Towards a Comprehensive Test Suite for Detectors of Design Patterns

Conference Paper · November 2009

DOI: 10.1109/ASE.2009.85 · Source: DBLP

CITATIONS
4

READS
99

2 authors:

Patrycja Wegrzynowicz
University of Warsaw
8 PUBLICATIONS **60** CITATIONS
SEE PROFILE

Krzysztof J. Stencel
University of Warsaw
98 PUBLICATIONS **412** CITATIONS
SEE PROFILE

2009 IEEE/ACM International Conference on Automated Software Engineering

Towards a Comprehensive Test Suite for Detectors of Design Patterns

Patrycja Wegrzynowicz
Software R&D Department
NASK Research and Academic Computer Network
Warsaw, Poland
patrycjaw@nask.pl

Krzysztof Stencel
Institute of Informatics
Warsaw University
Warsaw, Poland
stencel@mimuw.edu.pl

***Abstract*—Detection of design patterns is an important part of reverse engineering. Availability of patterns provides for a better understanding of code and also makes analysis more efficient in terms of time and cost. In recent years, we have observed a continual improvement in the field of automatic detection of design patterns in source code. Existing approaches can detect a fairly broad range of design patterns, targeting both structural and behavioral aspects of patterns. However, it is not straightforward to assess and compare these approaches. There is no common ground on which to evaluate the accuracy of the detection approaches, given the existence of variants and specific code constructs used to implement a design pattern. We propose a systematic approach to constructing a comprehensive test suite for detectors of design patterns. This approach is applied to construct a test suite covering the Singleton pattern. The test suite contains many implementation variants of these patterns, along with such code constructs as method forwarding, access modifiers, and long inheritance paths. Furthermore, we use this test suite to compare three detection tools and to identify their strengths and weaknesses.**

***Keywords*-test suite; design patterns; detection;**

I. Introduction

A design pattern [1] is a general reusable solution to a commonly occurring problem in software design. Design patterns facilitate the construction of quality designs. As well as being useful in the construction of software systems (forward engineering), they also aid in the analysis of 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 in gaining a better understanding of code and makes analysis more efficient in terms of time and cost. Moreover, detection of design patterns can aid in better documentation of code (e.g., generation or verification of documentation) or assessing its quality (e.g., 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 (e.g., [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]) can detect a fairly broad range of design patterns, targeting structural as well as behavioral aspects of patterns. However, the assessment and comparison of these approaches is not a straightforward task. There is no common ground on which to evaluate the accuracy (precision and recall) of the detection approaches, given the existence of implementation variants and specific code constructs used to implement a design pattern.

Until recently, research in the field of pattern detection has focused on novel approaches. However, the latest papers ([14], [15]) highlight the importance and difficulties of assessment and comparison of pattern detectors.

Following the advice of [14], who proposed the usage of '*a common set of patterns, both structural as well as behavioural, with well-defined implementation variants*' to assess the accuracy of pattern detectors, we focused on the construction of a comprehensive test suite that includes both the correct and incorrect implementation variants. The main goal was to facilitate construction of pattern variants as well as to provide a ready-to-use test suite. By providing a test suite, we aid in the comparison of various approaches for detecting design patterns. Additionally, the test suite allows us to identify the strengths and weaknesses of pattern detectors, which can be helpful in improving these tools.

Contributions of this paper:

1) We present a systematic approach to the construction of a comprehensive set of the correct and incorrect implementation variants of design patterns. In [16], we presented a few ad-hoc implementation variants of the Singleton pattern. Then, we realized that the creation of new implementation variants can be systematized in order to produce a complete test suite for a given design pattern. For the Singleton, we obtained 56 correct variants through this systematic process, rather than the few variants that could have been created ad-hoc.
2) As a proof-of-concept for Contribution 1, we present a test suite covering the correct and incorrect variants of Singleton. Although the variants have been implemented in Java, most of them can be easily adapted to

1527-1366/09 $29.00 © 2009 IEEE
DOI 10.1109/ASE.2009.85

other programming languages.

