Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1559–1566

# Relaxing Queries to Detect Variants of Design Patterns

Patrycja Węgrzynowicz, Krzysztof Stencel
Institute of Informatics
University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
Email: {patrycja, stencel}@mimuw.edu.pl

***Abstract*—Design patterns codify general solutions to frequently encountered design problems. They also facilitate writing robust and readable code. Their usage happens to be particularly profitable if the documentation of the resulting system is lost, inaccurate or out of date. In reverse engineering, detection of instances of design patterns is extremely helpful as it aids grasping high level design ideas. However, the actual instances of design patterns can diverge from their canonical textbook templates. Useful pattern detection tools should thus be able to identify not only orthodox implementations but also their disparate variants. In this paper, we present a method to generate queries to detect canonical instances of design patters. We formulate these queries so that they are language-agnostic. They precisely reflect the intents of the canonical implementations of design patterns. However, they abstract from any peculiarities of programming languages. Next, we show a systematic technique to relax these queries so that they also cover variant implementations of patterns. We discuss our proof-of-concept implementation of this approach in our prototype tool D-CUBED. Finally, we report the results of an experimental comparison of D-CUBED and state-of-the-art detectors.**

## I. Introduction

A DESIGN pattern [1] is a general reusable solution to a commonly occurring problem in software design. Design patterns facilitate forming quality designs. As well as being useful in the construction of software systems (forward engineering), they also aid analysing existing systems (reverse engineering).

Detection of design patterns is an important part of reverse engineering. There are a significant number of large software systems without proper documentation that nevertheless need to be maintained, extended, or modified. In such cases, reverse engineering is necessary. However, the process is usually time-consuming and error-prone, as most of the core analysis must be performed manually and some important aspects can be omitted. Detection of design patterns automates extraction of high-level design concepts, which helps gaining a better understanding of code and makes analysis more efficient in terms of time and cost. Moreover, detection of design patterns can aid documenting code (e.g., generating or verifying documentation) or assessing its quality (e.g., using metrics based on design patterns).

In recent years, we have observed a continual improvement in the field of automatic detection of design patterns in source code. Existing approaches [2]–[13] can detect a fairly broad range of design patterns, targeting structural as well as behavioural aspects of patterns. Until recently, the research in the field of pattern detection has focused on novel approaches. However, the papers [14], [15] highlight the importance of the accuracy (precision and recall) of detection methods.

To achieve high recall, we need to reduce the number of false negative results. For a design pattern detection method, this minimization of false negatives means that the method should be capable of detection of numerous implementation variants that preserve the meaning of a design pattern, even though the details of their implementations do not follow the canonical implementation. Therefore, following the advice of [14] that emphasises the importance of '*a common set of patterns, both structural as well as behavioural, with well-defined implementation variants*', we focus on a systematic approach to detect variants of design pattern.

The analysis of implementation variants revealed highly diverse ecosystem of possibilities. For a simple design pattern like the Singleton, we have identified 7 elemental variants as presented in [16]. For another design pattern like the Visitor, we have also identified several significantly different implementations as presented in [17]. Considering further combinations of those elemental variants, it seemed infeasible to enumerate all available variants of design patterns, thus we sought an automated way to generate them. As a starting point we assumed the canonical implementation of design patterns as described in [1]. In [18], we introduced *pattern-preserving transformations* that enable transforming an implementation variant of a design pattern into a new one while preserving the design pattern.

In this paper, we extend our method of detecting design patterns (based on first-order logic formulae as described in [13]) to include a systematic approach to relaxing queries capable of detecting implementation variants of design patterns.

Contributions of this paper are as follows:

1) We present a systematic approach to the construction of queries to detect implementation variants of design patterns. The approach is based on the application of specific and generic pattern-preserving transformations of a Prolog query representing the canonical variant of a design pattern.
2) As a proof-of-concept for Contribution 1, we present queries capable of detecting variants of the Singleton

978-1-4673-4471-5/$25.00 © 2013, IEEE

design pattern.
3) We evaluate our prototype tool (D-CUBED) with relaxed variant queries and compare it with two state-of-the-art pattern detectors (PINOT, DPD Tool).

## II. METHOD

Our method of detecting variants of design patterns consists of the following steps (see Figure 1):

1) A transformation of the UML class diagram of the canonical implementation of a design pattern to a logic query;
2) An application of specific pattern-preserving query transformations;
3) An application of generic pattern-preserving query transformations;

The starting point of the method is the UML class diagram of *the canonical implementation* of a design pattern. We define *a canonical implementation* as the original implementation provided by the authors of a design pattern (e.g., Gang of Four in their book [1]). Through a set of transformations applied to the UML diagram of the canonical implementation of a design pattern, we obtain a disjunction query used to detect the variants of this pattern.

