Accurate Design Pattern Detection Based on Idiomatic Implementation Matching in Java Language Context

Renhao Xiong, Bixin Li*
School of Computer Science and Engineering, Southeast University
Nanjing, China
{renhao.x, bx.li}@seu.edu.cn

***Abstract*—Design patterns (DPs) are widely accepted as solutions to recurring problems in software design. While numerous approaches and tools have been proposed for DP detection over the years, the neglect of language-specific mechanism that underlies the implementation idioms of DPs leads to false or missing DP instances since language-specific features are not captured and similar characteristics are not distinguished. However, there is still a lack of research that emphasizes the idiomatic implementation in the context of a specific language. A vital challenge is the representation of software systems and language mechanism.**

In this work, we propose a practical approach for DP detection from source code, which exploits idiomatic implementation in the context of Java language. DPs are formally defined under the blueprint of the layered knowledge graph (LKG) that models both language-independent concepts of DPs and Java language mechanism. Based on static analysis and inference techniques, the approach enables flexible search strategies integrating structural, behavioral and semantic aspects of DPs for the detection. Concerning emerging patterns and pattern variants, the core methodology supports pluggable pattern templates. A prototype implementation has been evaluated on five open source software systems and compared with three other approaches. The evaluation results show that the proposed approach improves the accuracy with higher precision (85.7%) and recall (93.8%). The runtime performance also supports its practical applicability.

***Index Terms*—Design Pattern Detection, Reverse Engineering, Software Comprehension, Knowledge Representation**

I. Introduction

In recent decades, the research of design patterns (DPs) is an active field in software engineering [1]. DPs are solutions to recurring problems in object-oriented (OO) software systems [2]. They encapsulate valuable design knowledge and satisfy a system's critical requirements with proper implementation [3]. The employment of DPs has the advantages to maintain the consistency between design and implementation and increase the software reusability and modularization [4]. If chosen appropriately, DPs typically improve certain aspects of software quality [5] [6]. In the maintenance phase of the software life cycle, DPs provide maintainers with considerable insight into the software structure and its internal characteristics [7]. They help to understand the original design of large and complex software systems and to facilitate the upgrade [8]. The search on DPs also spans the reengineering areas such as refactoring and re-documentation [9]. From the reverse engineering point of view, detecting pattern instances in software artifacts may help in design-to-code traceability and quality assessment [10].

Motivated by the comprehension, modification, and maintenance of systems that are lack of adequate documentation, numerous approaches and tools have been proposed for DP detection. Various techniques are applied including graph matching [1] [11], software metrics [10] [12], model checking [7] [13], and machine learning [14] [15].

While the concept of a DP is language-independent and can be applied to different OO programming languages, the implementation of a DP resides in the context of a specific language. To implement some structural aspects of a DP, the relationships are mapped to the mechanism of the target language [16]. For instance, the aggregation is usually implemented as reference attributes in Java or pointers in C++. The use of a standard container, e.g. Vector provided in JDK (Java Development Kit) [17], is chosen by experience or according to specific needs. The implementation of some DPs is very tricky and directly relevant to special mechanisms of a programming language [18]. Some programming languages even provide library classes that facilitate the implementation of a DP [19].

As the implementation captures a non-trivial idiom of the programming language to serve a concrete purpose, a DP is traceable because it is tied to the programming language and imposes a condition on a single software module [20]. In consequence, the neglect of language mechanism that underlies such implementation idioms leads to false or missing detection results since language-specific features of DPs are not captured and similar characteristics are not distinguished. However, there is still a lack of research that emphasizes the language mechanism and implementation idiom for DP detection.

In the context of a specific language, e.g. Java, an unavoidable challenge is how to represent the language constructs (e.g. class, field and method call), exact concepts (e.g. implementation, overloading and overriding) and fuzzy relationships in

* Corresponding author

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

which the participants and associations are not straightforward (e.g. composition, up-casting and unlimited multilevel inheritance) [21]. The following challenge is how they can be utilized for the detection. While the search space can be greatly reduced taking the advantage of precisely matching the language constructs and exact concepts, the representation also needs to adapt the fuzzy relationships which are one of the main sources of implementation variants. For example, a variant may reside in a multilevel inheritance hierarchy where an additional inheritance level, usually an abstract class, provides a default implementation for descendant classes [11]. Based upon these language constructs, concepts and relationships, the more abstract feature can be represented, e.g. the late-binding that relies on inheritance, overriding and up-casting.

Abstract features, fuzzy relationships, and their combination are essential for the detection of DPs and their variants, but are difficult to extract straightforwardly from the source code. Besides, the extensibility of a DP detection approach is highlighted with the steady increase in the number of design patterns in the literature and online repositories [14]. In the application scenario to support the detection of emerging DPs, approaches based on hard-coded algorithms are insufficient.

Focusing on the problem of knowledge representation in the context of Java, we propose a practical approach that leverages semantic web techniques for automatic DP detection. To model the concepts of DPs and the constructs of Java, a three-layer knowledge graph is constructed. After transforming the source code into ontologies, inference rules are applied to capture complex relationships and abstract features. To be continuously extensible, the methodology of our approach supports pluggable pattern templates, each of which describes a DP.

A prototype tool based on the proposed approach has been evaluated on five benchmark systems and compared with three approaches. The evaluation results support the accuracy and runtime efficiency of the proposed approach. All relevant resources of this study are available online [22].

The **subsequent sections** are structured as follows. Section II first presents the modeling of DPs and Java, and then describes the processes to detect DPs. We also demonstrate the detection of the canonical Observer pattern with the pattern template that characterizes both the structural and behavioral aspects. In Section III, an experimental evaluation is presented, after which we discuss the naming conventions used in the detection rules and potential extensions in Section IV, and debate the threats to validity in Section V. Following the discussion of related techniques and tools in Section VI, conclusions and future work are finally presented in Section VII.

II. Approach

As shown in Fig. 1, the pattern detection process of the proposed approach is composed of three sub-processes: (i) Ontology Generation, (ii) Knowledge Inference, and (iii) Pattern Template Matching. In this section, we first formally define DPs in description logic (DL) [23], which is important to solve the problems of missing roles and pattern instance identification in detection results. Then we describe the Layered Knowledge Graph and finally present the three sub-processes.

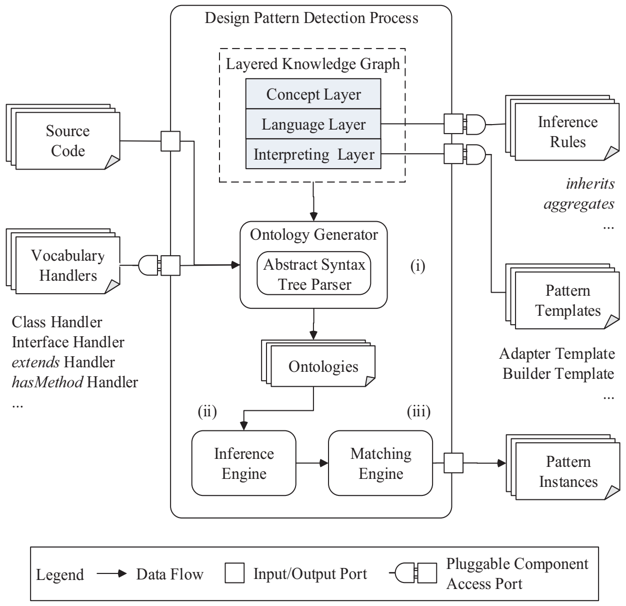


Fig. 1. Design pattern detection process

A. Definition of Design Patterns

In the GoF (Gang of Four) textbook [2], a DP is described by its name, intent, structure, and participants, etc. In the proposed approach, a DP is formally defined to capture the core structure and participants. In consequence, the detection of a DP is to satisfy the formal definition by matching the traceable clues in the source code. A DP is defined in **Def. 1** in DL.

