Detection of Diverse Design Pattern Variants

Conference Paper · January 2008

DOI: 10.1109/APSEC.2008.67 · Source: DBLP

2 authors:

Krzysztof J. Stencel
University of Warsaw
105 PUBLICATIONS **586** CITATIONS

SEE PROFILE

Patrycja Wegrzynowicz
University of Warsaw
8 PUBLICATIONS **93** CITATIONS

SEE PROFILE

2008 15th Asia-Pacific Software Engineering Conference

Detection of Diverse Design Pattern Variants

Krzysztof Stencel
Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warsaw, Poland
stencel@mimuw.edu.pl

Patrycja Węgrzynowicz
NASK Research and Academic Computer Network
Wąwozowa 18, 02-796 Warsaw, Poland
patrycjaw@nask.pl

Abstract

We propose a method for automatic detection of occurrences of design patterns. We also describe its proof-of-concept implementation and the results of comparative experiments with other tools. The method presented here is able to detect many nonstandard implementation variants of design patterns, while its efficiency is comparable to other state-of-the-art detection tools. Moreover, the method is highly customizable because an analyst can introduce a new pattern retrieval query or modify an existing one and then repeat the detection using the results of earlier source code analysis stored in a relational database.

1. Introduction

Design patterns [11] facilitate creation of quality designs. However, as well as being useful during the construction of software systems, they also aid in analyzing existing systems, e.g., reconstructing the documentation of a legacy system from its source code (*reverse engineering*). The automation of reverse engineering can be more efficient if design patterns are recognized. If they are properly detected, the analysis can reflect the design intentions more accurately. If these intentions are caught in the reconstructed documentation, the future maintenance and development of a legacy system are much easier and less costly.

The recognition of instances of design patterns in code is difficult, because the patterns are not formally defined. The only formal thing we have is the *canonical form* of each pattern. Instances of design patterns can diverge from the canonical form because of specific properties of the chosen programming language and additional design requirements implied by the nature of the problem being solved. Furthermore, design patterns are often independently invented and developed by programmers. Although the ideas behind these inventions are similar, details of their implementations can differ. Instances of design patterns are also often tangled together—a particular system function is usually implemented by the cooperation of a number of patterns.

In this paper we propose a scalable and flexible method of automatic pattern recognition. We also describe a proof-of-concept implementation and the results of experiments comparing our approach with other pattern recognition approaches.

Contributions of this paper .

- We present a pattern recognition method that is able to detect many nonstandard implementations of design patterns as well as standard implementations.
- We present its proof-of-concept implementation.
- We show experimental data that prove that this implementation is among the fastest pattern recognition tools. Although it is not the fastest, it has the following two advantages:
- First, it detects many nonstandard implementations of design patterns that are not detected by faster methods.
- Second, the running time of our method consists of program analysis, loading the result into a relational database and then running design pattern retrievals using SQL queries. If the results of the analysis are not satisfactory, the analyst can refine the SQL queries (to relax or tighten a pattern's characteristics) and repeat the analysis.
- The presented method is thus customizable because it allows efficient repetitive analyses using refined pattern definitions.

2. Motivating Examples

In recent years, we have observed a continuous improvement in the field of design pattern recognition. Current approaches can detect a fairly broad range of design patterns, targeting structural as well as behavioral aspects of patterns. However, these approaches are not perfect and sometimes

1530-1362/08 $25.00 © 2008 IEEE
DOI 10.1109/APSEC.2008.67

fail to capture source code intent. Moreover, there are patterns (like the Builder) that still have not been investigated in-depth.

2.1. Factory Method Pattern

The Factory Method pattern is one of the most popular patterns detected by the existing detection approaches. Its canonical implementation is simple and its intent is straightforward. However, even in such a relatively simple case, we can identify some corner cases among its implementation variants as well as in a usage context.