In [18], we introduced the concept of *pattern-preserving code transformations*, i.e., code transformations that preserve the intent of a design pattern. They were used to generate the implementation variants of design patterns in order to create test cases for pattern detectors. Here, we introduce a corresponding concept of *pattern-preserving query transformations*, which relax queries to extend their capabilities of detecting more implementation variants of design patterns.

In Step 1: Direct Transformation of UML to Logic, we transform the class diagram of the canonical implementation of a design pattern represented in UML to a formula using our custom metamodel (see Section II-A) expressed in logic. The output formula is a conjunction of predicates from our metamodel, describing such features as classes along with their fields and methods. Occasionally, we may need to add additional clauses to a conjunction query in order to represent important implementation details of a design pattern which are described in UML comments, other UML diagrams, or in the description of the pattern.

In Step 2: Specific Pattern-Preserving Query Transformations, we apply *specific pattern-preserving query transformations* to the query constructed in Step 1. *Specific pattern-preserving query transformations* are based on the concept of *specific pattern-preserving transformations* introduced in [18]. Each of them is a code transformations characteristic to a particular design pattern that preserves its intent. If such a transformation is applied to a correct implementation variant of a design pattern, it will produce a new correct variant. Exemplary specific pattern-preserving query transformations can be found in Section III.

In Step 3: Generic Pattern-Preserving Query Transformations, we apply *generic pattern-preserving query transformations* to the queries obtained in Step 2. Similarly to a specific pattern-preserving transformation, a *generic pattern-preserving query transformation* is based on the concept of a *generic pattern-preserving transformation* introduced in [18]. It is a code transformation referring to generic programming (e.g., abstractness, invocation, access modifiers) that preserves the design intent. The detailed description of generic pattern-preserving query transformations can be found in Section II-D.

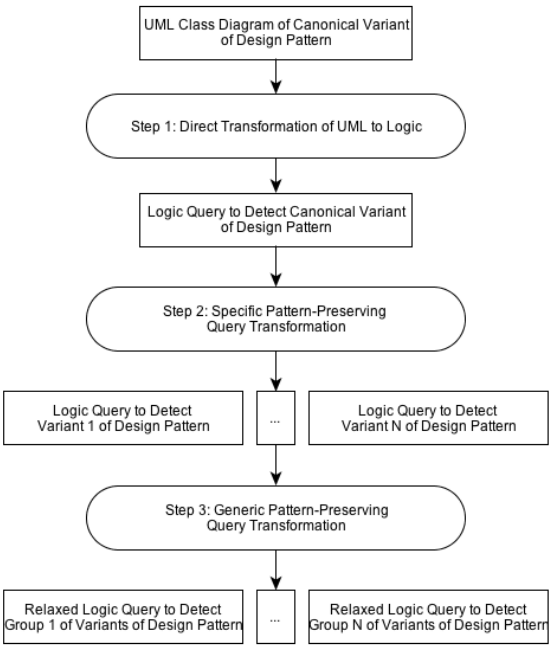


Fig. 1. The overview of the construction of the queries to detect the variants of a design pattern.

The final query to detect the variants of a given design pattern is the disjunction of the relaxed queries constructed in Step 3.

### A. Program Metamodel

The program metamodel used in our detection method has been introduced in [13]. It consists of a set of core elements and a set of relationships among those elements, both structural and behavioural. The metamodel has been designed to be *"as simple as possible, but not simpler"*, yet it is powerful enough to model a large set of object-oriented languages.

The program metamodel consists of the following core elements (Figure 2): types, fields-or-variables, operations, and instances. Most of them have their obvious object-oriented meaning. A type denotes either a class, an interface or any other language-specific data type construct (like Java enum). A field-or-variable includes two cases: (1) an instance field or static field of a class or an interface, and (2) a variable that is not a field e.g. a global variable (irrelevant to Java but not to C++). Similar to the field-or-variable element, an operation also covers two cases: (1) a method declared in a type, and (2) a function e.g., a global function. An instance has been defined as an equivalence class of the relation "objects

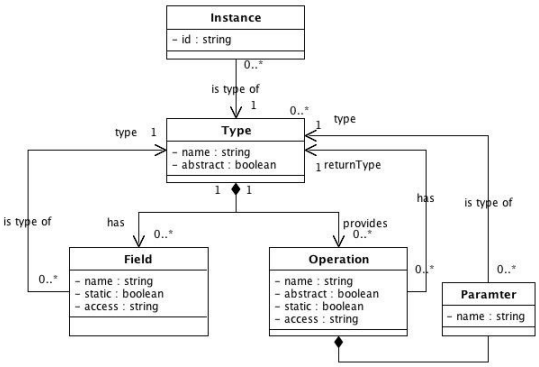


Fig. 2. The core elements of the program metamodel

constructed by the same `new`". All objects instantiated by the same `new` statement are treated as a single instance.