3) We evaluate and compare three pattern detectors (PINOT, DPD Tool, and D-CUBED) based on the test suite.
4) We identify the strengths and weaknesses of each pattern detector evaluated.

II. Construction of Test Suite

A comprehensive test suite for design pattern detection must cover both correct (i.e., conforming to the contract of a pattern) and incorrect (i.e., violating the contract of a pattern) implementation variants of design patterns.

Figure 1 presents a systematic approach to the construction of the correct and incorrect implementation variants of design patterns. To construct these two sets, we start with the standard implementation variant of a pattern (e.g., the one provided by GoF). Then, we apply pattern-preserving transformations (pattern-specific and generic) to this variant. In this way, we obtain the set of correct implementation variants. Based on the correct variants, we then produce the incorrect variants by applying pattern-distorting transformations.

A. Pattern-Preserving Transformations

A *pattern-preserving transformation* is a code transformation that preserves the design intent. If such a transformation is applied to a correct implementation variant of a design pattern, then this variant remains correct. We distinguish between two types of pattern-preserving transformations: generic and pattern-specific.

A *generic pattern-preserving transformation* refers to a generic programming concept, namely, abstractness, invocation, or other. Such a generic transformation can be safely applied to a design pattern, preserving its correctness. (It is applied only when appropriate; i.e., the variant contains a construct transformable by a particular generic transformation.)

A *specific pattern-preserving transformation* is characteristic to a particular design pattern. It deals with so-called pattern logical fragments, transforming them to different implementations valid in the context of a pattern.

A *pattern logical fragment* is a robust fragment of a design pattern that 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).

B. Pattern-Distorting Transformation

A *pattern-distorting transformation* is a code transformation that distorts the design intent. If such a transformation is applied to a correct implementation variant of a design pattern, then this variant becomes incorrect (i.e., it violates the contract of a pattern). We consider only pattern-specific distorting transformations because: (1) generic distorting transformations (applicable to any design pattern) would lead to random code quite distant from the original, and (2) we seek interesting cases for the test suite, and to be interesting they should resemble the correct ones.