Firstly, we must realize that the canonical variant with an abstract factory method imposes a strong constraint on code. In practice, a factory method often is not abstract, but returns a default instance. Pattern3 [23], an interesting approach to pattern detection based on graph similarity scoring, does not cover this variant and produces a false negative for the code shown in Figure 1. FUJABA [19], a state-of-the-art tool, also fails to discover it.

```
interface Product {...}
class ProductA implements Product {...}
class ProductB implements Product {...}
class Creator {
 // the factory method
 Product createProduct() {
  // create a default product instance
  return new ProductA();
 }
}
class ConcreteCreator extends Creator {
  // the overriden factory method
  Product createProduct() {
    return new ProductB();
  }
}
```

Figure 1. The createProduct method is a factory method but FUJABA and pattern3 do not detect it.

Secondly, it should be taken into account that the most popular implementation variant, a no-arguments factory method, is not the only one available. Another interesting variant involves a parameter passed to a factory method to parameterize the construction of a product (see Figure 2). PINOT [20] is a very efficient solution for pattern detection based on an interesting concept of static code analysis used in a search for specific code blocks. It goes a step further than FUJABA and Pattern3 and addresses the behavior of the Factory Method. However, PINOT produces a false negative result with the parameterized factory method.

```
interface Product {...}
class ProductA implements Product {...}
class ProductB implements Product {...}
class Creator {
 // the parameterized factory method
 Product createProduct(int type) {
  return new ProductA(type);
 }
}
class ConcreteCreator extends Creator {
 Product createProduct(int type) {
  if (type == A)
   return new ProductA();
  else if (type == B)
   return new ProductB();
  else
   return super.createProduct(type);
 }
}
```

Figure 2. The createProduct method is a parameterized factory method that is not recognized by PINOT, FUJABA, or pattern3.

Thirdly, design patterns are often combined to model complex requirements. Figure 3 shows a factory method that uses a singleton class to produce the product instances. Neither PINOT, FUJABA, nor Pattern3 detect this variant of the Factory Method.

3. Approach to Pattern Detection

Our aim was to capture the intents of created patterns and provide a way to discover as many of their implementation variants as possible. We have focused on the analysis of different implementations of the patterns (as in Section 2 for the Factory Method pattern) to find the essence of patterns.

Firstly, we established a simple metamodel of a program. The program metamodel consists of a set of core elements and a set of relations among these elements. The core elements represent basic object-oriented constructs such as types, methods, or instances, whereas the identified relations describe the structural and behavioral features of a program. The relations model such program characteristics as inheritance trees, call graphs, and sets of input and output values together with possible assignments to variables. The metamodel helps to avoid a strong connection to a particular programming language.

Then, we analyzed different implementation variants of the created patterns. This resulted in the first-order logic formulae that define four of the five GoF creational patterns:

```
interface Product {...}
class ProductA implements Product {...}
class Singleton implements Product {
 private static Singleton instance =
  new Singleton();
 private Singleton() {}
 public static Singleton getInstance() {
  return instance;
 }
}
class Creator {
 abstract Product createProduct();
}
class ConcreteCreator extends Creator {
 Product createProduct() {
  return Singleton.getInstance();
 }
}
```

Figure 3. A combination of a factory method and a singleton, not reported by PINOT, FUJABA, or pattern3.

Singleton, *Factory Method*, *Abstract Factory*, and *Builder*. The first-order logic definitions of the patterns abstract away a programming language or a particular analysis technique. They allow us to implement the detection method in various ways.

Finally, we developed a proof-of-concept of our methodical approach and we compare its accuracy and performance with the state-of-the-art detection tools. The prototype system, the D^3 (*D-cubed*)tool, discovers a program metamodel using the Recoder library and static analysis. D^3 stores the discovered relations in a database and transitively closes some of them. Because first-order logic queries can be implemented in SQL, the logic formulae describing the patterns have been manually translated to SQL queries and these queries are used to search for pattern instances.

4. Program Metamodel

The proposed program metamodel consists of a set of core elements and a set of relations among those elements. 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 proposed program metamodel consists of the following core elements: types, attributes-or-variables (further referred as attributes), operations, and instances. Most of them have their intuitive object-oriented meaning with one exception—the instance. An instance has been defined as an equivalence class of the relation "objects constructed by the same `new`[1] statement," i.e., all objects instantiated by the same `new` statement are treated as a single instance.

The identified elemental relations have been organized into two categories: structural relations and behavioral relations.