Def. 1 Design Pattern

$$\text{DesignPattern} \equiv \exists \mathit{containsRole}.\text{Role} \sqcap \exists \mathit{containsOper}.\text{Operation}$$

Def. 2 Observer Pattern

$$\begin{aligned}\text{Observer} \equiv\ & \text{DesignPattern} \sqcap \\ & \exists \mathit{containsRole}.\text{SubjectRole} \sqcap \\ & \exists \mathit{containsRole}.\text{ObserverRole} \sqcap \\ & \exists \mathit{containsOper}.\text{AttachOper} \sqcap \\ & \exists \mathit{containsOper}.\text{NotifyOper} \sqcap \\ & \exists \mathit{containsOper}.\text{UpdateOper}\end{aligned}$$

In the definition, a pattern is composed of relevant roles and operations. A role is a class involved in the structure of a DP, and an operation is a method declared in a participant class, which interacts with other participants. A role also can be an interface in Java as Java distinguishes an object's type and class [2]. We use **key roles** to uniquely identify a pattern instance. A pattern instance may contain only one key role, e.g. the Singleton role of a Singleton pattern, or contain multiple key roles, each combination of which identifies a different instance. Thus, one combination of key roles may correspond to multiple other roles. Similarly, **required operations** are key interactions that characterize relevant roles.

A concrete DP is further defined by constraining the DesignPattern in **Def. 1**. For example, the definition of the GoF pattern Observer is shown in **Def. 2** which contains two key roles (Subject, Observer) and three required operations (Attach, Notify and Update).

The problem of missing roles in detection results is pointed out in several works [6] [24] [25]. The main reasons for missing roles in practical application of DPs are: (i) the role, e.g. the Client role described in the textbook [2] [26], does not exist in some systems such as toolkits and APIs, (ii) some DPs, for instance, the Adapter DP, intend to work with unforeseen classes [2], and (iii) the role is eliminated in the implementation variant, e.g. the Bridge DP that does not have an abstract Implementor role [2]. Thus, in our approach, such roles are optional in contrast to non-optional key roles. A pattern instance without a ConcreteSubjectRole still satisfies **Def. 2** since the role is optional and not constrained. Nevertheless, their existence is necessary in some cases and can be used for the detection. The existence of multiple optional roles does not generate new pattern instances since an instance of a DP is identified by key roles. It makes sense when a DP contains a series of roles (e.g. the Product role of an Abstract Factory DP) that are not core participants. A variant of a given DP, which may have different key roles, can be generated by modifying the definition of the canonical DP.

The definition of a DP is language-independent. In the context of Java, the definition is further interpreted for the detection. Concerning the extensibility requirement, we organize the ontology models into a layered structure that we call Layered Knowledge Graph (LKG). The vocabularies of the definition reside in the language-independent layer of LKG.

In the DP detection process of the proposed approach, the definition of a DP is interpreted later to be satisfied as shown in Fig. 1. First, the ontology generator transforms the information extracted from the source code into ontologies through static analysis techniques. Then indirect relationships between compilation units are inferred. Finally, the inferred ontologies are matched with pattern templates for the detection of pattern instances. The core methodology of the proposed approach is pluggable, which integrates the vocabulary handler, inference rules and pattern templates as its extension.

B. Layered Knowledge Graph

In the proposed approach, we construct the LKG as the blueprint of all the ontology models. An ontology model, e.g. "Java" in Fig. 2, is an OWL (Web Ontology Language) model, to which several rule sets are attached. The prefixes in brackets (e.g. "java:") are used as short names referring to the models. The highest layer, concept layer, contains ontology models for the language-independent definition of DPs. The vocabularies of **Def. 1** and **Def. 2** reside in this layer. The model "StaticBehavior" is an abstraction of behaviors that are independent of specific OO languages. We use an event-based modeling mechanism in which an event represents a behavior and associates relevant facts. For example, a method call event named "mc" is created and linked to the caller method, the method called and the invoking expression, etc.

In the language layer, language-specific events are modeled based on both the "Java" model and the abstract "StaticBehavior" model. The "Java" model describes the concepts and constructs of Java, in which the language constructs are modeled as ontology classes and the relationships between the constructs are modeled as ontology properties. To give an example, the following triples, in form of (subject *predicate* object), describe a Java class StandardDrawing that extends CompositeFigure and has a field fListeners whose type is Vector.

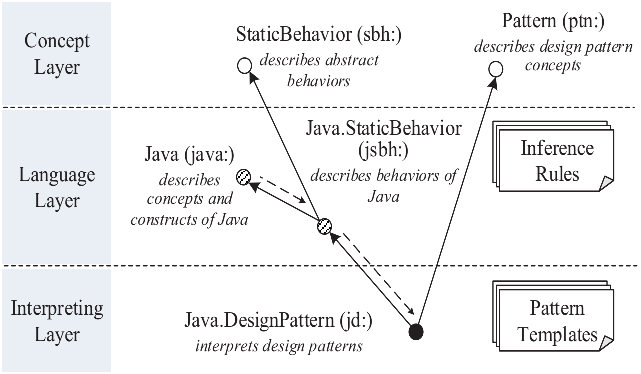

Concept Layer
StaticBehavior (sbh:)
describes abstract behaviors
Pattern (ptn:)
describes design pattern concepts
Language Layer
Java (java:)
describes concepts and constructs of Java
Java.StaticBehavior (jsbh:)
describes behaviors of Java
Inference Rules
Interpreting Layer
Java.DesignPattern (jd:)
interprets design patterns
Pattern Templates

Fig. 2. Layered knowledge graph (LKG)

(StandardDrawing *rdf:type* Class)
(StandardDrawing *java:extends* CompositeFigure)
(StandardDrawing *java:hasField* fListeners)
(fListeners *java:fieldTypeIs* Vector)

The Class mentioned in the first triple is an ontology class, which denotes the concept of a Java class. StandardDrawing's type (*rdf:type*) is Class; namely, it is an individual of Class. The individuals are linked by ontology properties (e.g. *java:extends*) to represent their relationships. Similarly, another class CompositeFigure in the second triple can be linked to its type AbstractClass by *rdf:type* and linked to the modifier "public" in its declaration by *hasModifier*. They are omitted in this example for brevity.

Based on straightforward relationships, e.g. "extends" and "implements", more complex language concepts and fuzzy relationships can be expressed. In Fig. 3, the two properties, *realizes* and *inherits*, are generalized from different combinations of classes and interfaces. They finally help to the generalization of the *isA* (up-casting) relationship. Strictly, the domain and range of these properties have a sharp boundary. The domain and range of the ontology property *isA* are both the union of Class and Interface. In a more general sense, the up-casting relationship is fuzzy because (i) the type (class or interface) of two participants is undecided, and (ii) the two participants reside in an undecided number of layers in the type hierarchy. To illustrate this, more than one *extends* may exist as shown in Fig. 3.a. Another example of composite relationships is the overriding of two methods, which can be inferred from method names, signatures, modifiers, and the inheritance hierarchy that they reside in. In the proposed approach, similar generalization techniques are applied for "fuzzy" matching.

Another model in the language layer, "Java.StaticBehavior", describes behaviors of Java, including method invocation, class instantiation, variable assignment, and field initialization, which can be extracted from the source code through static analysis. They represent the behavioral aspects of a DP.

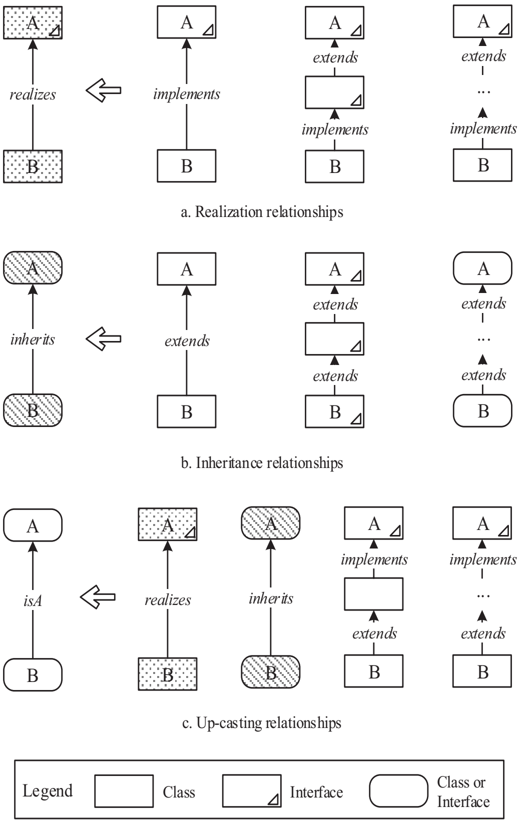


Fig. 3. Generalization of the relationships between classes and interfaces for "fuzzy" matching

Finally, down to the interpreting layer, the pattern templates are created for the matching. Based on all available vocabularies in higher layers, a pattern template is able to utilize multiple aspects: (i) the structure of a DP, (ii) behavioral interactions between DP roles, and (iii) naming conventions such as class names, method names and field names. Compared to hard-coded algorithms, the declarative way of the pattern templates supports flexible search strategies based on both exact and generalized relationships.