The extensional Prolog predicates describing core elements include: $isClass$, $isInterface$, $isEnum$, $isPrimitive$, $isType$, $isInstance$, $isField$, $isVariable$, $isFunction$, $isMethod$, $isParameter$, and others.

The elemental structural relations describe the relationships among core elements of a program, including memberships (i.e., fields and methods), modifiers (e.g., static, abstract, and access modifiers), and type system. The extensional Prolog predicates representing the elemental structural relations include: $isFieldOf$, $isMethodOf$, $hasParameter$, $hasReturnType$, $hasModifier$, $isTypeOf$, $isSubtypeOf$, and others. The intensional predicates $isTypeOf^*$ and $isSubtypeOf^*$ are the transitive closures of $isTypeOf$ and $isSubtypeOf$ respectively.

The elemental behavioural relations mostly refer to a data-flow and a call-flow, including instantiation as a specific call to a `new` operator. The extensional Prolog predicates representing the elemental behavioural relations include: $invokes$, $instantiates$, $hasInput$, $hasOutput$, and others. The intensional predicates $invokes^*$ and $instantiates^*$ are the transitive closures of $invokes$ and $instantiates$ respectively.

In order to illustrate the semantics of the predicates representing relations, here we present the definitions of $hasInput$ and $hasOutput$, two exemplary predicates related to the call-flow of a program:

$hasInput$

$hasInput(F, I)$, if and only if there is a potential execution path where the instance $I$ is passed as one of the input parameters or as a part of an input parameter to the operation $F$.
$hasInput(F, T)$, if and only if $hasInput(F, I)$ and $isTypeOf^*(I, T)$.

$hasOutput$

$hasOutput(F, I)$, if and only if there is a potential execution path where the instance $I$ is the output value or a part of the output value of the call to the operation $F$. The output means a return value as well as an output parameter.
$hasOutput(F, T)$, if and only if $hasOutput(F, I)$ and $isTypeOf^*(I, T)$.

### B. UML to Logic Transformation

By a logic query we understand a set of Horn [19] clauses as used in logic programming. These logic queries operate on the program metamodel from Section II-A. We treat UML class diagrams of design patterns as inquires to a codebase. Therefore, we translate the codebase queries in the form of the UML class diagrams to the logic queries operating on the metamodel.

The algorithm to transform a class diagram to a logic query is as follows:

1) For each class and interface in a class diagram, we produce a conjunction of predicates, describing the type ($isClass$ or $isInterface$), its supertypes ($isSubtypeOf$), and its modifiers ($hasModifier$).
2) For each method in a class or interface, we produce a conjunction of predicates, describing its enclosing type ($isMethodOf$), its signature ($numberOfParameters$, $hasParameter$, $hasReturnType$), and its modifiers ($hasModifier$).
3) For each field in a class or interface, we produce a conjunction of predicates, describing its enclosing type ($isFieldOf$), its type ($isTypeOf$), and its modifiers ($hasModifier$).
4) We model additional information (e.g., comments) contained in a class diagram on per-case basis.
5) The output query is the conjunction of previously generated queries (i.e., class queries, method queries, field queries, additional information queries).

Optionally, we may need to transform to a logic query other UML diagrams (e.g., a sequence diagram) of a design pattern or encode features described purely textually.

### C. Specific Query Transformations

A *specific pattern-preserving query transformation* is characteristic to a particular design pattern. It deals with, so-called, query logical fragments (i.e. parts of a query representing logical fragments of a pattern) transforming them into queries corresponding to different implementations valid in the context of a pattern.

A *pattern logical fragment* is a robust fragment of a design pattern which can be implemented using various programming techniques, e.g. the instantiation of a singleton instance (lazy or eager implementation). It is possible for a design pattern to have only one logical fragment (itself).

The identification of logical fragments of a design pattern as well as the transformations of these fragments are a purely manual process because they require understanding and abstraction of the semantics of a design pattern.

The application of the specific query transformations is an automated process, once we have query logical fragments and their transformations. Let us assume that we identified $N$ logical fragments $F_i : 1 \leq i \leq N$ and for each fragment $F_i$ we found a set of query transformations $T_{F_i} = T_j^i : 1 \leq j \leq K_i$. Then our algorithm to apply specific pattern-preserving query transformations is as follows:

1) We produce a Cartesian product of transformations over logical fragments, eliminating impossible combinations. As the result we obtain a set of possible transformation tuples $T = (t_1, ..., t_N) : t_i \in T_{F_i} \cup NONE$.
2) We apply the transformations tuples to the query obtained in Step 1.