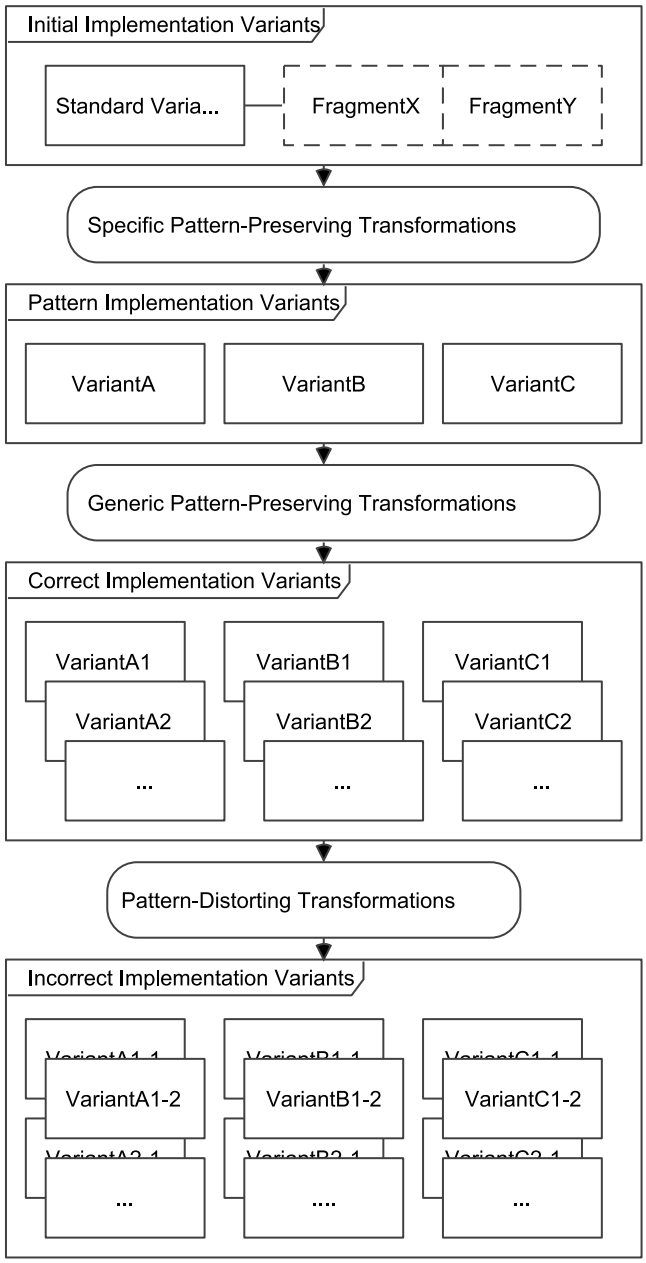

Figure 1. The systematic construction of correct and incorrect variants of a pattern.

III. Generic Transformations

The standard variants of design patterns are distilled and cleaned of all polluting code. However, in real world code, due to some additional functional or design requirements, there is often a need to add more code to an implementation

of a design pattern. Thus, it is important to distinguish between pattern-preserving and pattern-distorting constructs.

Figure 2 presents the identified generic pattern-preserving transformations: (1) abstractness transformations, (2) invocation transformations, (3) inheritance transformations, (4) aggregation transformations, and (5) method signature transformations

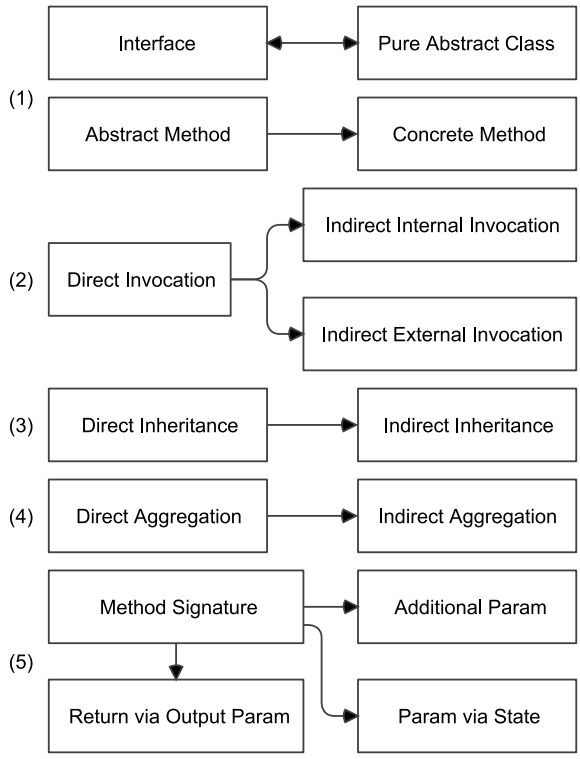

Figure 2. The pattern-preserving transformations.

Abstractness Transformations: These transformations refer to the property of abstractness for a class or a method. First, there is a two-sided transformation between an interface and a pure abstract class. This transformation is obvious since there exists a direct correspondence between these two constructs (often they are used interchangeably). The next transformation transforms an abstract method into a concrete method. In our opinion, this also is a natural transformation. In real world development, we often provide a default implementation rather than leave a method abstract. It is worthwhile to note that by combining these transformations (and possibly applying multiply), we can transform an interface into a concrete class. Summing up, as long as we can take advantage of polymorphic calls, abstractness or concreteness does not impact the intent of a pattern code.

```
abstract Product factoryMethod();
```

```
Product factoryMethod()
{ return new DefaultProduct(); }
```

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

```
abstract class Creator{...}
class CreatorA extends Creator{...}
```

```
abstract class Creator{...}
class Intermediary extends Creator{...}
class CreatorA extends Intermediary{...}
```