In the DP detection process as shown in Fig. 1, the top two layers contain all the vocabularies as the input of the Ontology Generator. The inference rules will be executed by the Inference Engine for knowledge inference in the dependency order of the ontology models that are attached the rule sets. A pattern template in the interpreting layer is a special inference rule set to satisfy the definition of a concrete DP based on the facts in the inferred ontologies. From the viewpoint of pattern instance matching, a pattern template is able to employ flexible search strategies based on the knowledge represented in higher layers. Each layer of the LKG is able to extend independently. For instance, in the language layer, existing rules can be modified, and new rules can be added. The layer is also able to support new programming languages since the concept layer is language-independent. In the interpreting layer, new pattern templates can be added based on available vocabularies.

C. Ontology Generation

The first step aims to represent the system as ontologies that can be accessed, inferred and searched. As shown in Fig. 1, the Ontology Generator, which integrates an Abstract Syntax Tree (AST) parser, transforms the source code to ontologies. As the structures and behaviors are modeled in the LKG, this process extracts the required information from the source code to generate the ontologies. Hard-coded algorithms are not appropriate mainly because (i) the information needed for DP detection is not explicit at the beginning and completely recording all details in the AST is difficult and inefficient, (ii) the language feature evolves, and (iii) the LKG intends to be extensible, thus new vocabularies can be added.

In the proposed approach, we use a vocabulary handler to deal with the extraction of one or more ontology classes or properties. A vocabulary handler is responsible for the generation of given vocabularies. Both the structural and behavioral vocabularies are filled by handlers. For instance, given the ontology class Interface, its handler enumerates all the interfaces and generates corresponding individuals of Interface with qualified names. Likewise, a handler of the *hasMethod* property enumerates all the methods of each class and interface, and associates each method individual with its declaring class. As for behavioral vocabularies, e.g. MethodInvocation and Assignment, it usually needs to extract statements and expressions.

To meet the needs of extensibility, it should be easy to add and update handlers, and they should not depend on each other. In our approach, different handlers are not coupled. They cooperate indirectly through the naming mechanism of ontology individuals. An individual is uniquely identified by a prefix and a local name. For example, the prefix of the individual "java:CH.ifa.draw.standard.StandardDrawing" is "java:". As we use the fully qualified name of a compilation unit as its local name, creating an individual with the same prefix and local name will reuse the existing one. The handlers follow the same naming mechanism, thus, for example, the *hasMethod* handler does not depend on the Class/Interface handler and the Method handler.

As shown in Algorithm 1, the Ontology Generator is based on the Visitor DP, it loads the LKG to initialize the ontology store (line 02, 03), after which it enumerates the vocabulary handlers (line 04). For each handler, its instance is dynamically loaded passing the ontology store (line 07). As the handler is a visitor of interested AST nodes, each handler instance is accepted by the compilation units to deal with concerned vocabularies (line 08-10). To illustrate, the handler of the *hasMethod* property is also shown (line 15-22). After a class or interface compilation unit accepts the HasMethodHandler instance, its visit method will be invoked passing the method declaration AST node. Within the visit method, the node is resolved (line 16). By accessing the declaring class of the method declaration node, a new ontology individual of Class/Interface is created, or an existing one is retrieved if it already exists in the ontologies. Finally, the (declaring class, *hasMethod* property, resolved method) triple is stored into the ontologies (line 21).

It does not need to build a full and complete AST since one handler is responsible for specific vocabularies. A handler

Algorithm 1 Ontology generation algorithm

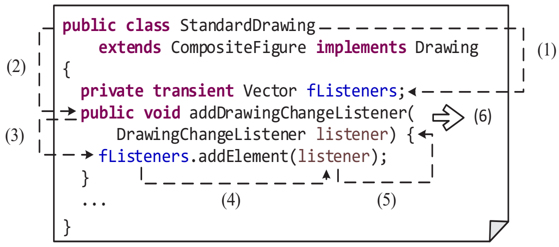
```
procedure OntologyGenerator::generate(astParser)
  lkg := loadLKG()
  ontologies := new OntologyStore(lkg)
  allHandlers := enumerateVocabularyHandlers()
  allCompilationUnits :=
    astParser.getCompilationUnits()
  for each handler in allHandlers do
    handlerInstance :=
      loadHandler(handler, ontologies)
    for each compilationUnit in
        allCompilationUnits do
      compilationUnit.accept(handlerInstance)
    end for
  end for
  return ontologies
end procedure

procedure HasMethodHandler::visit(methodNode)
  methodBinding := methodNode.resolveBinding()
  ontologies := getOntologyStore()
  subject := createOrRetrieveIndividual(
    methodBinding.getDeclaringClass())
  property := ontologies.getProperty(
    “java:hasMethod”)
  object := createOrRetrieveIndividual(
    methodBinding)
  ontologies.addTriple(subject, property, object)
end procedure
```

should only deal with straightforward extraction, e.g. Class individuals, Interface individuals and *hasField* relationships. Handlers are capable to extract information down to statement and expression level, e.g. field initialization and invoking expression of a method call. More complex concepts and fuzzy relationships are the responsibility of knowledge inference. The generator does not force the existence of any handler and the handlers do not depend on each other. Thus, the vocabulary handler repository can be easily extended.

D. Knowledge Inference

In this step, indirect knowledge is inferred upon the ontologies that contain facts extracted from the source code. An inference example of the aggregation that uses a standard container is illustrated in Fig. 4. Using JDK Vector, StandardDrawing *aggregates* DrawingChangeListener. Although the type parameter of Vector is omitted in the declaration of fListeners, the element type of Vector can be inferred since the Vector’s addElement method is called passing an argument whose type is DrawingChangeListener. Other kinds of aggregation, e.g. using arrays or other containers, also can be inferred. Without domain knowledge, such implementation idioms are difficult to capture. In the proposed approach, we use the property *aggregates* to denote the idiomatic implementation of the one-to-many aggregation [2]. The one-to-one aggregation is represented by the aforementioned property *hasField*.

Based on the generalization technique as shown in Fig. 3, complex relationships can be inferred from the facts extracted straightforwardly from the source code. The mentioned relationships, e.g. *inherits* and *isA*, are abstracted from explicit relationships, i.e. “extends” and “implements”. On the one hand, they generalize the variants that recur in the implementation, for example, multilayer “extends”. On the other hand, they represent the core concern of a concept without mentioning concrete details. To give an example, the core concern of applying a late-binding is up-casting, which forgets the type of an object [21]. As a comparison, the *isA* relationship forgets the number of layers and whether the participants are classes or interfaces, from which it is inferred. The vocabularies carrying extracted and inferred knowledge can be directly used as the elements of the “language” of pattern templates. The abstraction of core concerns also facilitates the concision of pattern templates by saving unnecessary details.



```
public class StandardDrawing
    extends CompositeFigure implements Drawing
{
  private transient Vector fListeners;
  public void addDrawingChangeListener(
      DrawingChangeListener listener) {
    fListeners.addElement(listener);
  }
  ...
}
```

(1) StandardDrawing has a field named fListeners whose type is java.util.Vector.
(2) It has a method named addDrawingChangeListener.
(3) The addElement method of Vector is called within the method.
(4) The invoking expression is fListeners and the argument is named listener.
(5) “listener” is defined in the parameter whose type is DrawingChangeListener.
(6) It is inferred that StandardDrawing *aggregates* DrawingChangeListener.

