system distinguishes itself as not being relied on the existence of source code to extract knowledge. It parses the Java bytecode instead. This is especially helpful for understanding a software system without source programs. In the next subsection, we show a sample output of our knowledge extractor subsystem and explain some essential concepts.

2.4 Example of OWL Knowledge Base

OWL is essentially a vocabulary extension of RDF. Both OWL and RDF serve as standard formats for the sharing and integration of data and knowledge [19]. However, when creating an OWL knowledge base, we need to provide a clean separation of the explicit OWL vocabulary with its associated schema definitions from the metadata encoded in RDF. Therefore, the semantic instances generated by the knowledge generator subsystem are serialized using RDF and linked to SCRO or any other OWL ontology (e.g. a design pattern ontology) via OWL reuse mechanisms.

To this end, the knowledge generator parses a given library and captures all structural descriptions of its program elements. These descriptions are then used to generate a separate RDF ontology that is compliant with SCRO's description of these program elements. This process amounts to instantiating an OWL knowledge base for the target framework. The generated RDF ontology is represented using Notation3 [4] (N3) syntax, which is a compact and readable serialization of RDF data models.

Listing 1 shows a partial RDF description of a Java interface obtained through parsing the JHotDraw [5] library – a Java framework for creating and manipulating structured 2D vector graphics.

```
@base  <http://www.indiana.edu/.../scro-kb.n3>.
@prefix scro: <http://www.indi.../scro.owl#>.
@prefix rdf:  <http://www.w3.../rdf-syntax-ns#>.
@prefix owl:  <http://www.w3.org/2002/07/owl#>.

<http://www.indi.../scro-kb.n3> owl:imports
 <http://www.indi.../scro.owl> .

<#org.jhotdraw.framework.Figure>
 rdf:type scro:InterfaceType ;
 scro:hasAccessControl scro:public ;
 scro:hasSuperType
  <#org.jhotdraw.util.Storable>;
 scro:use <#org.jhotdraw.framework.Connector>;
 scro:hasAbstractMethod
  <#org.jhotdraw.framework.Figure.draw[....]>;
          ....
          ....
<#org.jhotdraw.framework.Figure.draw[....]>
 rdf:type scro:AbstractMethod ;
 scro:hasOutputType <#void> ;
 scro:hasInputType <#java.awt.Graphics> ;
 scro:hasSignature
   "org.jhotdraw.framework.Figure.draw[...]" ;
 scro:hasAccessControl scro:public  .
          ....
          ....
```

Listing 1. Partial KB representations and RDF descriptions of JHotDraw

RDF refers to resources using the Internationalized Resource Identifier (IRI), which is a generalization of the Uniform Resource Identifier (URI). The PREFIX clause associates a prefix label with an IRI and serves as a local namespace abbreviation for that IRI. The rest of the statements shown in Listing 1 are descriptions of resources. In RDF terminology, a RDF *statement* is known as a *triple* that consists of a *subject*, a *predicate*, and an *object*. The subject refers to the resource being described and the predicate expresses a relationship between the subject and the object. An object can be either another resource or a literal value. For example, in the following triple,

```
<#org.jhotdraw.framework.Figure>
                rdf:type  scro:InterfaceType
```

the interface `Figure` is the subject, the ontology property `rdf:type` is the predicate, and the ontology class `scro:InterfaceType` is the object, where the prefix `scro` indicates that `scro:InterfaceType` is defined in the SCRO ontology. RDF triples are used to assert facts in the knowledge base. For example, the above triple asserts that `Figure` is indeed an individual of type `InterfaceType`. In Description Logic (DL) terminology, a collection of axioms that represent fact-assertions about concrete resources in the knowledge base is called Assertion Box (ABox). A set of axioms asserting constraints on the domain's vocabulary and its structure is called Terminology Box (TBox). ABox assertions are compliant with TBox assertions and together they make up the knowledge base.

The underlying data structure of RDF is a labeled and directed graph that is made from the set of triples in the knowledge base. Therefore, each node-edge-node in the RDF graph represents a triple. Fig. 3 shows partial graph representation of the RDF statements in Listing 1. This kind of graph representation is used as a basis where semantic queries are executed against its corresponding knowledge base.

[3] http://simile.mit.edu
[4] http://www.w3.org/DesignIssues/Notation3
[5] http://www.jhotdraw.org/

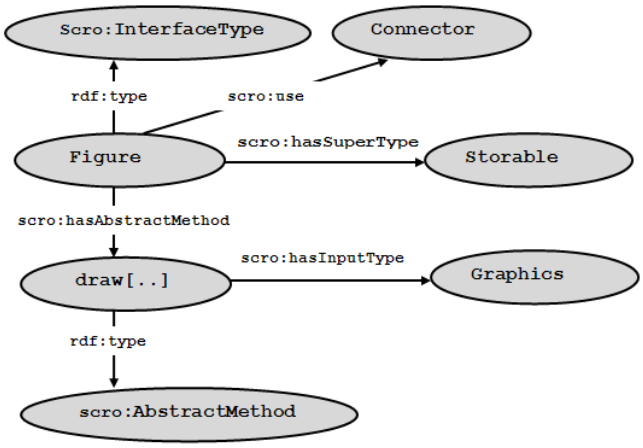

Fig. 3. Partial RDF triple representation of JHotDraw

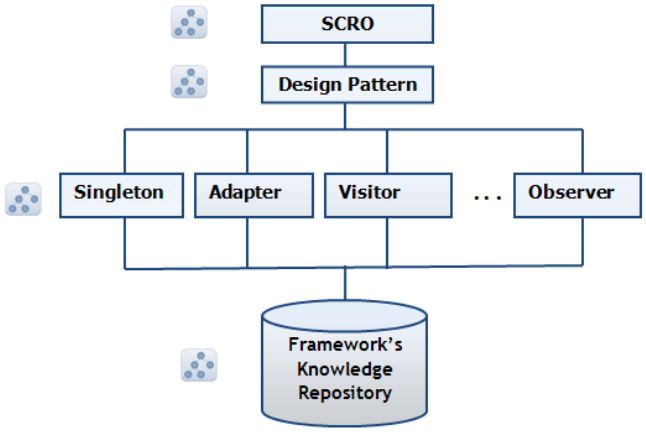

Fig. 4. Modular structure of ontology models for design recovery, linked via `owl:imports`

The following section provides a detailed discussion of how RDF descriptions are utilized in our approach. It further explains how our design pattern ontology submodels are used with the RDF knowledge base to enable a semantic-based detection of design patterns.

3 Design Pattern Recovery

Design pattern recovery is inherently a complicated task. This is due to the fact that patterns are basically *abstract* solution templates for common design problems. Therefore, when implemented in code, a design pattern's structure or behavior can take different forms. For example, when implementing the composite pattern in Java, some systems may use customized data structure instead of the collection classes in JDK library for maintaining object composition. When implementing visitor, state, and strategy patterns, some systems may favor delegation-based technique over the other alternatives.

Due to this abstract nature, design patterns cannot be effectively detected unless there exist formal, semantic, and explicit representations of both the conceptual knowledge of software artifacts and the collaborating participants as well as their roles in design patterns. The detection mechanism can be seen as a matching between the two explicit representations. This matching forms the basis for effective querying and successful retrieval of pattern instances. In the next subsection, we explain such *ontological* representation of design patterns.

3.1 Design Pattern Ontology Models

A proper representation of a design pattern needs to formalize its structural and behavioral description. This description includes the patterns' participating entities, their instances, roles, and collaborations. Therefore, we formally specify these aspects by reusing the vocabulary defined in SCRO and build definitions for each design pattern. The result is a modular and extensible structure of OWL ontologies linked together via the regular OWL reuse mechanisms – `owl:imports`. This structure is depicted in Fig. 4.

Essentially, when an ontology imports another ontology, all classes, properties, axioms, and individuals defined in the imported ontology are also available for use in the importing ontology. Fig. 4 shows that SCRO is directly imported by a design-pattern ontology. This ontology describes knowledge common to all design patterns, including concepts that classify design patterns into behavioral, creational, structural, and other categories. Further down in the hierarchy, a separate ontology is created for each design pattern that describes its essential participants and their properties, collaborations, restrictions, and the corresponding SWRL rules needed to detect this pattern. This modular structure promotes ontology reuse and allows SCRO to be used for various maintenance-related activities and even with reasoners that do not support SWRL rules.