### D. Generic Query Transformations

*Generic pattern-preserving query transformations* are based on generic pattern-preserving code transformations. Figure 3 presents the generic pattern-preserving transformations as identified in [18]: (1) abstractness transformations, (2) invocation transformations, (3) inheritance transformations, (4) aggregation transformations, (5) method signature transformations, and (6) access transformations.

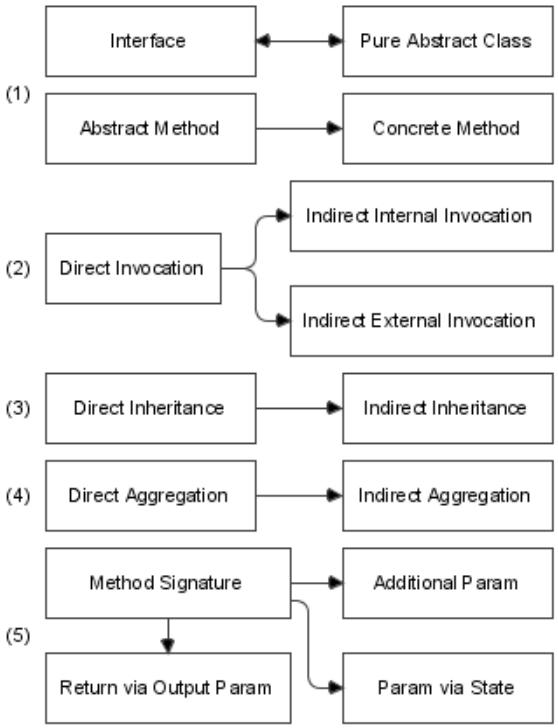


Fig. 3. The generic pattern-preserving code transformations.

*a) Abstractness Transformations:* These transformations refer to the property of being abstract for a class or a method. There is a two-sided transformation between interface and pure abstract class. This transformation is obvious since there exists a direct correspondence between these two constructs. They are often used interchangeably. The next transformation converts an abstract method to a concrete method. In our opinion this also is a natural transformation. It often happens that in the real world development programmers provide a default implementation instead of leaving a method abstract. By combining these transformations, we can transform an interface into a concrete class. Summing up, as long as we can take advantage of polymorphic calls, abstractness and concreteness do not impact the intent of a pattern code.

In terms of logic queries, the abstractness transformations lead to the following rewrite rules:

1) $isClass(Type) \rightarrow isClassOrInterface(Type)$
2) $isInterface(Type) \rightarrow isClassOrInterface(Type)$
3) $hasModifier(Class, 'abstract') \rightarrow true$
4) $hasModifier(Method, 'abstract') \rightarrow true$

*b) Inheritance Transformations:* The inheritance transformation introduces an intermediary level of inheritance, i.e. a direct subclassing is transformed into indirect. In real world software code such a construct can be the effect of a particular functional requirement or the complexity of a design problem. The length of an inheritance chain does not impact the intent of a pattern.

In terms of logic queries, the inheritance transformations lead to the following rewrite rule:

1) $isSubtypeOf(Type1, Type2) \rightarrow isSubtypeOf^*(Type1, Type2)$

*c) Invocation Transformations:* This group refers to transformations of invocation and instantiation statements. It is a popular refactoring. When a piece of code becomes complex, it is extracted into a separate method or class. This means that an invocation (or an instantiation), which remained direct until refactoring, is transformed into indirect. Depending on whether a new method or a new class is introduced, we call this indirect invocation internal or external respectively. Similarly to the length of an inheritance chain, the length of an invocation chain does not influence the logic of a pattern variant.

In terms of logic queries, the invocation transformations lead to the following rewrite rules:

1) $invokes(Method1, Method2) \wedge isMethodOf(Method1, Class) \wedge isMethodOf(Method2, Class) \rightarrow invokes^*(Type1, Type2) \wedge isMethodOf(Method1, Class)$
2) $invokes(Method1, Method2) \rightarrow invokes^*(Type1, Type2)$

*d) Aggregation Transformations:* These transformations follow the reasoning presented for invocations. Replacing a direct aggregation of an attribute (or a group of attributes) with an indirect aggregation does not influence the overall intent of a pattern implementation. Here is the example of such a transformation applied to the `observers` attribute:

```
class Observable {
    List<Observer> observers;
    ...
}
```

```
class Observable {
    ObserverList observerList;
    ...
}
class ObserverList {
    List<Observer> observers;
    ...
}
```

In terms of logic queries, the aggregation transformations lead to the following rewrite rule:

1) $isFieldOf(Field, Type) \rightarrow isFieldOfField^*(Field, Type)$

*e) Method Signature Transformations:* Here we present the transformations of method signatures. We have identified three such transformations:

- addition of a new (input) parameter,
- replacing the return value with an output parameter,
- passing an input value to a method via an object state instead of passing a parameter.