Fig. 4. Inference of the aggregation that uses JDK Vector

The inference is driven by inference rules. The facts can be expressed as triples as aforementioned. By matching the triples, the inference rule for Fig. 3.b can be expressed as:

(?b *java:inherits* ?a) ← (?b *java:extends*+ ?a)

In the rule, one or more *java:extends* properties (*java:extends*+) are matched to infer *java:inherits* using the grammar of SPARQL (Simple Protocol and RDF Query language) [27]. In this rule, the variables started with a question mark (?) denote matched subjects or objects of triples.

Inference rules are attached to ontology models in the LKG. Upon the ontologies generated in the previous process, the inference rules are executed by the Inference Engine. The pattern templates are special rules since they both are based on SPARQL update language. In the inference process of Fig. 1, the structure of LKG indicates the sequence of inference. In the LKG, lower layers depend on higher layers. As shown in Fig. 2, the interpreting model “Java.DesignPattern” depends on “Java.StaticBehavior”, and “Java.StaticBehavior” depends on “Java” and “StaticBehavior”. In the reversed direction, the execution path of the attached rules is dashed in Fig. 2. Thus, the inference order of the ontology models along the dashed path will be “Java”, “Java.StaticBehavior” and finally “Java.DesignPattern”.

The concerns of ontology models, inference rules and pattern templates are separated so that they can vary independently to support emerging patterns and pattern variants. The inference is constrained within the namespaces of the current model and dependent models. The vocabularies out of the range are not allowed to use. For example, there is only one available LKG namespace for the inference rules attached to the model

“Java”, i.e. its own namespace “java:”, since it does not depend on any other LKG model. Based on “Java” and “StaticBehavior”, all the knowledge about Java and static behaviors, including the inferred parts, is available to use in “Java.StaticBehavior” for further inference.

E. Pattern Template Matching

To detect pattern instances, the inferred ontologies are matched to pattern templates. A pattern template matches the facts about structural and behavioral aspects of a DP. As for semantic aspects, meaningful names are advised for the application of DPs [2] [26]. The naming conventions can also assist the detection by matching or filtering out related fragments that indicate specific intent. As a result of the matching, one individual of a concrete DP, e.g. the Observer defined in **Def. 2**, is created to represent each detected DP instance. It is linked to relevant roles and operations that satisfy the definition.

For demonstration purpose, we illustrate the detection of the canonical Observer pattern [2] as shown in Fig. 5. While this pattern is classified as a behavioral pattern in the catalog, both structural and behavioral aspects are utilized, which can also be applied to detect creational and structural patterns.

A challenge is to locate the aggregation relationship. An example to infer the *aggregates* property has been shown in Fig. 4. In practice, the inference of *aggregates* gathers idiomatic implementation based on frequently used standard containers. In consequence, the *aggregates* property can be directly used in the matching template in Fig. 5 (line 01).

The pattern template uses SPARQL grammar [27]. The query of SPARQL is based on triple matching. Matched individuals are bound to variables, e.g. “?subject” that starts with a question mark (?) (line 01). The letter “a” is short for the aforementioned “*rdf:type*” (line 02). While a dot (.) connects separate triples, a semicolon (;) connects two triples with the same subject (line 01). Square brackets ([]) can be used as an unnamed variable (line 06). The “union” expression provides alternatives matches, which can be roughly understood as “or” (line 02-04). Moreover, matched results can be restricted to satisfy a test (line 11, 18), or filtered by specifying the results must not match given triples (line 09, 12).

To explain concisely, Figure 5 describes the following structures and behaviors: (i) the Subject role aggregates the Observer role (line 01), (ii) the Observer is an abstract class or an interface (line 02-04), (iii) the Subject has a method “notify” (line 01); it invokes the “update” method of Observer inside a “for” or “while” loop block (line 05, 06), and (iv) the Subject has a method “attach” with a parameter whose type is Observer (line 07, 08); the “attach” method is not “private” to be visible outside the Subject, or at least reusable for its descendant (line 09). The template filters out the aggregation of the Subject role itself (line 11). It also filters out the aggregation in which relevant roles reside in an inheritance hierarchy to distinguish from a Composite pattern (line 12-20). At last, an individual that represents the detected pattern instance is created for each combination of the two key roles (line 22). It is named as “jd:dp.Observer.fbd032…”, whose suffix is generated by the SHA1 digital signature algorithm [27] taking the string concatenation of the two key roles as input.

```
Pattern Instance: Observer (?dp)
Key Roles: Subject (?subject), Observer (?observer)
Required Operations:
          Attach (?attach), Notify (?notify), Update (?update)
Matching Template:
01   ?subject java:aggregates ?observer;
         java:hasMethod ?notify.
02   { ?observer a java:Class; java:hasModifier "abstract" }
03   union
04   { ?observer a java:Interface }
05   ?observer java:hasMethod ?update.
06   [] jsbh:methodCaller ?notify;
         jsbh:methodCalled ?update; java:isInLoop true.
07   ?subject java:hasMethod ?attach.
08   ?attach java:hasParamType ?observer.
09   filter not exists { ?attach java:hasModifier "private" }
10
11   filter(?subject != ?observer)
12   filter not exists {
13     { ?subject java:isA ?observer }
14     union
15     { ?subject java:isA ?parent.
16       ?observer java:isA ?parent.
17       ?parent a java:Class.
18       filter(?parent != java:java.lang.Object)
19     }
20   }
21
22   bind(uri(concat(str(jd:), "dp.Observer.",
       sha1(concat(str(?subject), str(?observer))))) as ?dp)
```

Fig. 5. Pattern template of the canonical Observer pattern

Relying on detected key roles and required operations, optional roles and operations can be detected. Another challenge is to adapt the unlimited multilayer inheritance relationship between abstract roles and concrete roles, e.g. the Subject role and ConcreteSubject role of the Observer pattern. Based on fuzzy properties such as *inherits*, *realizes* and *isA*, the multilayer structure can be adapted and further constrained by other features. The evaluation results show that the illustrated Observer template detected additional true instances compared to rival approaches. The detection results, containing the roles, operations, and their associations, are updated to the ontologies for future access, e.g. visualization.

As we consider pattern templates as special rules, they are attached to the interpreting model “Java.DesignPattern”. As a result, the detection results are integrated into the ontologies as a whole and therefore have a natural connection with existing knowledge. While a DP is defined to contain specific roles and operations (**Def. 1**), additional information is ready to use, e.g. where the operation is declared in, what methods the operation invokes, and what other DPs the role is involved in. Not just reporting detected roles and operations is essential in some application scenarios such as visualization, detecting DP overlaps and to answer why a DP is detected by referring to the relationships between compilation units.

III. Evaluation

In this section, we evaluate the prototype tool SparT (Software Architectural Pattern Recognition Tool). (The letters “a” and “p” of the abbreviation are exchanged to be readable.) It was implemented based on the proposed approach. The evaluation follows the Goal Question Metric guidelines [28].

TABLE I. Open source software systems used for the evaluation

ID	System	#Files	#LOC	#Classes	#Methods
1	JHotDraw 5.1 (JHD)	144	8,419	173	1,332
2	JUnit 3.7 (JUN)	78	4,886	157	714
3	JRefactory 2.6.24 (JRF)	569	55,871	575	4,865
4	QuickUML 2001 (QUM)	156	9,249	228	1,096
5	PMD 1.8 (PMD)	446	41,321	505	3,680

A. Evaluation Setup

The goal of the evaluation was to assess the proposed approach to detect DP instances compared with rival approaches. To address our goal, the experimental evaluation aimed to answer the following research questions (RQ):

RQ1: What is the accuracy of the proposed approach to detect pattern instances?

RQ2: How is the accuracy of the proposed approach compared with other available approaches?

While RQ1 focuses on the accuracy of the proposed approach, RQ2 is to verify the accuracy compared with existing approaches in the literature. To assess the accuracy, we use **precision** to measure the fraction of relevant instances among detected instances and use **recall** to measure the fraction of relevant ones in detected instances among all relevant instances [29].