At the bottom of the reuse hierarchy, we place the RDF knowledge repository for a given software library. This repository is basically the triple store that we obtained via parsing and knowledge extraction as we described in Sect. 2.3 and Sect. 2.4. Since the ontology-import relationship is transitive, all semantic conditions defined in the upper-level ontologies are now available in the RDF repository. Therefore, this inference-ready repository now contains all the knowledge and semantic descriptions that we need for effective design recovery.

3.2 Design Pattern Detection

As pointed out earlier, when design patterns are implemented in code, they can take different structural and behavioral forms. This variation in implementation styles complicates the detection process. However, the design patterns' overall structures and the roles of their participants and interactions should be followed to pre-

serve the intent and applicability of the patterns. In our approach, we aim at providing flexibility and transparency such that patterns are specified externally using ontology formalisms and participants' responsibilities are depicted using semantic rules that can be easily understood and modified. After defining SCRO and the design pattern ontologies, for each design pattern, we define the SWRL rules that formally describe the pattern's structure and behavior. Once these rules are defined, they are automatically imported into the repository for the framework in question. All what is needed next is a domain-independent OWL-DL reasoner that is capable of computing entailments from the set of facts and SWRL rules defined in the ontologies.

Design patterns were classified by GoF into three different groups: structural, behavioral, and creational. Structural patterns deal with composition of classes and objects in order to find the simplest ways to realize the relationships between these entities. Behavioral patterns deal with ways in which classes or objects interact in order to find the best communication styles among these entities. Creational patterns deal with object creation mechanisms in order to create suitable objects for a given situation. To test the validity of our approach, we have written SWRL rules for eleven well-known patterns that span the three pattern groups. The rest of patterns can be easily supported as well. In the following three sections, we illustrate our detection mechanism using one pattern from each group as examples.

3.3 Detecting the Composite Design Pattern

Composite is a structural design pattern that is intended to reduce design complexity by allowing primitive objects and complex composition of objects to be treated uniformly. It composes objects into a tree structure with one hierarchy of participants so that leaf objects and composite objects in this hierarchy are treated equally. The structure of Composite using the Unified Modeling Language (UML) is depicted in Fig. 5.

Despite the fact that there is a formal structure for each of the design patterns, there usually exists many acceptable implementation variations. This phenomenon is a major obstacle for design pattern recovery and it can cause a large number of false positives. However, since it is impossible to account for all implementation variations, we provide flexibility such that users of our tool can formally define a variation and then refine it as needed. Below is an example variant composed of two sets of conditions for detecting the composite pattern. For simplicity, we used a *simplified* set of conditions and SWRL rules for the composite pattern. Additional variations can be easily specified and formalized by using the same procedure. Some of these variations will be discussed, other possible restrictions including even complex behavioral ones can be easily added to capture other debated aspects of this pattern.

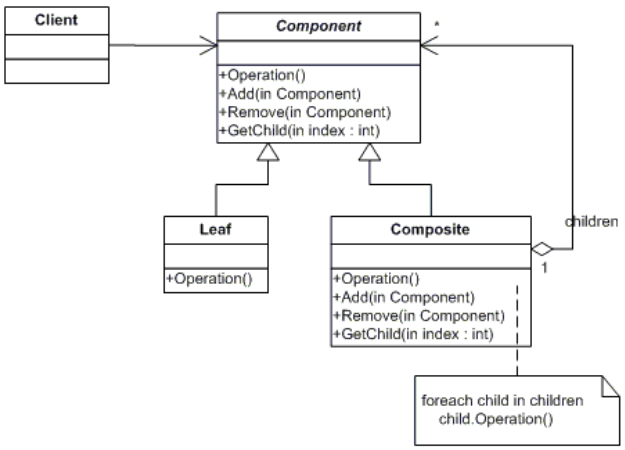

Fig. 5. Structure of the Composite design pattern in UML

1. The three participants of the composite pattern (leaf, composite, and component) fall on the same inheritance hierarchy. The component participant can be a direct or indirect super type of the other two participants. Furthermore, the component can be either a class or an interface. The operation defined in component should be overridden or implemented by the composite and the leaf classes.
2. The composite class is basically a component that maintains a collection of child elements, typically stored in a field that represents the composite pattern instance. This container field must have a structured datatype whose elements are not primitive. This includes arrays, built-in collection classes, and user-defined data structures. The datatype of the container must also provide child-manipulation methods for adding or removing children. In particular, the *add* operation must have a parameter of the component type thus allowing both leaf and composite components to be added to the composition. According to Fig. 5, the child-manipulation methods are also included in the main component interface. However, in order to avoid inflating leaf objects, recent implementations of this pattern tend to push them down in the composite class.

Preferably, we would like to use OWL-DL to formalize the above constraints. However, Description Logic is not expressive enough to encode these constraints. This is due to the fact that OWL and its DL support can only handle descriptions of infinite number of unstructured objects connected in a tree-like manner [24]. Therefore, we use SWRL to handle all non-tree-like situations and property chaining for design pattern constraints. SWRL extends OWL-DL with First Order Horn-like rules that are specified in terms of OWL classes, properties or individuals. A rule in SWRL is a logical statement about the ontology with two distinct parts: the antecedent (body) part and the consequent (head) part, each of which con-

tains 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)`, where both `var1` and `var2` are variables over OWL individuals in the knowledge base. Since each SWRL rule is considered an implication between the antecedent and the consequent parts of the rule, the reasoner will carry out the actions specified in the consequent part (i.e. asserting the consequent part) only if all the atoms in the antecedent are satisfied.

The following example shows a simple SWRL rule, where the antecedent part of the rule has a unary atom and another binary atom and the consequent part has only one unary atom.

```
ClassType(?aClass) ∧
hasAbstractMethod(?aClass, ?aMethod)
        ⇒
AbstractClass(?aClass)
```

Upon executing the above rule, a reasoner would classify every knowledge-base instance of type `ClassType` that happens to have an abstract method to be an instance of type `AbstractClass`. This process infers new facts and enriches the knowledge base with additional knowledge about these OWL individuals.

Listing 2 shows a sample of three SWRL rules for detecting the composite pattern. Please refer to SCRO and the Composite ontology for definitions of the OWL classes and properties used in this rule.

```
--Rule#1: Identifying Leaf Candidates:

scro:hasSuperType(?leaf, ?component) ∧
scro:hasMethod(?component, ?op-component) ∧
scro:methodOverriddenBy(?op-component,?op-leaf)∧
scro:isMethodOf(?op-leaf, ?leaf)
   ⟹
composite:hasLComponent(?leaf, ?component) ∧
composite:hasLeafOp(?leaf, ?op-leaf)  ∧
_____________________________________________

--Rule#2: Identifying Composite Candidates:

composite:hasLComponent(?composite, ?component)∧
scro:hasField(?composite, ?container) ∧
scro:hasStructuredDataType(?container, ?ctnrDT)∧
scro:hasMethod(?ctnrDT, ?insert) ∧
scro:methodInvokedBy(?insert, ?add-component) ∧
scro:isMethodOf(?add-component, ?composite) ∧
scro:hasInputType(?add-component, ?component)
   ⟹
composite:hasContainer(?composite, ?container)∧
composite:hasComponent(?composite, ?component)∧

_____________________________________________

--Rule#3: Identifying the Composite Pattern:

composite:hasLComponent(?leaf, ?component) ∧
composite:hasContainer(?composite, ?container)∧
composite:hasComponent(?composite, ?component)
   ⟹
composite:hasCompositeClass
                        (?container, ?composite)∧
composite:hasComponentClass
                        (?container, ?component)∧
composite:hasLeafClass(?container, ?leaf)
```

Listing 2. Sample SWRL rule specifications for detecting the Composite design pattern

The first rule in the listing broadly formalizes the first set of conditions that we identified for detecting this pattern. It identifies a candidate for the leaf class and the component class. In SCRO, the object property `hasSuperType` is made transitive with two transitive sub-properties. Namely `inherits` for type inheritance and `implements` for interface implementation. This transitivity allows the reasoner to infer all direct or indirect supertypes of a given class. In Composite, this transitivity allows the leaf to be a subtype of the composite class. The operations defined in the component class should be implemented by the composite and leaf classes. In SCRO, the object property `methodOverriddenBy` and its inverse property, `methodOverrides` apply to both classes and interfaces. In the consequent part of the rule, `hasLComponent` is defined in the composite ontology to identify individuals who are either leaf candidates or component candidates and assign those individuals to `LeafCandidate` and `ComponentCandidate` ontology classes that are defined within the composite ontology. The second atom identifies the candidates for the *operation*.