In terms of logic queries, the method signature transformations lead to the following rewrite rules:

1) $numberOfParameters(Method, N) \rightarrow true$
2) $isParameter(Param, Type) \wedge isParameterOf(Param, Method) \rightarrow hasInput(Method, Type)$
3) $hasReturnType(Method, Type) \rightarrow hasOutput(Method, Type)$

*f) Access Transformations:* These transformations relate to access modifiers of fields, methods, and classes. Basically, it means that access modifiers are not core elements of patterns and can be ignored during pattern detection. First of all, access modifiers are language-dependent feature. There are programming languages (e.g., JavaScript) that do not support access modifiers. Also, the semantics of the access modifiers may differ from one language to another (e.g., protected access in Java and C++). Therefore, we cannot rely on access modifiers and their semantics in a language-independent detection method. Moreover, there might be software requirements that limit the visibility of a field, method, or class (e.g., a Singleton class that is visible and accessible only in a package).

In terms of logic queries, the access transformations lead to the following rewrite rule:

1) $hasModifier(Element, Access) \rightarrow true$

## III. Examples

This section describes the application of our method of query relaxation to the Singleton pattern.

The Singleton pattern is the most popular pattern detected by the existing detection approaches. Its canonical implementation is simple, and the intent seems straightforward. However, by carefully analyzing the structure of this pattern, we can identify some corner cases among its implementation variants as well as in the usage context. The standard variant of the Singleton is shown in Figure 4.

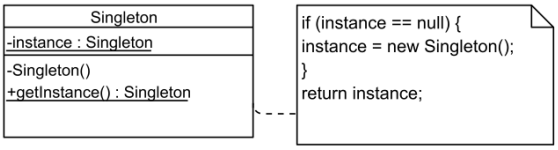


Fig. 4. The standard variant of the Singleton pattern.

### A. Step 1: UML to Logic Query Transformation

Below we present the query that reflects the canonical implementation of the Singleton. As such, it can be used to detect orthodox implementations of this design pattern.

A singleton class:
$$isClass(S) \wedge hasModifier(S, 'public')$$
A singleton constructor:
$$isConstructor(SCtr) \wedge isConstructorOf(S) \wedge numberOfParameters(SCtr, 0) \wedge hasModifier(SCtr, 'private')$$
A singleton getter (access point):
$$isMethod(Get) \wedge isMethodOf(Get, S) \wedge numberOfParameters(Get, 0) \wedge hasReturnType(Get, S) \wedge hasModifier(Get, 'static') \wedge hasModifier(Get, 'public')$$
A singleton field:
$$isField(Field) \wedge isFieldOf(Field, S) \wedge isTypeOf(Field, S) \wedge hasModifier(Field, 'static') \wedge hasModifier(Field, 'private')$$
An approximation of a lazy instantiation block:
$$isCodeBlock(Code) \wedge isCodeBlockOf(Get) \wedge instantiatesOptionally(Code, Instance) \wedge isTypeOf(Instance, S) \wedge writesOptionally(Code, Field, Instance)$$

### B. Step 2: Specific Query Transformations

In [18], we presented the logical fragments of the Singleton pattern along with the Singleton specific pattern-preserving code transformations. Here, we present the corresponding Singleton-preserving query transformations:

1) Instantiation: To Eager (A) – the lazy instantiation is replaced with an eager instantiation. The rewrite rule is as follows:
$$isCodeBlock(Code) \wedge isCodeBlockOf(Get) \wedge instantiatesOptionally(Code, Instance) \wedge isTypeOf(Instance, S) \wedge writesOptionally(Code, Field, Instance) \rightarrow isInitBlock(Code) \wedge instantiates(Code, Instance) \wedge isTypeOf(Instance, S) \wedge writes(Code, Field, Instance)$$
2) Placeholder: Inner Class (B) – the singleton instance is held as a static attribute of an inner class. The rewrite rule is as follows:
$$isFieldOf(Field, S) \rightarrow isFieldOf(Field, Inner) \wedge isClass(Inner) \wedge isMemberOf(Inner, S)$$