4.1. Structural Relations

The elemental structural relations describe a static type hierarchy of a program, including an override relation implied by the hierarchy.

istype
: $istype(A, B)$, if and only if A is a subtype of B or $A = B$. *Istype* is a transitive relation.

override
: $override(F, G)$, if and only if the operation F overrides the operation G. *Override* is a transitive relation. This relation only states an override and does not provide any rules of overriding because those vary from one programming language to another.

4.2. Behavioural Relations

The behavioral relations mostly refer to a data flow and a call flow including instantiation as a specific call to a `new` operator.

invocation
: $invocation(F, G)$, if and only if the operation F directly or indirectly invokes the operation G. This relation also captures polymorphic calls resulted from dynamic binding. *Invocation* is a transitive relation.

instantiation
: $instantiates(F, I)$, if and only if the operation F directly or indirectly instantiates the instance I. *Instantiation* is a transitive relation. In most cases, *instantiation* means that the operation F invokes a method G and there is a `new` statement in the body of G that instantiates I. However, *instantiation* also covers a case when a `new` call is a side-effect of a method call (e.g., a method call may result in the class being loaded and initialized). Again, we do not specify any language-dependent conditions for this relation but leave them to an implementor.

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

[1] Or its language-dependent equivalent.

output

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

instance2static

instance2static(I, A), if and only if there is a potential execution path where the instance I is being assigned to the static (or global) attribute A.

5. Pattern Definitions

5.1. Singleton

The intent of the Singleton pattern is to ensure that a class has only one instance. It implies that a Singleton instance should be reusable. The only way to keep a Singleton instance for future reuse is to store it in a static (or global) context. Existence of a static (or global) variable of a Singleton type is only a necessary condition. To identify a true Singleton, we must make sure there is no improper usage of a class, i.e., a new instance of a Singleton is instantiated only to initialize a static (or global) Singleton variable. As it might be hard to verify whether an instance is assigned to a static variable for each execution path, we propose to only verify whether such an execution path exists.

More formally, a class C is a candidate Singleton instance if and only if:

- There exists exactly one static attribute A of the type C; and
- For each instance of the type C, there is an execution path where the instance is assigned to a static (or global) attribute.

The formula below expresses the above definition using the relations described in Section 4.

$$\underbrace{\begin{array}{rl} \exists!\text{attr.}\, A: & A.staticOrGlobal = true \wedge \\ & A.type = C \wedge \\ \exists\text{inst.}\, I: & istype(I.type, C) \wedge \\ \forall\text{inst.}\, J: & istype(J.type, C) \Rightarrow \\ & \exists\text{attr.}\, B: instance2static(J, B) \end{array}}$$

$$\Downarrow$$

The class C is a candidate `Singleton`.

We also considered a design concept similar to the Singleton, in which the intent is to ensure that a class has a limited number of instances. We call it the *Limiton*. [11] mentions that one of the advantages of the Singleton is the possibility of accommodating a policy of a limited number of instances. The similarity between the Singleton and the Limiton is well illustrated by the object-oriented model of two solar systems: a single-star system (the example of the classic Singleton) and a binary-star system (the example of the Limiton). These two analogous implementations may be perceived as two variants of the same design concept. To detect Limiton candidates, we simply replace the "exists exactly one" quantifier in our formula with the standard "exists".

The above definition does not impose any structural constraint on a program except the existence of a static (or global) attribute of a Singleton class. Contrary to other solutions, we do not require the static attribute to be present in a Singleton class itself, nor do we force a static access method to be implemented, or depend on access modifiers.

5.2. Factory Method

A factory method (*createProduct*) is a key player in the Factory Method pattern. Our definition of this pattern states that a *Creator* class must contain a single *createProduct* method that is overridden in a subclass and the overridden method creates and returns instances of a *Product*. The formula below expresses the formal definition of the Factory Method candidates.

$$\underbrace{\begin{array}{rl} \exists\text{type}\, D: & C \neq D \wedge \\ \exists\text{oper.}\, F: & F \in C.operations \wedge \\ \exists\text{oper.}\, F_D: & F_D \in D.operations \wedge \\ & override(F_D, F) \wedge \\ \exists\text{inst.}\, I: & istype(I.type, F.returnType) \wedge \\ & instantiation(F_D, I) \wedge output(F_D, I) \end{array}}$$

$$\Downarrow$$

The class C is a candidate `Creator` of Factory Method.