The second rule formalizes the second set of conditions. It extends the first rule with more constraints in order to identify a composite class candidate. Since all constraints identified in the first rule for a leaf also apply to a composite, the first atom in the second rule initially collects all leaf candidates and make them composite candidates. It further refines these candidates with more constraints. The composite class maintains a collection of child elements, typically stored in a field that represents the composite pattern instance and this container field has a structured data type. In SCRO, structured types are types whose elements are not single data items. Furthermore, this structured type must provide a *insert* method that can be invoked by the composite's *add* operation to add additional children into the composition. Finally, the last atom ensures that the *add* operation stores only objects of type component.

Other child-management methods can be implemented the same way. Note that child-management methods must be implemented in the composite class and we choose not to require the component class to declare those methods and leave that decision to the users.

The third rule ensures that composite effectively delegates behavior to its components. It also maximizes program understanding by identifying and storing all the participants of this pattern.

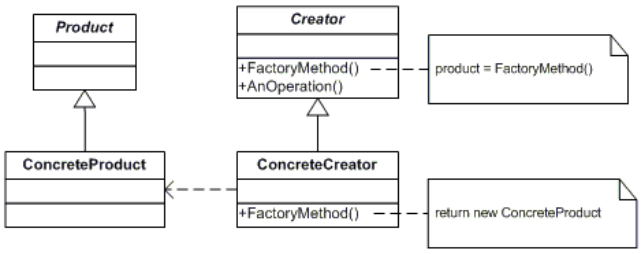

Fig. 6. Structure of the Factory Method design pattern in UML

3.4 Detecting Factory Method

Factory Method is one of the most commonly used creational patterns. The original structure of this pattern is depicted in Fig. 6. Factory Method defines an interface for creating objects (products) without specifying the exact class that will be instantiated. Instead, it lets subclasses decide what types of objects to instantiate.

Although the idea of Factory Method is simple, detecting this pattern is very challenging. This is due to the many different implementation variations of this pattern. Again, our approach is flexible enough for users to formulate customized constraints for this pattern. Below, we provide two sample conditions for detecting the Factory Method pattern:

1. Creator declares the factory method. This method is either an abstract or an instance method, which returns an object of type Product. ConcreteProduct is a concrete subtype of Product.
2. ConcreteCreator is a concrete subtype of Creator, which provides a concrete implementation of the factory method. This factory method overrides its counterpart in the base factory and returns an instance of a ConcreteProduct.

Listing 3 shows a sample SWRL rule that depicts the above two conditions. Atoms 1-4 and atoms 5-10 in the listing formalize the first condition and the second condition, respectively. The first unary atom in the rule ensures that ConcreteProduct is a concrete class so it can be instantiated by ConcreteCreator. The second atom ensures that a ConcreteProduct is a direct or indirect subtype of the base Product.

In this rule, we neither require the factory method in Creator to be an abstract method, nor do we make a distinction whether the base factory is a concrete class, an interface, or an abstract class. These are some of the variations of the Factory Method pattern [15]. The factory method in Creator can indeed be an instance method returning a reasonable default concrete product. We have found evidence of a concrete factory method in Creator when analyzing sample frameworks. For example, in JHotDraw, the method *connectorAt()* is defined in the *AbstractFigure* class to return the default product (*ChopBoxConnector*) while the actual factory method in the concrete class *PolyLineFigure* returns an instance of *PolyLineConnector*.

Atoms 5-10 depict the second condition almost verbatim. The factory method instantiates an instance of ConcreteProduct. This behavior is formalized by requiring a method call for a ConcreteProduct's constructor from within the factory method. The consequent part of the rule identifies each participant of this pattern.

Other variations for implementing this pattern do exist. For example, Creator may also declare another method (*AnOperation*), which calls the factory method to provide a product object. From our experiences with frameworks, this is a a uncommon case but it can certainly be added as an additional restriction. Furthermore, one might want to ensure that there are two distinct hierarchies, one for ConcreteCreator and another for ConcreteProduct. This can be achieved in SWRL by using the *differentFrom* symbol as follows:

```
differentFrom (?baseCreator, ?baseProduct)
```

The flexibility of SCRO's object properties allows us to modify the rules easily in order to enforce more constraints or relax others. We will see a concrete example of rule relaxation in the following section.

3.5 Detecting Visitor

Visitor is a behavioral design pattern that allows external operations to be performed on the elements of an object structure while not modifying the classes of the elements [15]. The idea is to keep the object structure intact by defining new structures of visitors representing the new behaviors of interest. As shown in Fig. 7, this pattern defines two separate class hierarchies: the Visitor hierarchy and the Element or Host hierarchy. 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 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.

Listing 4 shows a sample SWRL rule that depicts the above three conditions for detecting Visitor. Ev-

```
1.  scro:ConcreteClass(?concreteProduct) ∧
2.  scro:hasSuperType(?concreteProduct, ?baseProduct) ∧
3.  scro:hasMethod(?baseCreator, ?fMethodBase) ∧
4.  scro:hasOutputType(?fMethodBase, ?baseProduct) ∧

5.  scro:ConcreteClass(?concreteCreator) ∧
6.  scro:hasSuperType(?concreteCreator, ?baseCreator) ∧
7.  scro:hasInstanceMethod(?concreteCreator, ?factoryMethod) ∧
8.  scro:methodOverrides(?factoryMethod, ?fMethodBase) ∧
9.  scro:hasConstructor(?concreteProduct, ?cpConstructor) ∧
10. scro:invokesMethod(?factoryMethod, ?cpConstructor)

    ⟹

11. FactoryMethod(?factoryMethod)  ∧
12. hasBaseFactory(?factoryMethod, ?baseCreator) ∧
13. hasConcreteFactory(?factoryMethod, ?concreteCreator) ∧
14. hasBaseProduct(?factoryMethod, ?baseProduct) ∧
15. hasConcreteProduct(?factoryMethod, ?concreteProduct)∧
```

Listing 3. A sample SWRL rule for detecting the Factory Method pattern

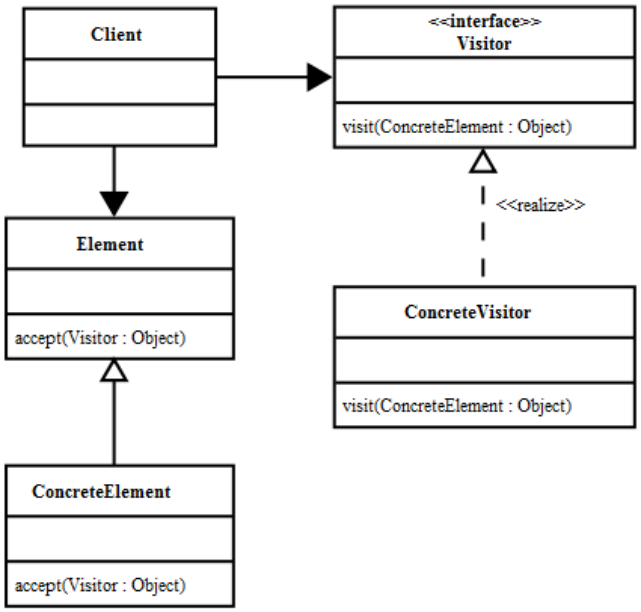

Fig. 7. Structure of the Visitor design pattern in UML

ery constraint is formalized using three different atoms. The `hasInputType` OWL object property represents a method's formal parameter and the `methodOverrides` property applies to both method overriding and interface method implementation. Upon classifying the Visitor ontology and its underlying knowledge base, a reasoner with a rule engine would infer and 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 4. Sample SWRL rule for detecting Visitor

Relaxing or adding more constraints to the requirements is relatively simple. For example, one might want to retrieve pattern instances that declare a supertype for all concrete hosts in the *Host* hierarchy. This can be accomplished by modifying the third condition and introducing a fourth condition as follows.

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 the double dispatching mechanism by calling the matching *visit* method and passes the host in to the visitor.
4. At the root of the Host hierarchy there is an interface or abstract class type. It represents the super type of all concrete hosts. It declares the abstract *hook* method, which takes Visitor as an argument.