Invocation Transformations: This group refers to transformations of invocation and instantiation statements. A popular refactoring approach, in cases when a piece of code becomes complex, is to extract code into a separate method or class. This means that an invocation (or an instantiation) that was direct prior to refactoring becomes 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.

```
instance = new Singleton();
```

```
instance = createInstance();
...
Singleton createInstance()
{ return new Singleton(); }
```

Aggregation Transformations: These transformations follow the reasoning presented for invocation. 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 an example of such a transformation applied to the observers attribute:

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

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

Method Signature Transformations: Here we present the transformations on method signatures. We have identified three such transformations:

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

The example below presents an addition of a new parameter to `factoryMethod` (in the real world, such a parameter can be used to parameterize a construction of an instance or just to provide necessary values to the constructor):

```
Product factoryMethod()
```

```
Product factoryMethod(String name)
```

IV. Singleton Test Suite

This section describes the construction of a test suite covering the Singleton design pattern. We present the Singleton-preserving and Singleton-distorting transformations. We also describe the subsequent steps to generate the correct and incorrect implementation variants of the Singleton. Additionally, we explain which of the generic transformations are applicable and how they are applied to this design pattern.

A. Standard Variant

The standard variant of the Singleton is shown in Figure 3. We identified the following logical fragments of this pattern:

1) Instantiation (of a singleton object) – in the standard variant, realized by lazy instantiation;
2) Placeholder (for a singleton instance) – in the standard variant, realized by a direct aggregation in the singleton class itself;
3) Access point (to a singleton instance) – in the standard variant, implemented as a static public getter method;
4) Finality (of a singleton class) – it is not possible to subclass a singleton in the standard variant (its constructor is private).

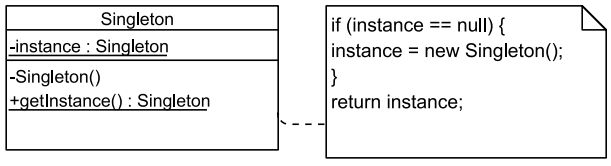

Figure 3. The standard variant of the Singleton pattern.

B. Singleton-Preserving Transformations

The following Singleton-preserving transformations on the identified logical fragment are proposed:

1) Instantiation: To Eager (A) – the lazy instantiation is replaced with an eager instantiation.
2) Placeholder: Inner Class (B) – the singleton instance is held as a static attribute of an inner class.
3) Placeholder: External Class (C) – the singleton instance is held as a static attribute of a class from within the same package.
4) Access Point: Inner Class (D) – the public static method is moved to an inner class of a singleton.
5) Access Point: External Class (E) – the public static method is added to a newly created class in the same package (the visibility of the Singleton access method is changed to protected).
6) Access Point: Attribute (F) – the static singleton instance is made public (eagerly instantiated, with the access method removed).
7) Access Point: Protected (G) – the visibility of the access method is changed to 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).

After the transformation of the standard variant of Singleton using the above transformations, we obtain 9 correct implementation variants of Singleton (A, B, C, D, E, F, G, and H, along with S - the standard variant).

C. Applicable Generic Transformations

In this subsection, we discuss the applicability of the generic pattern-preserving transformations to the before-generated variants. Table I provides an overview of which generic transformation can be applied to which Singleton variants.

The abstractness transformations are applicable only to the variant H, as it is the only variant with an abstract class. In addition to the original variant H, these transformations generated the interface singleton and the concrete singleton.

The internal invocation transformation is applicable to all variants (to an invocation of the singleton constructor), whereas the external invocation transformation can be applied only to H, where the constructor is protected (in Java, this also implies package visibility).

Similarly to the abstractness transformations, the inheritance transformation can be performed only on the variant H, which involves subclassing.

The indirect aggregation variant can be produced for each available initial variant.

Finally, the method signature can be applied to the access method (available in all variants except F), producing two new variants for each initial variant: an additional parameter

variant, and a return via an output parameter variant (as Java does not support pass-by-reference semantics, this is modeled with a helper class). Additionally, we can apply the additional parameter transformation to the constructor (available in all variants).

The total number of constructed correct implementation variants of Singleton is 56. The code of these variants is available at [17].