This definition covers two major variations [11] of the Factory Method implementation: (1) when *createProduct* is an abstract method; and (2) when *createProduct* has a concrete implementation. Moreover, it allows for an unlimited depth of inheritance as well as indirect instantiation. Thus, the definition can catch combinations of the Factory Method with other patterns (like the Singleton or Builder) used to obtain a product instance. In addition, the formula does not impose any constraint on parameters of a factory method, nor on the number of different products produced by a single class. This allows us to catch a *parameterized factory method* variant (when an override exists).

5.3. Abstract Factory

The Abstract Factory pattern is similar to that for the Factory Method. The difference is that the Factory Method

is about one method while the Abstract Factory is about a set of methods that produce related products. Thus, our definition of the Abstract Factory is similar to that of the Factory Method. The difference is that for the Abstract Factory we require at least two *createProduct* methods to be present.

$$\begin{aligned}
\exists \text{class}\, D: \quad & C \neq D \wedge \\
\exists \text{oper}.\, F, G: \quad & F \neq G \wedge \\
& F \in C.operations \wedge G \in C.operations \wedge \\
\exists \text{oper}.\, F_D: \quad & F_D \in D.operations \wedge \\
& override(F_D, F) \wedge \\
\exists \text{oper}.\, G_D: \quad & G_D \in D.operations \wedge \\
& override(G_D, G) \wedge \\
\exists \text{inst}.\, I: \quad & istype(I.type, F.returnType) \wedge \\
& instantiation(F_D, I) \wedge output(F_D, I) \wedge \\
\exists \text{inst}.\, J: \quad & istype(J.type, G.returnType) \wedge \\
& instantiation(G_D, J) \wedge output(G_D, J)
\end{aligned}$$

$$\Downarrow$$

The class C is a candidate `Abstract Factory`

5.4. Builder

Two major parts of the Builder pattern are: (1) creation of parts, and (2) assembling parts into a product. Our definition reflects these two parts. Firstly, we require at least one build part method to be present (and its override) that instantiates parts. Secondly, we require a get result method that returns a product with the parts created by the build part method. More formally, we can express these conditions using the following formula:

$$\begin{aligned}
\exists \text{class}\, D: \quad & istype(D, B) \wedge D \neq B \wedge \\
\exists \text{oper}.\, B_P: \quad & B_P \in B.operations \wedge \\
\exists \text{oper}.\, B_P^D: \quad & B_P^D \in D.operations \wedge \\
& override(B_P^D, B_P) \wedge \\
\exists \text{inst}.\, I: \quad & instantiation(B_P^D, I) \wedge \\
\exists \text{oper}.\, G_P \exists \text{class}\, E: \quad & istype(D, E) \wedge istype(E, B) \wedge \\
& G_P \in E.operations \wedge indirect_output(G_P, I)
\end{aligned}$$

$$\Downarrow$$

The class B is a candidate `Builder`.

Our Builder definition does not introduce any constraint on the length of an inheritance chain or a call flow. We do not use any language-specific constructs like access modifiers, nor do we require methods to be abstract.

6. Implementation

Detection of Diverse Design Pattern Variants: D^3 (*D-cubed*) is our tool developed as a proof of concept for our methodology. It detects creational design pattern candidates in Java source code using static analysis techniques and SQL.

The detection process consists of parsing, analysis, and detection. In the parsing step we use the Recoder tool [15] to create an abstract syntax tree (AST) from Java sources and then we construct the core elements based on the AST. The analysis step involves performing a set of analyses (structural analysis, call flow analysis, and data flow analysis) to discover the elemental relations. During the analysis phase, transitive closures of relations are computed if necessary. Then, the core elements and relations discovered by the analyses are stored in a relational database. At the detection step, the SQL queries are executed to discover pattern instances. The queries follow the design pattern definitions expressed in first-order logic and presented in Section 5.

The set of static analyses used to discover the elemental relations consists of the structural analysis, the call flow analysis, and the data flow analysis.