The result of these modifications is a new rule depicted in Listing 5.

```
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 5. Modified SWRL rule for detecting Visitor

4 Tool Implementation and Evaluation

We have implemented the SEMantic PATtern RECovery (Sempatrec) approach in a tool developed as a plug-in for the Eclipse IDE. As described in Sect. 2.3, Sempatrec automatically processes the Java bytecode for a software library, generates a RDF ontology, and stores the ontology locally in a pool of available repositories. This pool is maintained and managed using the Jena library [6] –an open-source framework for building Semantic Web applications. Once this process is completed, Sempatrec utilizes Pellet to provide classification, rule execution, and reasoning services over the knowledge base. Fig. 8 shows a snapshot of Sempatrec's main views in the Eclipse workbench.

The main view provided by Sempatrec is the pattern-detection view, which allows the user to select a repository (i.e. the RDF ontology for a given framework). Once the desired ontology is selected, Sempatrec automatically loads the required ontologies, invokes the reasoner to compute the subsumption class hierarchy, executes the rules, and runs a series of built-in SPARQL queries to capture and display the detected pattern instances. Sempatrec presents the results in an intuitive way: it uses the descriptions found in the respective ontology for each design pattern to capture and present the details of each detected pattern instance, which includes the participants as well as their relationships and roles in the design pattern.

Sempatrec uses either the built-in or user-customized constraints for each supported design pattern. The customized constraints can be formulated through a slightly modified version of the SWRL rules that we used for built-in constraints. Currently users cannot use Sempatrec to edit the built-in rules. Instead, they can edit the rules using a specialized ontology editor such as Protégé and then load them into Sempatrec.

In addition to providing the main pattern-detection view, Sempatrec also provides other views for visualizing the upper-level ontologies and for understanding their conceptual structure. The ultimate goal for Sempatrec is to provide a comprehensive environment for supporting library comprehension activities. Currently it provides a view for editing and running SPARQL queries against the knowledge base. This view can be used for various interrogations and search queries against the generated knowledge base. It can also be used for capturing structural descriptions of less complex design patterns such as Template-Method. A sample query that specifies the constraints of Template-Method using SPARQL conditions is shown in Listing 6.

```
PREFIX scro: <http://www.indi.../scro.owl#>
SELECT distinct ?tMethod ?tClass ?subClass
WHERE {
    ?tClass  a  scro:AbstractClass .
    ?tClass  scro:hasInstanceMethod ?tMethod .
    ?tClass  scro:hasAbstractMethod ?op1 .
    ?op1   scro:hasAccessControl scro:protected.
    ?tMethod   scro:invokesMethod ?op1 .
    ?subClass  scro:inherits ?tClass .
    ?subClass   scro:hasInstanceMethod ?op2 .
    ?op2   scro:methodOverrides ?op1 .
}
```

Listing 6. Sample SPARQL query for retrieving Template-Method instances

Sempatrec provides a facility for performing ontology-based search [20] over the repository to retrieve various kinds of source-code artifacts and explore their structural relationships. The structure provided by the ontologies and the generated knowledge can be examined to reason about the framework's artifacts and formulate search queries. For example, consider a developer who is reusing the JHotDraw framework and wishes to find all parameterless static methods that return an instance of interface *FigureEnumeration*. Listing 7 shows the sample SPARQL query that answers this question.

```
PREFIX scro:<http://www.indi.../scro.owl#>
PREFIX JHDraw:<http://www.indi.../JHotDraw.n3#>