3) Placeholder: External Class (C) – the singleton instance is held as a static attribute of a class from within the same package. The rewrite rule is as follows:
$isFieldOf(Field, S) \rightarrow isFieldOf(Field, Outer) \wedge isClass(Outer) \wedge inPackage(S, P) \wedge inPackage(Outer, P)$
4) Access Point: Inner Class (D) – the public static method is moved to an inner class of a singleton. The rewrite rule is as follows:
$isMethodOf(Get, S) \rightarrow isMethodOf(Get, Inner) \wedge isClass(Inner) \wedge isMemberOf(Inner, S)$
5) Access Point: External Class (E) – the public static method is added to a newly created class in the same package. The rewrite rule is as follows:
$isMethodOf(Get, S) \rightarrow isMethodOf(Get, Outer) \wedge isClass(Outer) \wedge inPackage(S, P) \wedge inPackage(Outer, P)$
6) Access Point: Attribute (F) – the static singleton instance is made public (eagerly instantiated, with the access method removed). The rewrite rule is as follows:
$isMethod(Get) \wedge isMethodOf(Get, S) \wedge numberOfParameters(Get, 0) \wedge hasReturnType(Get, S) \wedge hasModifier(Get, 'static') \wedge hasModifier(Get, 'public') \rightarrow hasModifier(Field, 'public')$
7) Access Point: Protected (G) – the visibility of the access method is changed to protected. The rewrite rule is as follows:
$hasModifier(Get, 'public') \rightarrow hasModifier(Get, 'protected')$
8) Finality: Abstractness with Subclassing (H) – the Singleton class is made abstract and a concrete subclass is provided (the visibility of the Singleton constructor is changed to protected). The rewrite rule is as follows:
$isTypeOf(Instance, S) \rightarrow isTypeOf(Instance, CS) \wedge isClass(CS) \wedge isSubtypeOf(CS, S) \wedge hasModifier(S, 'abstract')$

### C. Step 3: Generic Query Transformations

As in this paper it is infeasible to list all generated queries, we present the resulting query obtained after the application of the generic query transformations to the query being the result of the application of the specific query transformations $(A, NONE, NONE, NONE)$ (i.e., lazy instantiation changed to eager instantiation):

A singleton class:
$isClass(S)$
A singleton constructor:
$isConstructor(SCtr) \wedge isConstructorOf(S)$
A singleton getter (access point):
$isMethod(Get) \wedge isMethodOf(Get, S) \wedge hasOutput(Get, S) \wedge hasModifier(Get, 'static')$
A singleton field:
$isField(Field) \wedge isFieldOf(Field, S) \wedge isTypeOf(Field, S) \wedge hasModifier(Field, 'static')$
An approximation of an eager instantiation block:
$isInitBlock(Code) \wedge instantiates(Code, Instance) \wedge isTypeOf(Instance, S) \wedge writes(Code, Field, Instance)$

## IV. Evaluation

We compared our prototype tool D-CUBED [20] with two state-of-the-art pattern detection tools: PINOT [21] and FUJABA 4.3.1 [2].

PINOT is a command-line tool written in C++ and based on jikes (the IBM Java compiler). PINOT is available as open source for custom compilation. PINOT ran smoothly, offering high performance in pattern detection tasks. The detection algorithms are hard-coded in PINOT, thus it is hard to experiment by modifying the detection approach. PINOT produces a useful, verbose report summarizing detected pattern instances.

FUJABA is a visually appealing graphic tool suite that provides pattern inference facilities as a plug-in (Inference Engine). There was no problem in launching FUJABA. It provides a UML-like language for user-defined patterns, and presents detected pattern instances as oval annotations on class diagrams. Even though this visual presentation helps in better understanding of diagrams, a summary report might be useful as well.

Similarly to PINOT, D-CUBED is a command line tool written in Java. However, contrary to PINOT, its detection queries are not hard-coded. Instead, we use XSB Prolog, a deductive database, as the data store for our program metamodel and Prolog as the query language. To generate a program metamodel from source code, we use Recoder (a front-end Java compiler) and a set of custom analyses to inspect in detail the call-flow and data-flow of a program.

We have tested these three tools against the source code of JHotDraw60b1 [22]. JHotDraw is a Java GUI framework for technical and structured graphics. It has originally been developed as a design exercise by Erich Gamma and Thomas Eggenschwiler.

Table I presents the results of the tests against JHotDraw. Unfortunately, FUJABA threw an exception during its static analysis. PINOT and D-CUBED performed their detection without any problems, however they produced significantly different results. PINOT did not report any Singleton instances, whereas D-CUBED recognized seven Singleton candidates:

1) *Clipboard* — a true positive; a singleton documented in source code.
2) *DisposableResourceManager* — a true positive; a singleton documented in source code; different placeholder;
3) *ResourceDisposabilityStrategy* — a true positive; analogous to *DisposableResourceManager*; different placeholder;
4) *CollectionsFactory* — a true positive; a singleton with subclassing and delegated construction;
5) *Alignment* — a true positive; a singleton with 6 instances;
6) *FigureEnumerator* — a false positive; there is a single static enumerator representing an empty enumerator,

though other instances are created as well;

7) *HandleEnumerator* — a false positive; there is a single static enumerator representing an empty enumerator, though other instances are created as well;

PINOT did not detect any singleton instance because its detection algorithm relies on the presence of the standard structure and a lazy instantiation block. *Clipboard* uses eager initialization (variant A), whereas two next singletons represent a different placeholder variant (variant C). *CollectionsFactory* represents a singleton with subclassing (variant H). As we did not impose an exactly one instance constraint, we also found the variant with several instances available (*Alignment*).