Structural Analysis is responsible for discovery of the relations implied by the structure of the program, i.e., *istype* and *override* relations. First, the direct relations are discovered, and necessary transitive closures are then computed using DFS (Depth-First Search) algorithm.

Call Flow Analysis discovers the relations implied by the structure of invocations, i.e., *invocation* and *instantiation*. It is important that this analysis focuses on a type hierarchy and method bodies, but ignores the data flow. It means that the computed relations might include more elements than the actual number resulting from the program execution. This feature allows us to analyze a broader spectrum of potential program execution paths. Analogously to the structural analysis, first the direct relations are discovered and then transitive closures are computed where necessary.

Data Flow Analysis performs a simple static data flow analysis in a top-down and flow-sensitive way. It discovers the following relations: *input*, *output*, and *instance2static*. First, the operations are sorted using a topological sort. Then, an arbitrary number of iterations is performed to stabilize the input and output sets of each operation as much as possible, and values are set for each static attribute. In each iteration, the method bodies are interpreted using a simple Java interpreter developed specifically for this purpose. While interpreting, information about possible input, output, and static values are being collected.

Our choice of a database for the implementation was dictated by tests and research made during the early stages of the prototype implementation. As other options, we considered a logic inference engine or a code query system. We implemented a simple prototype in Prolog (the facts generated by the analyses). It showed poor performance (counted in hours) because of the need to compute transitive closures (when the transitive closures were computed by Prolog) or because of the large number of facts (when the transitive closures were generated beforehand). We also investigated code query systems, but these did not have the required

predicates and rules. Neither jquery [17] nor CodeQuest [13] support reasoning about data flow, which is crucial in our pattern definitions. Moreover, we believe that the use of a relational database and SQL opens a way to further optimization and even better performance.

7. Results

We compared D^3 with two state-of-the-art pattern detection tools: PINOT and FUJABA 4.3.1 (Inference Engine 2.1).

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. After minor problems with compilation (the result of changes in gcc) 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, but its further usage brought some stability and performance issues. FUJABA provides a UML-like language for user-defined patterns, and presents detected pattern instances as oval annotations on class diagrams. This visual presentation helps to better understand a diagram, but is not helpful when one wants to summarize results.

Similarly to PINOT, D^3 is a command line tool written in Java. However, contrary to PINOT, its detection queries are not hard-coded but are written in SQL and kept in a separate configuration file. Thus, it is possible to modify the existing queries as well as to add new ones. A text report produced by D^3 contains running times of subsequent phases as well as detected pattern instances.

We have tested these three tools against the demo source of "Applied Java Patterns" [22] (AJP) and JHotDraw60b1 [10]. The AJP demo source provides an exemplary Java implementation of GoF patterns. JHotDraw is a Java GUI framework for technical and structured graphics.

Tables 1 and 2 present the results of the tests against AJP and JHotDraw, respectively. PINOT and D^3 succeeded in detecting the AJP patterns, while FUJABA only managed to detect the Singleton successfully. With JHotDraw, FUJABA stopped inference after throwing a `StackOverflowException` while evaluating predicates during its static analysis. PINOT and D^3 performed detection without any problems. However, these two tools did not fully agree on the results.

There is a significant difference in detection of the Singleton pattern between PINOT and D^3. PINOT did not report any Singleton instances, whereas D^3 recognized four

Table 1. Results for pattern detection on AJP

	PINOT	FUJABA	D^3
Singleton	✓	✓	✓
Factory Method	✓	×	✓
Abstract Factory	✓	×	✓
Builder			✓
Prototype			

✓ the tool detects the pattern correctly
× the tool claims to detect the pattern, but fails in this case
the tool does not support detection of this pattern

Table 2. Results for pattern detection on JHotDraw

	PINOT	FUJABA	D^3
Singleton	0	×	4
Limiton			4
Factory Method	16	×	18
Abstract Factory	15	×	7
Builder			7
Prototype			

× the tool raised an exception
the tool does not support detection of this pattern

Singleton candidates and one Limiton candidate. After careful analysis, three candidates that were obvious Singleton instances were documented in the source code. The next candidate (`CollectionsFactory`) is also a Singleton instance, but is more complex because it is a combination of the Factory Method and Singleton. `Alignment` was reported as a Limiton instance, with six instances allowed.