SELECT distinct ?sm
WHERE {
  ?sm a scro:StaticMethod.
  ?sm scro:hasOutputType
    JHDraw:org.jhotdraw.frame.FigureEnumeration.
  OPTIONAL {?sm scro:hasInputType ?it}
  FILTER (!bound(?it))
}
```

Listing 7. Sample SPARQL search query

[6] http://jena.apache.org/

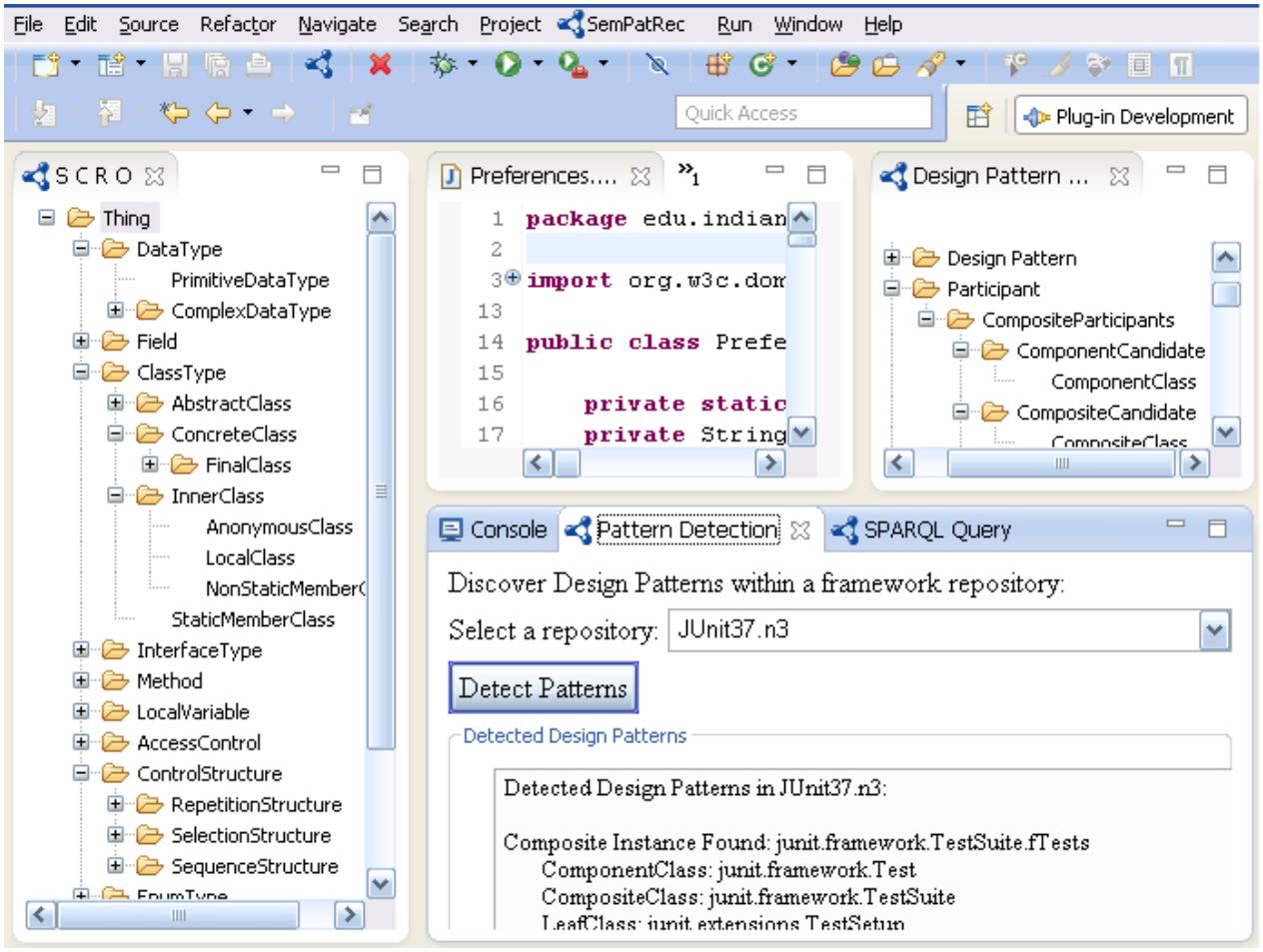

Fig. 8. Sempatrec snapshot: showing the detection view and part of SCRO and Composite taxonomies

4.1 Experimentation with Design Pattern Detection

In this section, we report our experiments for assessing the benefits of Sempatrec. In particular, we report the results of analyzing three open-source frameworks. We also compare Sempatrec's results with the state-of-the-art approaches in design pattern recovery. The hypotheses that we test in the experiments include:

H 1 *Ontology-based representation of source-code knowledge improves precision of pattern recovery when compared to other static analysis approach (i.e. less false positives).*

H 2 *Ontology-based representation of source-code knowledge provides flexibility through semantic rule relaxation, which can be effective in improving recall of design pattern instances (i.e. less false negatives).*

To some extent, the context in which these hypotheses are tested is subject to our interpretation of what makes a good experiment. Unfortunately, this phenomenon is the case for all pattern recovery tools because of the lack of independent benchmark frameworks and trusted baselines that can be used to evaluate and compare different approaches. However, there are a few well-known frameworks that are created with design patterns in mind and come with quite informative design pattern documentation. Therefore, these frameworks are usually used for performance evaluation purposes by many pattern detection tools. We have conducted our experiments on three of these frameworks: JHotDraw (release 5.1), JUnit[7] (release 3.7), and JRefactory[8] (release 2.6.24) – a tool for refactoring the internal structure of Java applications. The frameworks vary in size, represent different domains, and are known to have instances of the GoF design patterns. Also, they were used for evaluating other comparable approaches, which makes them a good fit for our purpose. Table 1 shows statistics of these frameworks and the generated knowledge base.

For ease of presentation, we first discuss the results of detecting the creational and structural design patterns that are supported by Sempatrec. This experiment also investigates the effects of rule relaxation and discusses the results reported by other two approaches – DeMIMA and SSA. DeMIMA [18] is a multi-layered structural approach that relies on analyzing source-code structures

[7] http://www.junit.org
[8] http://jrefactory.sourceforge.net/

Table 1. Statistics about frameworks used in evaluating Sempatrec

	JUnit v3.7	**JHotDraw v5.1**	**JRefactory v2.6.24**
Number of processed class files	99	172	570
Number of methods	559	1362	4998
Approximate KLOC	3.1	8.4	60
Number of generated OWL individuals	1411	2428	8565
Number of generated RDF triples	10546	23123	87187
Size of the RDF ontology	526KB	1.28MB	5.31MB

and utilizes explanation-based constraint-programming to identify design motifs. The Similarity Scoring Approach (SSA) [34] identifies patterns and their variants using a scoring algorithm to find a match between graph representation of both pattern descriptions and the software system at hand. We chose to compare our results with both DeMIMA and SSA for multiple reasons. Firstly the results obtained by both tools are relatively more effective than other tools. Secondly, these tools were the most closely related tools with readily available results. Thirdly, to some extent these tools support same patterns we use in our studies.

Later in the discussion, we present the results of detecting behavioral design patterns after adding a behavioral-specific approach, DPRE [9], to the set of comparable approaches. DPRE is a two-phase approach that consists of static as well as dynamic analysis for detecting behavioral design patterns. The novelties of these approaches are discussed in details in Sect. 5.

Table 2 shows instance detection statistics for creational and structural patterns, which are reported by DeMIMA, SSA, and the results of Sempatrec. For SSA, the table shows updated results found on the SSA web site after its initial publication. Table 3 shows the results for behavioral patterns as well as the results reported by DPRE.

The interpretation of a pattern instance in Table 2 and Table 3 can be derived from the corresponding rules for that pattern. For example, a Composite instance represents the child element's container, a Decorator instance represents the decorator class, an Observer instance is the *notify* method, a Visitor instance is the *interface* found at the root of the Visitor hierarchy, and an Adapter instance refers to object adapter which relies on composition as opposed to class adapter which relies on multiple inheritance. The case for SSA in some cases is a bit different. SSA counts the template class as a Template Method instance and the base factory as a Factory Method instance as opposed to the template method and the factory method in Sempatrec, respectively. SSA and DPRE also regards an Observer instance as the subject class as opposed to the *notify* method. Therefore, in order to make the experimental results of different approaches comparable, we have adjusted our results for these patterns to match those for SSA and DPRE. For example, in JHotDraw, Sempatrec returned a total of six Factory Method instances (i.e. six factory methods) but we reported a total of three instances (i.e. three base creators that define the six methods). The case for DeMIMA is unclear since we have no access to the actual results or the tool, which perhaps explains the occasional large number of detected instances and the corresponding low precision.

We evaluate the results using the metrics of *precision* and *recall* [28], which are typically used for evaluating contemporary information retrieval systems. In this context, these two metrics are defined as follows:

$$Precision = \frac{Number\ of\ True\ Positives}{Number\ of\ detected\ instances}$$

$$Recall = \frac{Number\ of\ True\ Positives}{Number\ of\ actual\ framework\ instances}$$

where a *True Positive* is pattern instance that is correctly detected. In order for an instance to be a true positive, the instance has to be explicitly stated in the framework's documentation or there has to be strong indications found by inspecting the available code comments or the source-code itself (e.g. through naming of types and methods). Furthermore, we have consulted with other trusted resources that document design pattern instances found in popular frameworks. The most notable resource is the P-MARt [17] public repository, which contains a database of design pattern information for selected software projects. In all the tests performed, none of the inferred pattern instances violates any of the structural or behavioral constraints that we specified for the patterns. This gives us confidence that the knowledge base accurately represents the structure and behavior of the source code.

However, as noted in Table 2 and Table 3, there are a few cases where the precision suffers since the identified instances were considered by our standards as *False Positives*. In all these cases, we could not find strong evidence by inspecting available documentation and the source code that those inferred instances are indeed true positives. For example, Table 3 shows that Sempatrec returned one false positive of Template Method in JHotDraw. In particular, class `ChangeConnectionHandle` in

Table 2. Creational and structural pattern detection results and evaluation measures of Sempatrec and other comparable approaches

		Sempatrec					**Sempatrec Relaxed**					**DeMIMA [18]**					**SSA [34]**				
		T[1]	TP[2]	FN[3]	P[4]	R[5]	T	TP	FN	P	R	T	TP	FN	P	R	T	TP	FN	P	R
Singleton	JUnit	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na
	JHotDraw	2	2	0	100%	100%	3	2	0	67%	100%	2	2	0	100%	100%	2	2	0	100%	100%
	JRefactory	4	4	-	100%	-	14	8	-	57%	-	4	2	-	14%	-	12	8	-	67%	-
Factory Method	JUnit	0	0	0	na	na	0	0	0	na	na	23	0	0	0%	na	0	0	0	na	na
	JHotDraw	3	3	0	100%	100%	3	3	0	100%	100%	189	3	0	2%	100%	2	2	1	100%	67%
	JRefactory	2	1	-	50%	-	5	3	-	60%	-	71	1	-	1%	-	1	1	-	100%	-
Abstract Factory	JUnit	0	0	0	na	na	0	0	0	na	na	27	0	0	0%	na	na	na	na	na	na
	JHotDraw	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	na	na	na	na	na
	JRefactory	0	0	-	na	-	3	0	-	0%	-	167	0	-	0%	-	na	na	na	na	na
Composite	JUnit	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%
	JHotDraw	1	1	0	100%	100%	3	1	0	33%	100%	3	1	0	33%	100%	1	1	0	100%	100%
	JRefactory	0	0	-	na	-	0	0	-	na	-	0	0	-	na	-	0	0	-	na	-
Adapter	JUnit	1	1	0	100%	100%	3	1	0	33%	100%	9	0	1	0%	0%	6	1	0	17%	100%
	JHotDraw	18	8	-	45%	-	25	10	-	40%	-	28	1	-	4%	-	23	10	-	44%	-
	JRefactory	22	13	-	59%	-	36	15	-	42%	-	47	17	-	36%	-	26	17	-	65%	-
Decorator	JUnit	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%
	JHotDraw	2	1	2	50%	33%	7	3	0	43%	100%	13	1	2	8%	33%	3	1	2	33%	33%
	JRefactory	0	0	-	na	-	1	0	-	0%	-	1	0	-	0%	-	0	0	-	na	-
Average Precision					82.18%					51.92%					26.54%					75.10%	
Average Recall						90.43%					100%					88.84%					85.71%

[1] Total number of detected instances by Sempatrec and those reported by other approaches
[2] Number of detected True Positives
[3] Number of False Negatives (Actual number of framework instances - Number of True Positives)
[4] Precision value
[5] Recall value

package `CH.ifa.draw.standard` was returned as a template class. However, it is not counted as true positive since we could not find any evidence to support Sempatrec's claim. The only template class that was confirmed by Sempatrec, SSA, DPRE, PMARt, and the available documentation is the `AbstractFigure` class in package `CH.ifa.draw.standard`.

Recall statistics, however, are much harder to obtain since it requires the identification of all *False Negatives* – actual pattern instances that were not detected. Most framework documentation does not explicitly report all pattern instances. Thus manual identification through source-code inspection is subject to one's interpretation of what constitutes a pattern instance. For large frameworks such as JRefactory, the task of identifying all pattern instances is cost prohibitive. Thus, the recall information for JRefactory is absent from the tables. This is due to JRefactory's large size and complexity and that JRefactory does not have enough internal and external documentation of its design patterns. Recall information is also missing for Adapter and State/Strategy patterns for JHotDraw since we were unable to confirm the exact number of actual instances based on code inspection, public datasets, or other JHotDraw documents.