Unfortunately, query relaxation lead to two false positives. Therefore, the current method turned out to be too flexible. Our current goal, i.e. improving recall, decreased precision of the method. To achieve high precision, we need to filter out false positives. For a design pattern detection method, this minimization of false positives translates into understanding code constructs that violate the principles of a design patterns, even though the overall structure of a given piece of code resembles that of the design pattern. This is part of our undergoing research.

TABLE I
THE RESULTS OF SINGLETON DETECTION ON JHOTDRAW

| | PINOT | FUJABA | D-CUBED |
|---|---|---|---|
| Singleton | 0 | × | 7 (5 true positives) |

× the tool raised an exception

## V. RELATED WORK

There exists a number of proposed approaches to design pattern recognition, yet these approaches often lack in terms of accuracy, flexibility, or performance. A large number of approaches uses only structural information in order to detect design pattern instances (e.g. [23], [24]), but there also exist several approaches that exploit behavioural information contained in source code (e.g. FUJABA [2]–[4], Hedgehog [5], PINOT [6], [7]). We compare the existing detection methods to ours in terms of the concept of an approach, the architecture of a solution, and the mechanisms used to search for patterns.

A metamodel-based approach is not new. There exist several approaches that make use of a metamodel. Ptidej [10] is based on the PADL metamodel (the ancestor of PDL). However, a pattern in PADL is defined as a list of the required entities (a simple conjunction). Thus, it is hard to express more complex logic conditions. Moreover, it does not provide any support to data flow. SPQR [9] uses denotational semantics known as the $\rho$-calculus together with the set of elemental design patterns that capture call flow information. Similarly to PADL, there is no support to data flow. Hedgehog [5] utilises a Prolog-like language (Spine) to construct a pattern definition. Again, Spine's support to data flow is limited, but additional rules can be introduced. Also MAISA uses a metamodel. Its metamodel is UML and with its help it defines the structural patterns.

Current detection approaches utilise significantly different techniques to identify the instances of design patterns in source code. The most popular technique is the use of a logic inference system. This idea has been applied in Pat [25], where each structural pattern has been associated with a separated set of rules and Prolog interpreter has been used to search for patterns. Also [8] utilises a logic inference system to detect patterns in Java and Smalltalk based on a language-specific naming and coding conventions. SPQR and FUJABA also employ a logic inference engine to reason about the pattern instances.

Ptidej [10] uses a constraints solver with automatic constraint relaxation to detect sets of entities similar to a design pattern. A related work of Ptidej [26] utilises program metrics and a machine learning algorithm to fingerprint design motifs roles. One more approach that uses machine learning techniques is [27]. It enhances a pattern-matching system [11], [12] by filtering out false positives.

Numerous approaches simply navigate over a program abstract syntax tree to find the instances of patterns. They perform static or dynamic analyses to capture the behaviour of a program. Usually their detection algorithms are hard coded and tailored to a particular programming language. PINOT [6] performs static analysis to identify pattern-specific code-blocks. It is a lightweight solution performing recognition in an efficient manner. The approach described in [7] has two phases. In the first phase (the static analysis of the code structure) the abstract syntax tree is analysed in order to select the set of candidates to be pattern instances. In the second phase (the dynamic analysis of a program run) the messages passed are examined to check whether a candidate instance from the first phase is rejected or accepted.

Our approach and the prototype D-CUBED utilises structural information, but it extends the structure-driven approaches by targeting behaviour of the patterns. D-CUBED seems similar to the database driven approaches like DP++ [28] or SPOOL [29], but while these approaches address only the structural patterns, D-CUBED addresses the creational and behavioural patterns utilising the elemental relations to capture code intent.

A completely different approach is taken by the authors of DPJF [30]. They run a number of detection tools and the fuse their results. Such an approach may boost both precision and recall depending on the tuning of parameters. In this paper we focus on our specific method that can improve DPJF when their maintainers upload new version of D-CUBED.

## VI. CONCLUSION

In this paper we proposed a general method to generate queries (logic programs) that detect disparate implementation variants of design patterns. First, we produce a strict query that reflect only the canonical textbook version of a design pattern. Then, we relax this query in order to allow multiple variants.

We implemented this approach in our prototype tool D-CUBED. We experimentally verified it with respect to state-of-

the-art detectors. The results are promising, since D-CUBED has detected several non-trivial variants of the Singleton that have not been revealed by other tools.

Apparently, our approach increases the recall of the detection process. However, higher recall may possibly imply lower precision. As the next step in our research, we plan to limit the number of false positives, i.e. detected instances that are not real incarnations of the design patterns. We will attempt tightening the detection queries by *adding* conditions that detect features that actually *violate* the intent of the given design pattern.