The results for the Factory Method and Abstract Factory also differ between PINOT and D^3. Firstly, PINOT assumes that the Abstract Factory instances are a subset of the Factory Method instances, whereas D^3 reports distinct sets of instances for these two patterns. Because of this feature, some classes (e.g., `CreationTool`) were reported as Factory Method instances by PINOT, whereas D^3 classified them as Abstract Factory candidates. In addition, some differences arose because D^3 reports different pattern instances from the same inheritance tree. Thus, to make the comparison more informative, we evaluated the results of the Factory Method and Abstract Factory together, limited only to distinct inheritance trees. The main difference is because PINOT analyzes library classes, while D^3 analyzes only sources. PINOT reported five Factory Methods from the Java standard library (e.g., `List`, `Set`, and `Image`). Furthermore D^3 discovered correctly five instances of the Factory Method/Abstract Factory that were not recognized

by PINOT. This difference arose because of a slightly different static analysis and discovery of instantiation paths.

Finally, PINOT was not able to recognize the Builder pattern. The number of Builder instances detected by D^3 is not very high, but none of them is an obvious Builder instance. Three Builder candidates might be perceived as Builder. It shows that our approach is useful but can be further improved because our definition did not include a *Director* role. Modification of our Builder definition to cover a *Director* role should reduce the number of false positives.

Table 3 shows the running times for the parsing, analysis, and detection phases from the test performed against JHotDraw. The test was performed on the machine with a 2 Ghz Intel Centrino Duo processor with 2 GB RAM running Windows XP and MySQL 5.0.27.

Table 3. Detailed running times of D^3 on JHotDraw (45.6 KLOC)

Phase	Time (in seconds)
Parsing	1.29
Structural Analysis	4.01
Call Flow Analysis	2.34
Data Flow Analysis	6.40
Insertion into Database	17.67
Detection	4.13
Total	35.84

Compared with other tools, D^3 runs relatively fast. It is a little slower than PINOT but significantly faster than FUJABA. PINOT analyzes JHotDraw in $\tilde{7}$ s on the same machine, while it takes $\tilde{2}0$ min for FUJABA to analyze Java AWT 1.3. PINOT has its detection algorithms hard-coded; thus, it is impossible to modify them except by modifying the source code. D^3 uses simple SQL queries to detect patterns; thus, any user can create a custom pattern query. Additionally, PINOT must be run every time detection is to be performed, while our tool stores relations and elements in a database so one can perform a database initialization once and then re-use discovered elements and relations to detect various patterns. Moreover, PINOT relies on a specific block to be present in the source code, so its flexibility is lower and it is not able to detect some variants of the patterns that are successfully recognized by our method. For example, PINOT does not recognize the parameterized Factory Method, nor different variants of the Singleton.

8. Related Work

Despite a number of developed approaches to recognition of design patterns, there is still room for improvement, especially in the recognition of behavior-driven patterns, pattern variants, and performance. Many approaches use only structural information to detect design pattern instances, but there are also several approaches that exploit behavioral information contained in the source code.

Structure-driven approaches are mostly based on type hierarchies, association and aggregation relationships, method modifiers, method signatures, and method delegations. Although these approaches use similar types of information, they leverage various representations and search mechanisms.

Some approaches use language support and hard-coded algorithms (e.g., [5]), whereas others store information in a database (e.g., DP++ [3] or SPOOL [18]). A popular method is to use a logic inference system. [7] utilized a logic inference system, SOUL, to detect patterns in Java and Smalltalk based on language-specific naming and coding conventions. SPQR, described in [21], employs a logic inference engine, but to represent a program and pattern definitions, it uses the denotation semantics known as the ρ-calculus. Another structural approach is [23], in which the design pattern detection method is based on graph similarities.

In addition to structure-driven approaches, some methods targeting the behavior of patterns have been invented. These behavior-driven approaches employ various types of analysis, including static and dynamic program analysis, fuzzy sets, or machine-learning techniques, to capture the intent of code elements.