To answer the research questions, we carried out the experimental study on five open source systems as shown in Table I. The items in the table are measured by the Understand tool [30], including the number of code files (#Files), the number of lines of code (#LOC), the number of classes (#Classes) and the number of declared methods (#Methods). The selection of these systems is based on the following requirements: (i) they are publicly available online, (ii) they have been used to evaluate DP detection approaches in several studies in the literature [7] [11] [31] [32], (iii) the detailed pattern instances are reported independently by these studies, and (iv) they are implemented in Java since our approach focuses on Java systems.

In particular, to answer RQ2, we compared the detection results of SparT with the ones reported by Rasool and Mäder [31] (RaM), the DPD (Design Pattern Detection) tool [11], and the DPF (Design Pattern Finder) tool [32]. We choose these approaches since they published detected pattern instances of evaluated systems. More specifically, RaM was evaluated on systems ID1-4 in Table I; DPD was evaluated on systems ID1-3 and DPF was evaluated on systems ID1-5.

Two main barriers toward the evaluation are, first, there is no gold standard available although the research on DP detection is not new [1] [8] [33]; second, the detection results reported by different approaches are in different formats. As manually identifying the actual instances is expensive and prone to subjectiveness [7], to eliminate the first barrier, the benchmarks were constructed following the evaluation procedure proposed in existing researches [1] [7]. First, we conducted a thorough investigation on the instances reported by [7], [11], [31] and [32]. Second, three Ph.D. students independently analyzed the source code, documentation, and online resources to verify the instances reported by investigated approaches and our approach. At last, disagreed instances were finally decided with a discussion by the full group. To eliminate the second barrier, we unified the format of detection results, which enables a direct visual comparison of different approaches.

TABLE II. Number of pattern instances detected by RaM, DPD, DPF, and SparT

Approach	JHD		JUN		JRF		QUM		PMD	
	D [a]	***TP***	***D***	***TP***	***D***	***TP***	***D***	***TP***	***D***	***TP***
RaM	101	48	16	8	N/A	N/A	27	18	/	/
DPD	80	66	11	11	79	67	/	/	/	/
DPF	124	53	29	11	155	66	51	21	56	27
SparT	66	63	16	15	110	89	36	29	37	31
(BM) [b]	69		15		95		32		31	

a. "D": detected; "TP": true positives; BM: benchmark.

b. The detailed results on 22 types of DPs are provided online [22].

To put the proposed approach into practice, we have implemented SparT based on Jena, an open source Java framework for building semantic web and linked data applications [34]. The ontology models are composed of a series of OWL files and attached rule files. The rules and pattern templates are based on SPARQL update language [27] to query and manipulate the ontology data. The Ontology Generator integrates the JDT (Java Development Tools) based AST parser [35], and the generated ontologies are stored in Jena TDB (Triple Database). At last, the inference rules are executed by the Inference Engine, an ARQ based SPARQL processer [36].

The tool was implemented in an extensible way on the basis of the pluggable core methodology of the proposed approach. Initially, we progressively constructed the LKG that provides 202 vocabularies, based on which the pattern templates for 22 GoF patterns and their implementation variants were also built. The vocabularies also can be directly used to create new pattern templates.

We consider that the proposed approach is not applicable to Façade since the pattern lacks explicit structural and behavioral features in the guidelines of its implementation. In addition, same as the compared approaches, our approach does not distinguish the State and Strategy pattern since both their structure and behavior are quite similar. Finally, all the tests were performed on a developer desktop computer (Intel Core i7 3770, 8GB RAM). It needs little effort to submit the evaluated system to SparT, and all the subsequent processes are automatic.

B. Evaluation Results

For each system and approach, Table II shows the number of detected instances aggregating all the patterns. The extended version of the evaluation results presented in this section is provided as an online appendix [22]. Unavailable items are not reported ("N/A", "/"). We tried to download the instance repository of JRefactory reported by RaM, but the website link on [37] is broken ("N/A"). For QuickUML and PMD, not both RaM and DPD are evaluated on them ("/"). Based on the number of detected (D) and true (TP) instances in Table II, we calculated the precision (P) and recall (R) as shown in Table III. An additional group (Aggregated) of precision, recall, and F_1-

TABLE III. Comparison of RaM, DPD, DPF, and SparT in terms of precision and recall

Approach	JHD		JUN		JRF		QUM		PMD		(Aggregated)		
	P [a]	***R***	***P***	***R***	***P***	***R***	***P***	***R***	***P***	***R***	***P***	***R***	***FS***
RaM	47.5%	69.6%	50.0%	53.3%	N/A	N/A	66.7%	56.3%	/	/	51.4%	63.8%	56.9%
DPD	82.5%	**95.7%**	**100%**	73.3%	**84.8%**	70.5%	/	/	/	/	84.7%	80.4%	82.5%
DPF	42.7%	76.8%	37.9%	73.3%	42.6%	69.5%	41.2%	65.6%	48.2%	87.1%	42.9%	73.6%	54.2%
SparT	**95.5%**	91.3%	93.8%	**100%**	80.9%	**93.7%**	**80.6%**	**90.6%**	**83.8%**	**100%**	**85.7%**	**93.8%**	**89.5%**

a. “P”: precision; “R”: recall, “FS”: F_1-score.

score (FS) is reported by aggregating all the evaluated systems of each approach. The F_1-score is a harmonic average of the precision and recall, where they have equal weight [29].

1) RQ1: What Is the Accuracy of the Proposed Approach to Detect Pattern Instances?

In response to RQ1, the results presented in Table III demonstrate that SparT achieved very high precision and recall. We can observe from Table III that SparT obtains a precision of 85.7% and a recall of 93.8% on average. In all the cases, the precision is greater than 80.6% and the recall is greater than 90.6%. Most of the false positives, i.e. 34 of 38, are Adapter (9 of 38) and State/Strategy (25 of 38) instances. The main reason is, as the delegation is involved in several DPs such as Builder, Prototype and Proxy etc., Adapter and State/Strategy are in lack of unique structural and behavioral features in the implementation as the clues to distinguish from other patterns and non-patterns. In particular, we allow the inexistence of some roles, which adapts specific implementation variants but may weaken the characteristics of a pattern. For example, as we use key roles to uniquely identify a pattern instance, the Object Adapter pattern with the key roles Adapter and Adaptee has the same structure and behavior with the State pattern with the key roles Context and State.

It is worth mentioning that SparT reported additional instances which were not detected by rival approaches. For QuickUML, two more Observer instances were reported, whose Subject / Observer role is AbstractDiagramModel / DiagramModelListener and AbstractSelectionModel / DiagramSelectionListener respectively. The implementation of the Subject role is based on a customized WeakList that extends JDK AbstractList. The aggregation that uses WeakList was captured in the inference process of our approach but RaM and DPF failed to detect. An Iterator instance missed by DPF was also detected for PMD, whose ConcreteAggregate role DocumentNavigator instantiates the ConcreteIterator role AttributeAxisIterator.

2) RQ2: How Is the Accuracy of the Proposed Approach Compared with Other Available Approaches?

In response to RQ2, the comparison of the results reported in Table III shows that the average precision of SparT is slightly better than the best competitor, DPD, and the average recall of SparT outperforms DPD. We can observe that, for JHotDraw, the precision of SparT (95.5%) is significantly better than DPD (82.5%), and the recall (91.3%) is close to it (95.7%). For JUnit and JRefactory, the precision of SparT (93.8%, 80.9%) is close to DPD (100%, 84.8%), and the recall (100%, 93.7%) is significantly better than it (73.3%, 70.5%). In average, SparT obtains similar precision with DPD (85.7% vs. 84.7%), but the recall is significantly better (93.8% vs. 80.4%). Finally, we can observe that SparT obtains a better F_1-score (89.5%) than DPD (82.5%).