Despite the lack of complete information, we have found some evidence of false negatives in JHotDraw for observer and decorator patterns. Sempatrec missed two decorator instances and three observer instances. After examining JHotDraw documentation and code inspection, we confirmed that the five observer instances detected by DPRE were indeed true positives. To our best knowledge, these are the only known true positives, two of which were detected by Sempatrec. The only observer instance that was detected by Sempatrec, SSA, and DPRE is the case where a `DrawingChangeListener` object acts as an observer to a `StandardDrawing` object. The other observer instance inferred by Sempatrec and confirmed by P-MARt, DeMIMA, and DPRE is that the interface `FigureChangeListener` is observing a `Figure` object. For decorator, the only instance that was both

Table 3. Behavioral pattern detection results and evaluation measures of Sempatrec and other comparable approaches

		Sempatrec					**DeMIMA [18]**					**SSA [34]**					**DPRE [9]**				
		T	TP	FN	P	R	T	TP	FN	P	R	T	TP	FN	P	R	T	TP	FN	P	R
Template Method	JUnit	1	1	0	100%	100%	11	0	1	0%	100%	1	1	0	100%	100%	na	na	na	na	na
	JHotDraw	2	1	0	50%	100%	31	1	0	3%	100%	5	1	0	20%	100%	1	1	0	100%	100%
	JRefactory	6	6	-	100%	-	49	0	-	0%	-	17	17	-	100%	-	0	0	-	na	-
Observer	JUnit	1	1	0	100%	100%	4	1	0	25%	100%	1	1	0	100%	100%	na	na	na	na	na
	JHotDraw	4	2	3	50%	40%	7	2	3	29%	40%	3	1	4	33%	20%	9	5	0	56%	100%
	JRefactory	0	0	-	na	-	1	0	-	0%	-	0	0	-	na	-	0	0	-	na	-
Visitor	JUnit	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	na	na	na	na	na
	JHotDraw	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na
	JRefactory	2	1	-	50%	-	4	2	-	50%	-	2	1	-	50%	-	2	1	-	50%	-
State - Strategy	JUnit	4	3	0	75%	100%	8	0	3	0%	0%	3	2	1	67%	67%	na	na	na	na	na
	JHotDraw	39	8	-	21%	-	21	6	-	29%	-	44	11	-	25%	-	79	22	-	29%	-
	JRefactory	7	0	-	0%	-	22	2	-	9%	-	11	0	-	0%	-	1	1	-	100%	-
Average Precision					60.67%					14.50%					55%					67%	
Average Recall						88%					67%					77.40%					100%

detected by Sempatrec and SSA is the abstract class `DecoratorFigure`, which is used to decorate figures with borders and other features.

Table 3 also shows that Sempatrec missed some template method instances when operated on JRefactory. In general Sempatrec missed some observer, decorator, and template method instances caused by overly strict constraints. For example, the current constraints for Template Method pattern requires the primitive operation that is called by the template method to be declared *protected*. However, this restriction is not a standard requirement. We found evidence of that in both JHotDraw and JUnit. It is obvious that relaxing this requirement can help detect the missed instances. This intuition is confirmed in all three cases. As shown in Table 2, when we ran Sempatrec with relaxed rules, it was able to recover the missing decorator instances. This offers some evidence to support the hypothesis H2.

For comparison purpose, we have defined relaxed constraints for both observer and the template method pattern. Using these relaxed constraints, Sempatrec inferred 21 template method instances in JRefactory, 13 of which are confirmed true positives. It also inferred 15 observer instances in JHotDraw, 5 of which are true positives. This result provides further support for hypothesis H2.

DeMIMA also provides facilities for relaxing strict constraints and replacing them with weaker ones through the use of explanation-based constraint programming in order to find similar micro-architectures of design motifs. The flexibility of Sempatrec and DeMIMA in giving the user full control over specifying patterns in terms of rules or constraints is indeed an added benefit for pattern detection. In particular, this facility leads to more practical ways of dealing with issues related to the implementation variation problems that are discussed in Sect. 3.2. For example, for Singleton pattern, it is possible to account for the variations of lazy instantiation or the way that the static reference is defined by slightly modifying the corresponding constraints. Note that while relaxed constraints can reduce false negatives, they may introduce more false positives and result in lower precision (see Table 2).

In terms of pattern support, there are some differences between the four approaches. SSA does not support the Factory Method pattern. SSA and DeMIMA support the Prototype pattern, which is not supported by Sempatrec. DPRE supports all behavioral patterns and nothing else. Moreover, it is the only approach that differentiates between State and Strategy, all other approaches treat the two patterns the same way due to their structural similarity. Therefore, the results shown in Table 3 for DPRE represent the combined State and Strategy instances. (e.g. in JHotDraw, DPRE detected 36 State instances and 43 Strategy instances).

Table 2 shows that Sempatrec achieved higher average precision than other tools for the frameworks in question. Therefore, showing strong support for hypothesis H1. However, Table 3 clearly shows that DPRE achieved highest precision and recall when detecting behavioral design patterns in JHotDraw and JRefactory (DPRE did not report results on JUnit). DPRE is specialized in detecting behavioral patterns by employing dynamic analysis techniques, which are helpful in identifying the order in which messages are communicated among pat-

tern participants at runtime. The lack of dynamic information also explains the existence of some false positives in the case of Sempatrec, DeMIMA, and SSA. In general, dynamic information can be effective in detecting behavioral patterns if there exist representative sets of test cases. However, these tests are not always available and there is no way to determine whether behavioral information collected from running the selected test cases is the typical behavior of the tested code. This may explain the relatively lower precision of DPRE for detecting observer, state, and strategy patterns in JHotDraw.

As an automatic pattern retrieval tool, Sempatrec performed quite well for detecting behavioral patterns. This is in part due to the fact that not all behavioral patterns have intricate behavioral aspects that require dynamic analysis. Therefore, we can rely on their strong structural descriptions for recovery purposes, which can lead to good results when static information in method bodies is effectively captured and semantically represented.

The overall results in Table 2 and Table 3 indicate that Sempatrec performed well in most cases in terms of precision and recall, which shows support for hypotheses H1 and H2. The results also show that there should be some tradeoff between precision and recall since design patterns are abstract in nature and their interpretation is subjective.

We also collected the runtime statistics, which show that the performance of Sempatrec is within acceptable range. On an Intel 32-bit processor machine with 512MB of memory allocated to the Java Virtual Machine, it took 3 seconds for JUnit, 4 seconds for JHotDraw, and 13.5 seconds for JRefactory to parse the framework, process the ontologies, and prepare the knowledge base for querying. Once this process is done, SPARQL query execution completes almost instantaneously, with little runtime overhead. However, the reasoner takes more time when operating on an ontology with SWRL rules. On average, the runtime for the reasoner to execute SWRL rules for all patterns is about 28 seconds on JUnit, 3.6 minutes on JHotDraw, and 11.3 minutes on JRefactory, which is comparable to other approaches. Note that the parsing of framework library and generation of the knowledge base is performed only once. Once the reasoner executes the rules and generates the classification hierarchy, any further interrogation of the knowledge base requires little or no runtime overhead.

The runtime statistics indicate that the runtime for library parsing, ontology loading, and query execution is minimal when compared to the runtime for the reasoner to classify the ontologies and execute SWRL rules. *Pellet*, the ontology reasoner that we used, does not claim optimal performance. However, it does have an integrated rule engine that provides a complete and sound support for SWRL, which adds to runtime overhead. In fact, Pellet is continuously improving its support for SWRL. The current experiments were run using Pellet version 2.3, which have better runtime than our previous experiments using older versions of Pellet.

Lastly, 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, the way the rules are written affects the reasoning time. For example, reasoning with unary atoms is usually less efficient than reasoning with binary atoms especially in large knowledge bases. SCROs structure allows users to avoid using unary atoms in rules. Moreover, when authoring SWRL rules, one should pay close attention to the Open World Reasoning (OWR) that SWRL and OWL supports and most importantly, DL-safety should be ensured to achieve decidability. DL-Safe rules [24] are a subset of SWRL rules such that each variable is bound only to known individuals explicitly stated in the ABox. Over all, we believe that our tool performs well and will have better performance with more efficient ontology reasoners.

4.2 Discussion

Our case studies show support for our hypotheses in terms of achieving better precision and recall for detecting pattern instances. In this subsection, we briefly discuss a few key points that helped Sempatrec achieve this outcome.

Firstly, Sempatrec uses a pure ontology-based knowledge representation mechanism which ensures consistent and formal functional representation of design patterns, their participants, and the system under study. This representation is flexible since it can be used to detect patterns in libraries written in any programming language. In fact, the ontologies themselves can be extended to support any applications that reason about rich descriptions of software knowledge. Another key advantage of the representation is attributed to the fact that descriptions contained in a given ontology can always be extended with new facts and assertions, which can be from another ontology. The reasoner can combine the new knowledge with the existing knowledge base and compute entailments of additional information that was not explicitly stated.