Table I
APPLICABILITY OF THE GENERIC TRANSFORMATIONS TO THE SINGLETON VARIANTS.

	A	B	C	D	E	F	G	H	S
Abst.	×	×	×	×	×	×	×	✓ (2)	×
Inv.	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (2)	✓ (1)
Inh.	×	×	×	×	×	×	×	✓ (1)	×
Aggr.	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)
Meth. Sign.	✓ (3)	✓ (3)	✓ (3)	✓ (3)	✓ (3)	✓ (1)	✓ (3)	✓ (3)	✓ (3)

✓ The group of the transformations is applicable to the variant
× The group of the transformations is not applicable to the variant
(N) The number of new variants generated by the transformation group

D. Singleton-Distorting Transformations

A violation of the Singleton contract occurs if, in addition to the controlled instance, another instance is created. To create another instance of a singleton, one needs access to its constructor.

To identify the Singleton-distorting transformations we have considered two cases:

- relaxation of the visibility of a Singleton constructor, thus making possible an instantiation of a Singleton via external classes (consider that even with a private constructor, the contract of a Singleton may be violated e.g. by an inner class or by the Singleton itself)
- instantiation of another Singleton object.

The proposed Singleton-distorting transformations include:

1) Visibility: Public – the visibility of a singleton constructor is changed to public (56 new instances).
2) Internal Instantiation: By Singleton – another instance is created and used by the singleton itself (56 new instances).
3) Internal Instantiation: By Inner Class – another instance is created and used by an inner class of the singleton (56 new instances).
4) External Instantiation: By Package Class – another instance is created and used by a different class in the same package (1 new instance).
5) External Instantiation: By Subclass Class – another instance is created and used by a subclass (1 new instance).
6) External Instantiation: By External Class – another instance is created and used by a class outside the package of the singleton (56 new instances, combined with the transformation of the constructor to public visibility).

We have grouped these variants into two sets:

- obvious violation of the Singleton contract (all variants generated by Internal or External Instantiation; with a total count of 170)
- accessibility violation of the Singleton constructor (total count of 56)

E. Real Code Examples

To prove that the variants generated by our approach exist in real code, we provide a series of examples located in JHotDraw60b1 [18].

JHotDraw is a Java GUI framework for technical and structured graphics, initially developed as a design exercise by Erich Gamma, one of the fathers of design patterns.

The `DisposableResourceManager` (Figure 4) is a singleton documented in the code. It can be obtained from the standard variant via the following set of the pattern-preserving transformations:

1) Instantiation: To Eager (singleton-preserving transformation)
2) Finality: Abstractness with Subclassing (singleton-preserving transformation)
3) Placeholder: External Class (singleton-preserving transformation).

The next example (Figure 5) required an additional transformation (in addition to the transformations for `DisposableResourceManager`), namely:

1) Internal Invocation (generic pattern-preserving transformation).

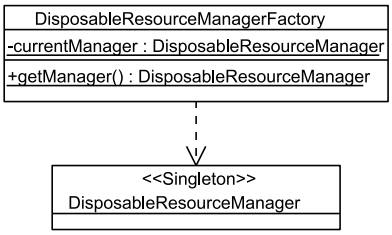

Figure 4. The Disposable ResourceManager singleton (of JHotDraw).

V. EVALUATION OF PATTERN DETECTORS

In this section, we show the results of evaluation of three state-of-the-art pattern detectors (PINOT, DPD Tool, and D-CUBED). The evaluation was conducted using the test suite proposed in this paper.

PINOT [6] is a command-line tool written in C++ and based on jikes — the IBM Java compiler. The detection

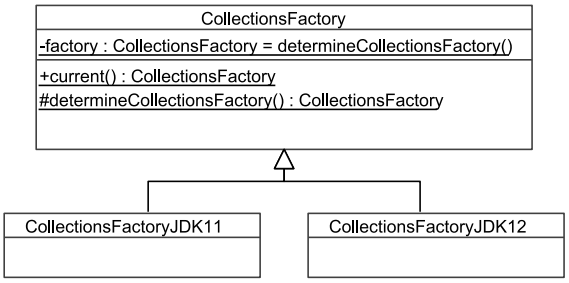

Figure 5. The CollectionsFactory singleton (of JHotDraw).

algorithms are hard-coded in PINOT, so it is hard to experiment by modifying the detection approach. PINOT produces a useful verbose report summarizing detected pattern instances. Each detected pattern instance contains the classes and methods, together with their roles in the pattern.