For JHotDraw, DPD reported 39 State/Strategy instances which include 14 false positives. SparT detected 2 less true positives (23), but only reported 1 false positive. SparT achieves better precision since it employs flexible search strategies to offset the lack of unique features, e.g. constraining the Request operation to actually invoke the Handle operation, filtering out Context and State roles which reside in an inheritance hierarchy to distinguish from Composite and Decorator pattern, and semantically filtering out the delegation with clear intent (e.g. indicating by the field name “builder”, “singleton” or “prototype”) etc. For JUnit, SparT also reported two more true instances of Adapter than DPD, i.e. TestRunner/CounterPanel (junit.swingui) and TestRunner/StatusLine (junit.swingui) (Adapter and Adaptee role respectively). The Adapter role TestRunner and the Target role TestListener reside in a three-layer inheritance hierarchy via an abstract class BaseTestRunner. In the proposed approach, the pattern template only needs to apply the *isA* property combining with other constraints to detect multilayer variants.

3) Runtime Performance

Table IV presents the execution time of the DP detection processes for each system. The Ontology Generation and Knowledge Inference processes only execute once for each system as the input of the Pattern Template Matching process. The presented time of the Pattern Template Matching process is the sum of all the 22 pattern templates.

As aforementioned, the Ontology Generation process enumerates the compilation units and fills the ontologies, in the search space of which the Knowledge Inference and Matching processes infer composite vocabularies and match the pattern templates. Thus, the execution time depends on the size of the systems and the composition relationships of compilation units. We can observe from Table IV that the Ontology Generation and Pattern Template Matching processes dominate the time consumption. As for the DPs, aggregating all systems, the most time-consuming pattern is Adapter. The main reason is that its pattern template composites a group of implementation variants to offset the lack of unique characteristics, which raises the search cost. Table IV shows that the execution time increases with respect to the number of classes (#Classes in Table I). Moreover, SparT presents good scalability on runtime performance as the size of the evaluated systems increases.

DPD reports its preprocessing and detection time on JUnit, JHotDraw, and JRefactory. On JUnit and JHotDraw, the Ontology Generation and Knowledge Inference phase of SparT

TABLE IV. EXECUTION TIME OF THE DESIGN PATTERN DETECTION PROCESSES

Process [a]	JHD	JUN	JRF	QUM	PMD
Ontology Generation	0.77	0.36	3.05	0.98	2.69
Knowledge Inference	0.07	0.04	0.22	0.06	0.20
Pattern Template Matching [b]	0.89	0.42	2.95	0.49	2.10
(Total)	1.74	0.82	6.22	1.53	4.99

a. The time is measured in seconds.

b. The presented time is the sum of all the 22 pattern templates.

take more time (0.40, 0.84) than the preprocessing phase of DPD (0.27, 0.51) since SparT extracts compilation units down to statement and expression level; SparT outperforms DPD when scales to JRefactory (3.27 vs. 3.80). The matching phase of SparT outperforms the detection phase of DPD on all the patterns reported by DPD. We do not compare the time performance with RaM and DPF since their tools and execution time are unavailable.

IV. Discussion

Naming conventions. We considered that the structural and behavioral aspects are more reliable than naming conventions. We did not assume that the classes, methods or fields must follow specific naming rules or must not since naming conventions are a double-edged sword for the detection. In the evaluated pattern templates, only method and field names that are clear and explicit were used as an assistant, e.g. the field “prototype” but not its acronyms or abbreviations.

Linguistic and dynamic extensions. Referring to the semantic and temporal aspects of DPs, existing researches support potential extensions of the proposed approach. While we used simple semantic information, the extensibility of the LKG allows additional ontology models, even new layers to describe linguistic aspects of DPs [38].

The sequence of operations is another important aspect referring to the behaviors of DPs. Separate method invocation events extracted through static analysis imply possible method invocation sequence. For example, the facts (a *callsMethod* b) and (b *callsMethod* c) imply a possible invocation path: a, b and then c. However, the actual behaviors depend on the input data and may not be decided until runtime.

In a typical operation sequence of Observer, serval observers are attached to the subject; after a state change, the subject notifies all observers. For the implementation that clearly expresses the state, we can specify a variable assignment event as the clue of a state change. In consequence, it introduces a temporal description of different events. Based on existing work [39] [40], the ontologies are able to describe runtime behaviors by incorporating temporal data. A detailed discussion of these extensions is beyond the focus of this work; however, they are worth further investigation to improve the accuracy.

V. Threats to Validity

The report format of detection results could be a threat to internal validity. As the accuracy is evaluated by comparing detected instances with actual instances, the difference in the formats of the instances reported by different approaches results in inconsistent numbers of instances although the results are actually the same. The report of pattern instances varies in required roles and how to distinguish different instances of the same pattern. Thus, the accuracy cannot be evaluated and compared properly only by the number of instances. In the evaluation, we use key roles to uniquely identify an instance based on the formal definition of DPs. A thorough investigation was conducted on the instances reported by RaM, DPD, DPF, and SparT. Finally, the results were made uniform for the evaluation of accuracy and the comparison of these approaches.

A threat that could affect the external validity is the selection of software systems. We notice that DP detection approaches have been evaluated on many other systems over the years, e.g. Apache Ant [41], Java AWT [42] and Log4J [43] etc. While some of these systems are still active in the development community, we cannot download the evaluated versions in the literature since they are no longer maintained. In other cases, only the number of instances detected in the systems are reported in the literature. In consideration of study reproducibility, we have published all available resources online [22].

VI. Related Work

In this section, we discuss related techniques and tools proposed in the literature.

DP detection techniques. As structural features are an important aspect of DPs, structural analysis is considered as the basic technique underlying the approaches based on, for instance, graph matching [1] [11] [32] [42] and similar matrix [44]. In approaches based on graph matching, the classes and their relationships are represented as nodes and edges. The candidates are matched with DPs through graph theory algorithms including decomposition and isomorphism [45]. Unlike graph matching, approaches based on similar matrix define structural characteristics in terms of weight and matrix.

Actually, structural analysis techniques are seldom applied alone in these approaches, since the DPs in similar structures are difficult to distinguish only through structural characteristics. The approaches that exploit both structural and behavioral characteristics of DPs report higher precision [18]. While a set of approaches analyze static behaviors in the source code [46] [47] [48] [49] [50], the others execute the software programs and trace their dynamic behaviors [7] [18] [51] [52]. Several techniques are proposed, e.g. model checking [7] [13] and Prolog [18] [25], for the representation and analysis of dynamic behaviors.

However, because of the problems of runtime efficiency and code coverage of test cases, in practice, dynamic analysis is often used to assist structural analysis in reducing false positives [53].

Another technique, software metrics [10] [12] [54] [55], is also employed to aid the detection since the use of DPs is likely to exhibit good properties. Besides graph matching, the search of pattern instances is performed by various other techniques such as SQL (Structured Query Language) queries [31] [46] [49] and rules [56] [57] according to the representation of software systems and DPs.

Various other techniques include visual language parsing [48], text processing [4] [19] and machine learning [14] [24]

[58] [59]. Visual language parsing techniques are usually applied to identify the structures and relationships of class diagrams extracted from the source code through static analysis [48]. In contrast, text processing techniques are more concerned and appropriate in semantic aspects. Rasool et al. [19] put annotations into the source code, which are parsed later to facilitate the detection of structural DPs with regular expressions. Focusing on the problem of DP selection, Hussain et al. [4] organize DPs based on text categorization techniques to automatically suggest more appropriate DPs for a given design problem.

Nevertheless, several of these approaches claim to be language-independent [1], and some others do not exploit domain knowledge [18]. Thus, they are not able to capture language-specific features. In fact, most approaches rely on a language-specific parser, but the language aspects are not well expressed to be general and customizable.

A recent work [14] leverages deep learning algorithms for the organization and selection of DPs based on text categorization. To reduce the size of training examples for DP detection, a clustering algorithm is proposed by Dong et al. [58] based on decision tree learning. While [59] purely employs artificial neural networks to perform the detection, [24] uses machine learning aiming to enhance DP detection by filtering out false positives that cannot be structurally distinguished.

However, the applicability of machine learning approaches suffers from the drawbacks and limitations of collecting datasets and training machine learning models [58]. As a contrast, our approach is driven by inference rules that reflect the knowledge of domain experts, thus it has the advantage of interpretability.