Secondly, pattern recovery is often complicated by the implementation variation problem (cf. Sect. 3.2). Using ontology formalism, this problem can be considerably mitigated since different variations can be formally specified as ontology-based metadata and the detection tool can process the descriptions independently.

On the technical side, Sempatrec is usable and practical. Pattern descriptions including the roles and interactions of participants are not hard-coded within the tool, which makes it easy for users to change the role constraints without rebuilding the entire tool. Furthermore, our parser is very effective in capturing all structural and

behavioral aspects of program code that can be detected statically.

In all, the approaches we examined in these experiments provided invaluable insights that helped us improve our tool. In fact, Sempatrec is considered in some ways an extension to some of these works.

4.3 Threats to Validity

There exists several external threats to the validity of our exploratory experiments. In particular, the validity of the experiments is limited by the choice and number of software frameworks used. Firstly, we used frameworks that we previously knew that they contain considerable number of various design patterns. Secondly, we experimented with only three frameworks. Although this is more than the average number for other approaches, positive outcomes from additional experiments may further strengthen our conclusions. Thirdly, although we have used frameworks of varying sizes, we are unable to establish the scalability of the reasoner's performance without testing it on more large-scale framework libraries. In addition, we have not investigated how the variations of the SWRL rule encoding of the pattern-detecting constraints may affect the precision and recall as well as the runtime performance of our tool.

Finally, we experimented with only 12 GoF patterns [9]. We followed the lead of other approaches for selecting these commonly-used and representative patterns that span the three GoF pattern groups. Although we expect similar outcomes for the rest of GoF patterns, a more comprehensive study may yield additional discoveries.

We expect that additional experiments may reduce some of these threats. In particular, a tool usability and user experience study may give us insight on how potential users are interacting with Sempatrec to determine potential problems with defining customized SWRL rules for pattern detection. Furthermore, more case studies with other framework libraries and with alternative reasoners would produce more accurate statistics on precision and recall as well as runtime performance. For example, our study on JRefactory did not produce any recall metrics since we are unable to identify all of its pattern instances due to the lack of design documentation. Additional experiments when large-scale and well-documented benchmark libraries become available would mitigate these threats.

5 Related Work

In this section, we present an overview of the current state-of-the-art in design pattern recovery. For clarity, we broadly categorize the approaches based on their detection strategy for capturing the behavioral aspects of program code – static analysis or dynamic analysis. For each approach within a category, we briefly discuss its methodology and knowledge representation mechanism.

5.1 Static Analysis

Static analysis approaches attempt to build models that capture the structural aspects of source code such as class relationships and their dependencies. For example, Tsantalis et al.[34] proposed a Similarity Scoring Approach (SSA) for detecting pattern and modified pattern instances. SSA 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 to populate the matrices for a particular system. This representation is matched using a similarity scoring algorithm with pattern descriptions that are hard-coded within the tool.

Other approaches utilize informal annotations and XML-like mark-up language tags to represent software knowledge. Rasool and Mäder[30] proposed an approach based on creating a semi-formal XML-based definitions of the patterns' structural features such as classes, their relationships, and method return types. These feature definitions are then cataloged and searched using feature-specific search techniques in order to recover pattern instances. Balanyi and Ferenc [3] proposed an approach that describes design patterns externally using their own DPML language – an XML-based pattern description language. Similar to our approach, users of the system are granted full control over these descriptions. This approach relies on analyzing the source code to obtain its Abstract Semantic Graph (ASG) representation. This representation is then matched with the DPML descriptions to detect pattern instances. This work was later extended by applying conventional machine-learning techniques to the results of the previous work in order to enhance performance and reduce false positives [14].

Our approach to encoding knowledge is primarily different than XML-based approaches due to the fact that we define formal ontologies to explicitly describe and encode software knowledge. In particular, OWL-DL ontologies have formal foundations backed by Description Logic that enables computing entailments. Consequently, the expressive power and reasoning support provided by these descriptions enabled us to define reusable and extensible inference rules and evaluate these rules using well-defined and established tools.

Closely related to our approach is the Pat system [22], which uses Prolog rules to recover structural design patterns by utilizing a CASE tool that extracts design information in C++ code. In particular, design patterns are described using a variation of Prolog predicates and information about program elements and their roles are

[9] The Mediator pattern is also supported but no results included since Semaptrec did not recover any pattern instances in these frameworks and other tools do not support this pattern

represented as Prolog facts. Recently, the Web of Patterns (WOP) proposal [11] and the work presented in [21] explored the notion of utilizing ontologies for structuring the source-code knowledge. Although WOP is backed by a tool implementation for Java code, specific Java features that are usually used to detect design patterns (e.g. inner classes and interfaces) are not supported. However, the issues with WOP are justified since the primary focus of the project was to facilitate knowledge sharing about patterns, anti-patterns, and refactoring. Our aim, however, is to provide a semantic-based approach for design pattern detection and to provide an aid for understanding, conceptualizing, and reasoning about source-code knowledge.

Although these approaches provided valuable insights into utilizing semantic technologies for pattern detection, there were no formal ontologies defined or used by these approaches except the WOP project, which provided basic ontology templates that must be defined and extended by users [10]. On the other hand, our work contributed a thoroughly defined set of OWL-DL ontologies that capture design patterns and major features of object-oriented source code knowledge. We further provided a mechanism for modularizing and instantiating these ontologies in order to obtain inference-ready knowledge bases that are used as basis for improving the precision of detecting design patterns.

Static analysis approaches can be effective in detecting structural patterns but may fall short when detecting behavioral and some creational patterns. Therefore, many approaches, including Sempatrec, try to overcome this limitation by capturing more details about the behavioral aspects of source-code such as analyzing method bodies and method invocations. Examples of these approaches include DeMIMA [18], DPJF [5], and PINOT [32]. DeMIMA is a semi-automatic approach for detecting micro-architectures that are similar to design motifs found in models obtained from source-code. This approach uses explanation-based constraint programming and constraint relaxation to identify these micro-architectures in a multi-layered fashion. The first two layers are devoted for obtaining an abstract model of source code including specifics about classes and various class relationships such as aggregation, association, and use relationships. The third and final layer identifies design patterns in the obtained abstract model.

DPJF is based on combining some variations of the output of other existing detection tools. However, DPFJ relaxes some of the restrictive conditions used in other tools and combines structural and behavioral constraints in order to improve precision and recall. Similar to our approach, DPFJ utilizes structural and behavioral analysis on program elements. However, behavioral analysis in DPFJ is performed using Control Flow Graphs (CFG). This kind of graphs is known to have scalability issue. Similarly, PINOT represents design patterns using CFGs. PINOT processes the Abstract Syntax Tree (AST) of method bodies to build CFG representation for program elements. The CFG is then examined to verify the existence of restrictions related to a particular design pattern.

5.2 Dynamic Analysis

Dynamic analysis approaches [8,9,31,36] utilize information obtained through executing and monitoring the running code in order to get a more accurate realization of patterns' behavior at run-time. The goal is to capture the behavioral relationships that can distinguish different patterns with similar structures from each other.