Hedgehog [4] and PINOT [20] both apply static analysis techniques to capture program intent. Hedgehog tries to identify some semantic facts that can later be used by a proof engine. PINOT performs hard-coded static analysis to identify pattern-specific code blocks. [14] and [16] utilize dynamic analysis to understand program behavior. [14] is an interesting approach that uses the concept of metapatterns. PTIDEJ [1] identifies distorted microarchitectures in object-oriented source code using explanation-based constraint programming. The related work on PTIDEJ [12] utilizes program metrics and a machine-learning algorithm to fingerprint design motifs and roles. One more approach that uses machine-learning techniques is [8]. It enhances a pattern-matching system [2, 9] by filtering out false positives.

Our approach and the prototype D^3 utilize behavioral information in addition to structural information. The structural model (the core elements and structural relations) is similar to the set of structural predicates presented in [6]. Compared with other approaches, our method shows high accuracy in detection of the Singleton, Factory Method, and Abstract Factory patterns in real source code. It detects variants of the patterns. Our prototype implementation also shows good performance, which is often a serious is-

sue of the methods based on logic programming. Moreover, the proposed method is flexible, allowing addition of new queries or modification of existing ones. In addition, SQL makes the tool more approachable to an average developer because most developers know SQL (not a case with logic programming).

9. Conclusions and Future Work

We have presented an efficient and flexible method for detecting design patterns. We also described a prototype implementation, D^3. We have presented experimental data that prove that the described method is as efficient as other pattern detection approaches. Furthermore, we are convinced that our method could be more useful than the others because it is able to detect nonstandard implementations of design patterns. Last but not least, the method is flexible because the definitions of detected design patterns are stored outside the tool and can be easily modified to improve design pattern retrieval and obtain more suitable results.

Future work encompasses extending the method to all other design patterns. We already have promising but still preliminary results for behavioral patterns. The translation of first-order logic formulae characterizing design patterns as SQL queries is very laborious and error-prone. An automatic translator (existing or developed from scratch) for this purpose will be very useful. One could also consider pattern detection for languages without static type systems (e.g., Python), where a kind of dynamic analysis or sampling of program execution could be used.

References

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

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

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

[4] 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*, pages 224–232, New York, NY, USA, 2005. ACM.

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

[6] J. Dong, T. Peng, and Z. Qiu. Commutability of design pattern instantiation and integration. In *TASE*, pages 283–292. IEEE Computer Society, 2007.

[7] J. Fabry and T. Mens. Language independent detection of object-oriented design patterns. *Computer Languages, Systems and Structures*, 30(1–2):21–33, 2004.

[8] 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*, pages 295–304, Washington, DC, USA, 2005. IEEE Computer Society.

[9] R. Ferenc, J. Gustafsson, L. Müller, and J. Paakki. Recognizing design patterns in C++ programs with integration of columbus and maisa. *Acta Cybern.*, 15(4):669–682, 2002.

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

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

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

[13] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In D. Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[14] S. Hayashi, J. Katada, R. Sakamoto, T. Kobayashi, and M. Saeki. Design pattern detection by using meta patterns. *IEICE Transactions*, 91-D(4):933–944, 2008.

[15] D. Heuzeroth, U. Aßmann, M. Trifu, and V. Kuttruff. The COMPOST, COMPASS, Inject/J and RECODER tool suite for invasive software composition: Invasive composition with COMPASS aspect-oriented connectors. In R. Lämmel, J. Saraiva, and J. Visser, editors, *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 357–377. Springer, 2006.

[16] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. In *IWPC*, pages 94–104. IEEE Computer Society, 2003.

[17] Jquery. http://www.cs.ubc.ca/labs/spl/projects/jquery/.

[18] 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*, pages 226–235, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[19] J. Niere and L. Wendehals. An interactive and scalable approach to design pattern recovery. Technical report, 2003.

[20] 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*, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[21] J. Smith and D. Stotts. Formalized design pattern detection and software achitecture analysis. Technical Report TR05-012, Dept. of Computer Science, University of North Carolina, 2005.

[22] S. A. Stelting and O. M.-V. Leeuwen. *Applied Java Patterns*. Prentice Hall Professional Technical Reference, 2001.

[23] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Software Eng.*, 32(11):896–909, 2006.