Presented tools. Based on these techniques, many tools are presented over the decades. The aforementioned tools are examples that employ static analysis techniques: while DPD [11] and DPF [32] are based on similar matrix and graph matching respectively, RaM [31] combines multiple search techniques. In DPF, the structures of software systems and DPs are modeled with a domain specific language, after which complete ASTs are built and the system is represented as a graph. Finally, the detection is performed by traversing the graph with a graph matching algorithm. As a contrast, the underlying methodology of DPD is also graph matching, but graphs are represented as matrices to calculate the similarity score of graph vertices. As for RaM, structural and behavioral characteristics of DPs are represented with XML (Extensive Markup Language). In consequence, multiple search techniques are used in the detection stage, including SQL and regular expression.

Another tool Ptidej, presented by Gueheneuc et al. [15], uses metrics values as the input of a machine learning algorithm which generates a set of rules to characterize pattern roles. The tool MARPLE, which is presented in a two-phase (training phase and test phase) approach [9], also applies machine learning techniques. PRAssistor is a tool based on dynamic behavior analysis [18]. It employs Prolog to characterize the temporal aspect of behaviors. The tool ePAD presented in a recent work [7] verifies the behavioral aspects of candidates through model checking. DPD, Ptidej and four other tools are investigated and evaluated [60]. As a conclusion, the authors recommend that tools should be flexible for customization.

In this work, we present a practical approach based on static analysis, which exploits structural, behavioral and semantic characteristics of DPs. It supports flexible search strategies and allows the customization of pattern templates.

VII. Conclusions and Future Work

We have presented an ontology-based approach for DP detection in the context of Java language. While existing approaches reach a consensus on matching both structural and behavioral characteristics of DPs for the detection, the failure to capture language-specific implementation leads to false positives or false negatives during the matching. To represent the domain knowledge of Java, the proposed approach constructs the LKG as the blueprint of ontology vocabularies and sequenced inference. The DPs are formally defined taking into consideration the problems of missing roles and pattern instance identification. To match implementation idioms, the pattern templates are able to employ flexible search strategies that integrate structural, behavioral and semantic aspects of DPs. Moreover, the core methodology of the proposed approach enables pluggable vocabulary handlers and pattern templates to support the evolution of itself and the detection of emerging patterns and pattern variants.

A prototype implementation of the proposed approach, i.e. SparT, has been evaluated on five open source software systems and compared with three other approaches. A thorough investigation has been conducted on the pattern instances reported by these approaches for the comparison. The evaluation results demonstrate that the proposed approach achieves better accuracy than the best rival DPD. SparT also exhibits good practical applicability on runtime performance. We have deployed SparT online and visualized the detection process [61].

The future work is twofold. First, based on the representation ability of ontologies, we can further reduce the number of false instances by exploiting semantic information in comments, annotations, and available documentation. To achieve this, the consistency between code and comments will be assessed, and text processing techniques can be employed. Second, as new pattern templates are naturally supported by the proposed approach, we plan to ease the creation of pattern templates for practical use. SparT will be enhanced with a pattern template builder that is able to visually construct pattern templates with mouse clicks in form of graphs.

Acknowledgment

We would like to thank the reviewers. Their insightful comments helped to improve this paper. This work was supported partially by the National Key R&D Program of China under Grand No. 2018YFB1003902, the National Natural Science Foundation of China under Grand No. 61572126, No. 61872078 and No. 61402103.

References

[1] B. B. Mayvan and A. Rasoolzadegan, “Design Pattern Detection Based on the Graph Theory,” *Knowledge-Based Systems*, vol. 120, pp. 211–225, 2017.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] R. K. L. Bass, P. Clements, *Software Architecture in Practice, 3rd ed.* Addison-Wesley Professional, 2012.

[4] S. Hussain, J. Keung, and A. A. Khan, “Software Design Patterns Classification and Selection Using Text Categorization Approach,” *Applied Soft Computing*, vol. 58, pp. 225–244, 2017.

[5] A. Ampatzoglouaba, “A methodology to assess the impact of design patterns on software quality,” *Information & Software Technology*, vol. 54, no. 4, pp. 331–346, 2012.

[6] J. Dong, Y. Zhao, and T. Peng, “A review of design pattern mining techniques,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 6, pp. 823–855, 2009.

[7] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, “Detecting the behavior of design patterns through model checking and dynamic analysis,” *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 4, pp. 1–41, 2018.

[8] D. Yu, P. Zhang, J. Yang, Z. Chen, C. Liu, and J. Chen, “Efficiently Detecting Structural Design Pattern Instances Based on Ordered Sequences,” *Journal of Systems and Software*, vol. 142, pp. 35–56, 2018.

[9] M. Zanoni, F. Arcelli Fontana, and F. Stella, “On applying machine learning techniques for design pattern detection,” *Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.

[10] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem, “Object-oriented design patterns recovery,” *Journal of Systems and Software*, vol. 59, no. 2, pp. 181–196, 2001.

[11] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, “Design pattern detection using similarity scoring,” *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[12] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, “Predicting the existence of design patterns based on semantics and metrics,” *International Arab Journal of Information Technology*, vol. 13, no. 2, pp. 310–319, 2016.

[13] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, “Improving behavioral design pattern detection through model checking,” in *European Conference on Software Maintenance and Reengineering*, ser. European Conference on Software Maintenance and Reengineering. Los Alamitos: IEEE Computer Soc, 2010, pp. 176–185.

[14] S. Hussain, J. Keung, A. A. Khan, A. Ahmad, S. Cuomo, F. Piccialli, G. Jeon, and A. Akhunzada, “Implications of Deep Learning for the Automation of Design Patterns Organization,” *Journal of Parallel and Distributed Computing*, vol. 117, pp. 256–266, 2018.

[15] Y. G. Gueheneuc, H. Sahraoui, and F. Zaidi, “Fingerprinting design patterns.” Los Alamitos: IEEE Computer Soc, 2004, pp. 172–181.

[16] J. Niere, W. Schafer, J. P. Wadsack, L. Wendehals, and J. B.-G.-A. A. A. Welsh, “Towards Pattern-Based Design Recovery,” pp. 338–348, 2002.

[17] Oracle, “Java API Specifications,” https://www.oracle.com/technetwork/java/api-141528.html, accessed: 2018-10-10.

[18] H. Y. Huang, S. S. Zhang, J. Cao, and Y. H. Duan, “A practical pattern recovery approach based on both structural, and behavioral analysis,” *Journal of Systems and Software*, vol. 75, no. 1-2, pp. 69–87, 2005.

[19] G. Rasool, I. Philippow, and P. Maeder, “Design pattern recovery based on annotations,” *Advances in Engineering Software*, vol. 41, no. 4, pp. 519–526, 2010.

[20] J. Gil and I. Maman, “Micro patterns in java code,” *Acm Sigplan Notices*, vol. 40, no. 10, pp. 97–116, 2005.

[21] B. Eckel, *Thinking in Java, 4th Edition*. Prentice Hall PTR, 2005.

[22] Renhao Xiong, Bixin Li, “Dataset for: Accurate Design Pattern Detection Based on Idiomatic Implementation Matching in Java Language Context,” https://github.com/Megre/Dataset4SparT-SANER-26th, accessed: 2018-12-07.

[23] G. Antoniou and F. V. Harmelen, *A Semantic Web Primer, 2nd Edition (Cooperative Information Systems)*. The MIT Press, 2008.

[24] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, “Design pattern mining enhanced by machine learning,” in *Proceedings-IEEE International Conference on Software Maintenance*, ser. Proceedings-IEEE International Conference on Software Maintenance. Los Alamitos: IEEE Computer Soc, 2005, pp. 295–304.

[25] A. Binun and G. Kniesel, “Dpjf - design pattern detection with high accuracy,” in *European Conference on Software Maintenance and Reengineering*, ser. European Conference on Software Maintenance and Reengineering, T. Mens, A. Cleve, and R. Ferenc, Eds. New York: IEEE, 2012, pp. 245–254.