## REFERENCES


[1] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, *Design Patterns*. Addison-Wesley, 1994.

[2] J. Niere and L. Wendehals, "An interactive and scalable approach to design pattern recovery," Tech. Rep., 2003.

[3] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002, pp. 338–348.

[4] J. Niere, J. P. Wadsack, and L. Wendehals, "Handling large search space in pattern-based reverse engineering," in *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2003, p. 274.

[5] A. Blewitt, A. Bundy, and I. Stark, "Automatic verification of design patterns in java," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 224–232.

[6] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 123–134.

[7] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic design pattern detection," in *IWPC*. IEEE Computer Society, 2003, pp. 94–104.

[8] J. Fabry and T. Mens, "Language independent detection of object-oriented design patterns," *Computer Languages, Systems and Structures*, vol. 30, no. 1–2, pp. 21–33, 2004.

[9] J. Smith and D. Stotts, "Formalized design pattern detection and software achitecture analysis," Dept. of Computer Science, University of North Carolina, Tech. Rep. TR05-012, 2005.

[10] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien, "Instantiating and detecting design patterns: Putting bits and pieces together," in *ASE*. IEEE Computer Society, 2001, pp. 166–173.

[11] Z. Balanyi and R. Ferenc, "Mining design patterns from C++ source code," in *ICSM '03: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2003, p. 305.

[12] R. Ferenc, J. Gustafsson, L. Müller, and J. Paakki, "Recognizing design patterns in C++ programs with integration of columbus and maisa," *Acta Cybern.*, vol. 15, no. 4, pp. 669–682, 2002.

[13] K. Stencel and P. Węgrzynowicz, "Detection of diverse design pattern variants," in *APSEC '08: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 25–32.

[14] N. Pettersson, W. Löwe, and J. Nivre, "On evaluation of accuracy in pattern detection," in *First International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE'06)*, October 2006. [Online]. Available: http://cs.msi.vxu.se/papers/PLN2006a.pdf

[15] L. J. Fulop, R. Ferenc, and T. Gyimothy, "Towards a benchmark for evaluating design pattern miner tools," *Software Maintenance and Reengineering, European Conference on*, vol. 0, pp. 143–152, 2008.

[16] K. Stencel and P. Wegrzynowicz, "Implementation variants of the singleton design pattern," in *OTM Workshops*, ser. Lecture Notes in Computer Science, R. Meersman, Z. Tari, and P. Herrero, Eds., vol. 5333. Springer, 2008, pp. 396–406.

[17] K. Stencel and P. Węgrzynowicz, "Visitor pattern revisited for recognition," in *ADBIS (local proceedings)*, P. Atzeni, A. Caplinskas, and H. Jaakkola, Eds. Tampere University of Technology. Pori. Publication, 2008, pp. 154–166.

[18] P. Węgrzynowicz and K. Stencel, "Towards a comprehensive test suite for detectors of design patterns," in *ASE*. IEEE Computer Society, 2009, pp. 103–110.

[19] A. Horn, "On sentences which are true of direct unions of algebras," *J. Symb. Log.*, vol. 16, no. 1, pp. 14–21, 1951.

[20] P. Węgrzynowicz and K. Stencel, "The good, the bad, and the ugly: three ways to use a semantic code query system," in *OOPSLA Companion*, S. Arora and G. T. Leavens, Eds. ACM, 2009, pp. 821–822.

[21] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *ASE*. IEEE Computer Society, 2006, pp. 123–134.

[22] E. Gamma and T. Eggenschwiler, "JHotDraw," http://www.jhotdraw.org/, 1996–2008.

[23] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Software Eng.*, vol. 32, no. 11, pp. 896–909, 2006.

[24] K. Brown, "Design reverse-engineering and automated design pattern detection in Smalltalk," Master's thesis, University of Illinois at Urbana Campaign, 1997.

[25] L. Prechelt and C. Krämer, "Functionality versus practicality: Employing existing tools for recovering structural design patterns," *J. UCS*, vol. 4, no. 11, pp. 866–882, 1998.

[26] Y.-G. Gueheneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," in *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 172–181.

[27] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 295–304.

[28] J. Bansiya, "Automating design-pattern identification," *Dr. Dobbs Journal*, 1998.

[29] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé, "Pattern-based reverse-engineering of design components," in *ICSE '99: Proceedings of the 21st international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 226–235. [Online]. Available: http://portal.acm.org/citation.cfm?id=302622

[30] A. Binun and G. Kniesel, "DPJF - design pattern detection with high accuracy," in *CSMR*, T. Mens, A. Cleve, and R. Ferenc, Eds. IEEE, 2012, pp. 245–254.