For example, De Lucia et al. [8] proposed an approach for detecting behavioral design patterns. In this approach, pattern candidates are initially identified in a static analysis phase, which involves performing analysis on class diagrams in order to capture structural information about patterns' participants. The dynamic analysis phase involves code instrumentation of the candidates in order to trace their method invocations at run-time. Pattern behaviors are then verified by monitoring the execution of the instrumented program on a representative test suite to obtain the sequence in which relevant methods are called and the order in which they run. This approach was further extended [9] by introducing a model checker that is able to improve the initial set of identified pattern candidates in the static analysis phase. This approach was backed by a prototype named DPRE and validated by applying this tool on a set of behavioral design patterns.

Wang and Tzerpos [36] proposed a pattern recovery tool for Eiffel source-code. This approaches uses REQL scripts to define the static structure and RSF format to define the dynamic behavior for each design pattern. Sartipi and Hu [31] utilize dynamic analysis to set the stage for detecting design patterns specified using a specialized Pattern Description Language (PDL). Both structural and approximate matching algorithms are used to identify pattern instances.

In general, dynamic analysis can be very effective in detecting behavioral patterns, especially when the detection tool is backed by a representative and complete set of test cases for the analyzed frameworks. However, the issue of selecting representative test cases can be a viable threat to the reliability of dynamic analysis [8]. We believe that some behavioral design patterns that have strong structural descriptions such as the visitor pattern can still be detected with the absence of dynamic data, especially when proper semantic descriptions of pattern participants are utilized and when static behavioral aspects are effectively captured. Furthermore, due to the flexibility and usability of semantic-based detection mechanisms, they can be combined with other approaches that are based on dynamic analysis in order to improve the precision of detecting behavioral patterns.

6 Conclusions and Future Work

We have presented a reverse-engineering approach that enhances program understanding through the automatic recovery of design patterns from source code. In this approach, we relied solely on the semantic representations of design patterns encoded with ontology constructs and SWRL rules. We have showed that our approach can be used for basic ontology-based search over a populated knowledge base. We demonstrated the utility of the approach by evaluating our implementation with three framework libraries. The detection process is based on logical inference, which only requires a rule-based reasoner that is capable of processing SWRL rules. Once the rules are processed, the reasoner would infer pattern instances based on a matching between the semantic constraints specified in these rules with source-code descriptions found in a knowledge base representing the framework at hand. This approach distinguishes itself as being precise, extensible, and practical.

As a future work, we will investigate the effectiveness of applying inter-procedural point-to analysis [12] to detecting behavioral patterns. Point-to analysis can be used to obtain more precise type information to capture some aspects of the dynamic behavior of software libraries. For behavioral patterns, precision is usually enhanced by obtaining dynamic information from monitoring activity of the executed program. This enables a detection mechanism to discriminate, for example, the Observer design pattern from the Bridge pattern. Another future work is to investigate the possible tool extensions to support the detection of anti-patterns (undesired or counterproductive design practices) and to identify possible refactoring opportunities.

An overreaching goal of this work is to develop a comprehensive semantics-enabled environment that addresses major software understanding issues. In this environment, not only ontologies play an effective role in design recovery, ontologies will also provide effective support for code refactoring, software component retrieval, and automatic code recommendation.

References

1. Alnusair, A., Zhao, T.: Towards a model-driven approach for reverse engineering design patterns. In: 2nd International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE'09) at ACM/IEEE MoDELS'09. Denver, CO, USA (2009)
2. Alnusair, A., Zhao, T.: Source Code Representation Ontology (SCRO), design pattern ontologies, and framework ontologies (2012). Available online at: http://www.indiana.edu/~awny/index.php/research/ontologies
3. Balanyi, Z., Ferenc, R.: Mining design patterns from C++ source code. In: Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM'03), pp. 305–314. Amsterdam, The Netherlands (2003)
4. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American **284**(5), 34–43 (2001)
5. Binun, A., Kniesel, G.: DPJF - design pattern detection with high accuracy. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12), pp. 245–254. Szeged, Hungary (2012)
6. Chen, L., Shadbolt, N.R., Globe, C.A.: A semantic web-based approach to knowledge management for grid applications. IEEE Transactions on Knowledge and Data Engineering **19**(2), 283–296 (2001)
7. Cinneide, M.O., Nixon, P.: A methodology for the automated introduction of design patterns. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'99), pp. 463–472. Oxford, UK (1999)
8. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Behavioral pattern identification through visual language parsing and code instrumentation. In: Proceedings of 13th European Conference on Software Maintenance and Reengineerig (CSMR'09), pp. 99–108. Kaiserslautern, Germany (2009)
9. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Improving behavioral pattern detection through model checking. In: Proceedings of 14th European Conference on Software Maintenance and Reengineerig (CSMR'10), pp. 176–185. Madrid, Spain (2010)
10. Dietrich, J., Elgar, C.: A formal description of design patterns using OWL. In: Proceedings of the Australian Software Engineering Conference (ASWEC 2005), pp. 243–250. Brisbane, Australia (2005)
11. Dietrich, J., Elgar, C.: Towards a web of patterns. In: Workshop on Semantic Web Enabled Software Engineering (SWESE), pp. 117–132. Galway, Ireland (2005)
12. Emami, M., Rakesh, G., Hendren, L.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), pp. 242–256 (1994)
13. Falbo, R., Guizzardl, G., Duarte, K.C., Natali, A.C.: Developing software for and with reuse: An ontological approach. In: ACIS International Conference on Computer Science, Software Engineering, Information Technology, e-Business and Applications (CSITeA-2002), pp. 311–316. Foz do Iguacu, Brazil (2002)
14. Ferenc, R., Beszedes, A., Fulop, L., Lele, J.: Design patterns mining enhanced by machine learning. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), pp. 295–304. Budapest, Hungary (2005)
15. Gamma, E., Helm, E., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series (1995)
16. Gruber, T.R.: A translation approach to portable ontology specification. Knowledge Acquisition **5**(2), 192–220 (1993)
17. Guéhenéuc, Y.G.: P-MARt: pattern-like micro architecture repository (2007). Retrieved July 27, 2012, from: http://ptidej.dyndns.org/downloads/pmart/
18. Guéhenéuc, Y.G., Antoniol, G.: DeMIMA: A multilayered approach for design pattern identification. IEEE

Transactions on Software Engineering **34**(5), 667–684 (2008)
19. Horrocks, I.: Ontologies and the semantic web. Communications of the ACM **51**(12), 58–67 (2008)
20. Jasper, R.J., Uschold, M.F.: A framework for understanding and classifying ontology applications. In: IJCAI99 Workshop on Ontologies and Problem-Solving Methods. Stockholm, Sweden (1999)
21. Kirasić, D., Basch, D.: Ontology-based design pattern recognition. In: Proceedings of the International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES'08), pp. 384–393 (2008)
22. Kramer, C., Prechelt, L.: Design recovery by automated search for structural design patterns in object oriented software. In: Proceedings of Working Conference on Reverse Engineering (WCRE96), pp. 208–215 (1996)
23. Manola, F., Miller, E.: RDF primer. W3C Recommendation (2004). Retrieved July 7, 2012, from: http://www.w3.org/TR/rdf-primer/
24. Motik, B., Grau, B.C., Sattler, U.: Structured objects in OWL: Representation and reasoning. In: Proceedings of the International World Wide Web Conference (WWW'08), pp. 555–564. Beijing, China (2008)
25. Motik, B., Patel-Schneider, P.F., Grau, B.C.: OWL Web Ontology Language reference. W3C Recommendation (2009). Retrieved July 4, 2012, from: http://www.w3.org/TR/owl2-direct-semantics/
26. Noy, F.N., McGuinness, D.L.: Ontology development 101: A guide to creating your first ontology. Stanford Knowledge System Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report. SMI-2001-0880 (2001)
27. O'Connor, M.J., Knublauch, H., Tu, S., Grosof, B., Dean, M., Grosso, W., Musen, M.: Supporting rule systems interoperability on the semantic web with SWRL. In: Proceedings of the 4th International Semantic Web Conference. Galway, Ireland (2005)
28. Olson, D.L., Delen, D.: Advanced Data Mining Techniques. Springer, Verlag (2008)
29. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation (2008). Retrieved July 7, 2012, from: http://www.w3.org/TR/rdf-sparql-query/
30. Rasool, G., Mader, P.: Flexible design pattern detection based on feature types. In: Proceedings of the International Conference on Automated Software Engineering (ASE'11), pp. 243–252. Lawrence, KS, USA (2011)
31. Sartipi, K., Hu, L.: Behavior-driven design pattern recovery. In: Proceedings of the Twelfth International Conference on Software Engineering and Applications(SEA'08), pp. 179–185. Orlando, Florida, USA (2008)
32. Shi, N., Olsson, R.A.: Reverse engineering of design patterns from Java source code. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp. 123–132. Tokyo, Japan (2006)
33. 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), 51–53 (2007)
34. Tsantalis, T., Stephanides, A., Halkidis, S.: Design pattern detection using similarity scoring. IEEE Transactions on Software Engineering **32**(11), 896–909 (2006)
35. Uschold, M., Healy, M., Williamson, K., Clark, P., Woods, S.: Ontology reuse and application. In: Proceedings of the 1st International Conference on Formal Ontology in Information Systems (FOIS 1998), pp. 179–192 (1998)
36. Wang, W., Tzerpos, V.: Design pattern detection in Eiffel systems. In: Proceedings of the 12th working conference on reverse engineering (WCRE'05), pp. 165–174 (2005)