[26] S. J. Metsker and W. C. Wake, *Design Patterns in Java*. Addison-Wesley Professional, 2006.

[27] World Wide Web Consortium, “SPARQL 1.1 Update,” https://www.w3.org/TR/sparql11-update, 2013.

[28] V. R. B.-G. Caldiera and H. D. Rombach, “Goal question metric paradigm,” *Encyclopedia of software engineering*, vol. 1, pp. 528–532, 1994.

[29] C. Sammut and G. I. Webb, *Encyclopedia of Machine Learning*. Springer Science & Business Media, 2011.

[30] E. Ronchieri, M. G. Pia, and F. Giacomini, “First Statistical Analysis of Geant4 Quality Software Metrics,” in *21st International Conference on Computing in High Energy and Nuclear Physics (Chep2015), Parts 1-9*, ser. Journal of Physics Conference Series, vol. 664, 2015.

[31] G. Rasool and P. Mäder, “A customizable approach to design patterns recognition based on feature types,” *Arabian Journal for Science and Engineering*, vol. 39, no. 12, pp. 8851–8873, 2014.

[32] M. L. Bernardi, M. Cimitile, and G. Di Lucca, “Design pattern detection using a dsl-driven graph matching approach,” *Journal of Software-Evolution and Process*, vol. 26, no. 12, pp. 1233–1266, 2014.

[33] B. B. Mayvan, A. Rasoolzadegan, and Z. G. Yazdi, “The State of the Art on Design Patterns: A Systematic Mapping of the Literature,” *Journal of Systems and Software*, vol. 125, pp. 93–118, 2017.

[34] Apache Software Foundation, “Jena Ontology API,” http://jena.apache.org/documentation/ontology, accessed: 2018-06-10.

[35] Oracle, "Eclipse Java Development Tools (JDT)," http://www.eclipse.org/jdt, accessed: 2018-08-01.

[36] Apache Software, "ARQ - A SPARQL Processor for Jena," http://jena.apache.org/documentation/query/index.html, accessed: 2018-07-28.

[37] G. Rasool and P. Mäder, "Pattern Instances Reported by Rasool and Mäder," https://lahore.comsats.edu.pk/research/groups/serc/designpatterns.aspx, accessed: 2018-10-10.

[38] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, "A Design Pattern Detection Approach Based on Semantics," in *Software Engineering Research, Management and Applications 2012*, ser. Studies in Computational Intelligence, Lee, R, Ed., vol. 430, 2012, pp. 49–63.

[39] J. Tappolet and A. Bernstein, "Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL," in *Semantic Web: Research and Applications*, ser. Lecture Notes in Computer Science, vol. 5554, 2009, pp. 308–322, 6th European Sematic Web Conference, Heraklion, GREECE, MAY 31-JUN 04, 2009.

[40] C. Gutierrez, C. Hurtado, and A. Vaisman, "Temporal RDF," in *Semantic Web: Research and Applications, Proceedings*, ser. Lecture Notes in Computer Science, GomezPerez, A and Euzenat, J, Ed., vol. 3532, 2005, pp. 93–107, 2nd European Semantic Web Conference, Heraklion, GREECE, MAY 29-JUN 01, 2005.

[41] M. Esmaeilpour, V. Naderifar, and Z. Shukur, "Design pattern mining using distributed learning automata and dna sequence alignment," *Plos One*, vol. 9, no. e1063139, 2014.

[42] M. Oruc, F. Akal, and H. Sever, "Detecting design patterns in object-oriented design models by using a graph mining approach," R. JuarezRamirez, S. J. Calleros, H. J. Oktaba, C. Fernandez, R. A. Vera, G. L. Sandoval, and J. A. Cisneros, Eds. New York: IEEE, 2016, pp. 115–121.

[43] S. Z. Yang, A. Manzer, and V. Tzerpos, "Measuring the quality of design pattern detection results." NEW YORK: IEEE, 2015, pp. 53–62.

[44] J. Dong, Y. Zhao, and Y. Sun, "A Matrix-Based Approach to Recovering Design Patterns," *IEEE Transactions on Systems Man and Cybernetics Part A-Systems and Humans*, vol. 39, no. 6, pp. 1271–1282, 2009.

[45] A. Pande, M. Gupta, and A. K. Tripathi, "A new approach for detecting design patterns by graph decomposition and graph isomorphism," in *Communications in Computer and Information Science*, ser. Communications in Computer and Information Science, S. Ranka, A. Banerjee, K. K. Biswas, S. Dua, P. Mishra, R. Moona, S. H. Poon, and C. L. Wang, Eds. Berlin: Springer-Verlag Berlin, 2010, vol. 95, pp. 108–119.

[46] K. A. Mohamed and A. Kamel, "Reverse Engineering State and Strategy Design Patterns Using Static Code Analysis," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 1, pp. 568–576, 2018.

[47] C. Park, Y. H. Kang, C. S. Wu, and K. K. Yi, "A static reference flow analysis to understand design pattern behavior." Los Alamitos: IEEE Computer Soc, 2004, pp. 300–301.

[48] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, vol. 82, no. 7, pp. 1177–1193, 2009.

[49] K. Stencel and P. Wegrzynowicz, "Detection of diverse design pattern variants," in *Asia-Pacific Software Engineering Conference*, ser. Asia-Pacific Software Engineering Conference. Los Alamitos: IEEE Computer Soc, 2008, pp. 25–+.

[50] Z. Balanyi and R. Ferenc, "Mining design patterns from c++ source code," in *Proceedings-IEEE International Conference on Software Maintenance*, ser. Proceedings-IEEE International Conference on Software Maintenance. Los Alamitos: IEEE Computer Soc, 2003, pp. 305–314.

[51] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," in *International Workshop on Program Comprehension*, ser. International Workshop on Program Comprehension. Los Alamitos: IEEE Computer Soc, 2003, pp. 94–103.

[52] H. Zhu, I. Bayley, L. J. Shan, and R. Amphlett, "Tool support for design pattern recognition at model level," in *Proceedings International Computer Software and Applications Conference*, ser. Proceedings International Computer Software and Applications Conference. New York: IEEE, 2009, pp. 228–+.

[53] F. Arcelli, F. Perin, C. Raibulet, and S. Ravani, "Design pattern detection in java systems: A dynamic analysis based approach," in *Communications in Computer and Information Science*, ser. Communications in Computer and Information Science, L. A. Maciaszek, C. GonzalezPerez, and S. Jablonski, Eds. Berlin: Springer-Verlag Berlin, 2010, vol. 69, pp. 163–+.

[54] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, "Using metric-based filtering to improve design pattern detection approaches," *Innovations in Systems and Software Engineering*, vol. 11, no. 1, pp. 39–53, 2015.

[55] F. Arcelli Fontana and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," vol. 181, no. 7, pp. 1306–1324, 2011.

[56] B. Di Martino and A. Esposito, "A rule-based procedure for automatic recognition of design patterns in uml diagrams," *Software-Practice & Experience*, vol. 46, no. 7, pp. 983–1007, 2016.

[57] O. AlSheikSalem and H. Qattous, "An expert system for design patterns recognition," *International Journal of Computer Science and Network Security*, vol. 17, no. 1, pp. 93–101, 2017.

[58] J. Dong, Y. T. Sun, and Y. J. Zhao, "Compound record clustering algorithm for design pattern detection by decision tree learning." New York: IEEE, 2008, pp. 226–+.

[59] S. Alhusain, S. Coupland, R. John, and M. Kavanagh, "Towards machine learning based design pattern recognition," Y. Jin and S. A. Thomas, Eds. New York: IEEE, 2013, pp. 244–251.

[60] G. Rasool, P. Maeder, and I. Philippow, "Evaluation of design pattern recovery tools," in *Procedia Computer Science*, ser. Procedia Computer Science, A. Karahoca and S. Kanbul, Eds. Amsterdam: Elsevier Science Bv, 2011, vol. 3, pp. 813–819.

[61] Renhao Xiong, Bixin Li, "Software Architectural Pattern Recognition Tool (SparT)," http://www.spart.group, accessed: 2018-10-10.