DPD Tool (Design Pattern Detection Tool) [19] is a desktop application written in Java. It uses a graph-based similarity scoring algorithm to detect the patterns, offering almost immediate results to the user. The report produced is robust and simple, yet it provides all necessary information about the detected pattern instances.

Similarly to PINOT, D-CUBED [13] is a command line tool written in Java. However, contrary to PINOT, its detection queries are not hard-coded but instead written in SQL and separated in a configuration file. Thus, it is possible to modify existing queries as well as to add new ones. A text report produced by D-CUBED contains running times of subsequent phases as well as detected pattern instances. Each pattern instance is identified by its key class (for example, *Creator* in case of the Factory Method patterns).

Table II
RESULTS OF THE PINOT AGAINST THE CORRECT VARIANTS.

	A	B	C	D	E	F	G	H	S
Clean	✓	×	×	×	×	×	×	×	✓
Abst.	–	–	–	–	–	–	–	×	–
Inv.	×	×	×	×	×	×	×	×	✓
Inh.	–	–	–	–	–	–	–	×	–
Aggr.	×	×	×	×	×	×	×	×	×
Meth. Sig.	✓ (2/3)	×	×	×	×	×	×	×	✓ (2/3)

✓ The tool correctly reported the pattern variants
(K/N) The tool correctly reported *K* out of *N* pattern variants
× The tool failed to report the pattern variants
– The combination is not allowed

Tables II, III, and IV present the results of the detectors (PINOT, DPD, D-CUBED, respectively) when executed against the correct implementation variants. PINOT shows a very high number of false negatives, whereas D-CUBED performs exceptionally well on this set, finding all instances. DPD performs at medium accuracy. These three tools use very different search algorithms:

Table III
RESULTS OF THE DPD TOOL AGAINST THE CORRECT VARIANTS.

	A	B	C	D	E	F	G	H	S
Clean	✓	×	×	✓	✓	✓	✓	✓	✓
Abst.	–	–	–	–	–	–	–	✓	–
Inv.	✓	×	×	✓	✓	✓	✓	✓	✓
Inh.	–	–	–	–	–	–	–	✓	–
Aggr.	×	×	×	×	×	×	×	×	×
Meth. Sig.	✓	×	×	✓	✓	✓	✓	✓	✓

✓ The tool correctly reported the pattern variants
× The tool failed to report the pattern variants
– The combination is not allowed

Table IV
RESULTS OF THE D-CUBED AGAINST THE CORRECT VARIANTS.

	A	B	C	D	E	F	G	H	S
Clean	✓	✓	✓	✓	✓	✓	✓	✓	✓
Abst.	–	–	–	–	–	–	–	✓	–
Inv.	✓	✓	✓	✓	✓	✓	✓	✓	✓
Inh.	–	–	–	–	–	–	–	✓	–
Aggr.	✓	✓	✓	✓	✓	✓	✓	✓	✓
Meth. Sig.	✓	✓	✓	✓	✓	✓	✓	✓	✓

✓ The tool correctly reported the pattern variants
× The tool failed to report the pattern variants
– The combination is not allowed

- PINOT searches for a static getter method which simple returns the singleton instance or performs lazy initializations. As the result shows, PINOT imposes an additional structural constraint on a singleton class — it must have a static field of the same type as the enclosing class.
- DPD depends only on the presence of a static field of the same type as the enclosing class. It works well in many cases, but it gets lost when the static field is moved to another place.
- D-CUBED verifies only the assignment paths from the instance to the static attributes, completely ignoring the structural aspect. In this way, D-CUBED is resistant to any structural changes that preserve the intent (e.g., if we assign the created singleton instance to a static field for future reuse).

In evaluations on the incorrect implementation variants, the results turned out to be even more diverse. It is important to note that we ran the incorrect variant test only for the incorrect variants corresponding to the true positive correct variants of a given tool. PINOT does not detect violations of the Singleton instance creation; it claims that all variants violating the rule are positive implementations of the Singleton (all false positives). Even though Pinot analyzes the behavior, it is satisfied to find a lazy instantiation block, while ignoring the leakage of another instance. DPD does not have the chance to catch this violation, as it analyzes only the structure (also all false positives). Only D-CUBED detected these violations (all true negatives).

PINOT requires a singleton constructor to be private, thus it did *not* report the public accessibility incorrect variants as

singletons, whereas DPD Tool and D-CUBED ignore access modifiers, and both reported all of the public accessibility incorrect variants as singletons.

VI. Related Work

In recent years, we have observed a continual improvement in the field of design pattern recognition. There now exist a number of detection approaches, targeting structural as well as behavioral aspects of patterns. However, these approaches are not perfect and sometimes fail to capture source code intent. An assessment and comparison of the existing approaches to pattern detection is not a simple task.

[14] highlights the importance of a proper evaluation of accuracy in pattern detection. They enumerated six major issues related to accuracy measurement: (1) design patterns and variants, (2) pattern instance type, (3) exact and partial matching, (4) system size, (5) precision and recall, and (6) control set. To make detection approaches more comparable, they proposed the use of (among others): (1) a common set of implementation variants of patterns and (2) well annotated control sets for a set of applications.

[15] follows the second advice of [14]: to construct, with community effort, control sets for a set of applications. They created a web application to support this task. The database stores: (1) pattern detection tools, (2) software applications with manually inspected instances of design patterns, and (3) the results of runs of detection tools against software. All the data can be viewed and modified in different contexts. Users also have the ability to add new pattern instances, new tools, or new software. In this way, the benchmark can evolve with the support of the entire community.

A benchmark based on existing software (such as in [15]) is obviously useful. It tests and compares the detectors on real world sources. However, to provide a broad spectrum of implementation variants, such a benchmark must include at least dozens of applications. Each application must be carefully reviewed to identify all the correct instances of patterns. Such a manual review, considering the size of real-world software source code, is time-consuming and error-prone. Thus, the number of false negatives will be difficult to determine. Moreover, the implementation variants present in real software are often polluted with other code imposed by various functional or design constraints. Therefore, an identification of the strengths and weaknesses of a particular detector (in regards to specific code constructs or techniques) is more difficult in such a benchmark. These are the main reasons why the focus of our work is on the first suggestion of [14], i.e., well-defined implementation variants of design patterns. We present a systematic approach to construction of variants of design patterns. In this way, we can achieve a comprehensive set of correct and incorrect implementation variants leveraging various well-defined programming techniques. A measurement of accuracy (precision and recall) becomes easier when we can easily identify false positives and false negatives in the results. The techniques used to generate the implementation variants can provide hints to researchers on how to improve their detection approaches by the explicit identification of variants or constructs that were incorrectly detected.

VII. Conclusion

We have presented a systematic approach to construction of the well-defined correct and incorrect implementation variants of design patterns. The approach is based on the concept of pattern-preserving (generic as well as pattern specific) and pattern-distorting transformations. Then, we described an application of this method to construct the comprehensive set of implementation variants for the Singleton pattern. This test suite has been used to assess the accuracy of three pattern detectors. The test suite has proved its usefulness by revealing certain drawbacks of existing approaches.

Future work encompasses automated support for applying the transformations (using a custom tool or one of the existing code transformation/refactoring tools), since the (semi-)manual creation of variants is tedious and error-prone. The available transformations can be further extended to include more language-specific constructs. The goal is to provide a comprehensive test suite covering all GoF patterns.

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. Wegrzynowicz, "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] P. Wegrzynowicz, "The singleton benchmark," http://www.dcubed.pl/.

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

[19] 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.