Evaluation of Accuracy in Design Pattern Occurrence Detection

Niklas Pettersson, Welf Löwe, and Joakim Nivre

Abstract—Detection of design pattern occurrences is part of several solutions to software engineering problems, and high accuracy of detection is important to help solve the actual problems. The improvement in accuracy of design pattern occurrence detection requires some way of evaluating various approaches. Currently, there are several different methods used in the community to evaluate accuracy. We show that these differences may greatly influence the accuracy results, which makes it nearly impossible to compare the quality of different techniques. We propose a benchmark suite to improve the situation and a community effort to contribute to, and evolve, the benchmark suite. Also, we propose fine-grained metrics assessing the accuracy of various approaches in the benchmark suite. This allows comparing the detection techniques and helps improve the accuracy of detecting design pattern occurrences.

Index Terms—Patterns, object-oriented design methods, measurement techniques, evaluation, reverse engineering, reengineering, restructuring.

1 Introduction

Understanding a system's design is essential for its maintenance and further development. Since software systems may be large and complex, developed under tight deadlines with frequently changing requirements, the design is almost always scarcely documented. It is therefore important that software comprehension be supported by design analysis tools. However, design is hard to analyze automatically since it is not tangible. Design patterns (e.g., standard GoF Design Patterns [1] or domain-specific patterns) describe standard design solutions to recurring problems. Identifying occurrences of such patterns in the code may be very helpful in the understanding process. Identifying antipattern [2] occurrences, i.e., identifying design flaws, may also be helpful when understanding a software system. Other applications for design pattern occurrence detection include software quality measurement [3] and optimization [4], [5].

Different design pattern detection methods exist varying in dimensions such as flexibility, efficiency, and accuracy. A method is *flexible* if patterns can be easily specified in some specification language—in contrast to a method that only supports some set of patterns hard-coded in the matching algorithm. A method is *efficient* if it requires memory and execution time polynomial in the size of the software system. A method is *accurate* if it finds all occurrences of the specified pattern and nothing else.

- *N. Pettersson and W. Löwe are with the School of Computer Science, Physics, and Mathematics, Linnaeus University, 351 95 Växjö, Sweden. E-mail: {niklas.pettersson, welf.lowe}@lnu.se.*
- *J. Nivre is with the Department of Linguistics and Philology, Uppsala University, Box 635, 751 26 Uppsala, Sweden. E-mail: joakim.nivre@lingfil.uu.se.*

Manuscript received 7 Mar. 2008; revised 3 Oct. 2009; accepted 11 Oct. 2009; published online 12 Nov. 2009.
Recommended for acceptance by W. Schaefer.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-03-0093.
Digital Object Identifier no. 10.1109/TSE.2009.92.

Comparability of detection methods (or tools) with respect to accuracy is interesting in itself. It also facilitates comparison with respect to other nonfunctional properties; it is important that accuracy of detection methods is approximately on the same level before comparing their efficiency or flexibility.

In the relatively young field of design pattern occurrence detection (design patterns themselves are relatively new), the evaluation of accuracy has not yet reached the maturity of other areas of computer science, e.g., parsing of natural languages. For instance, no standard benchmarks are available that facilitate the comparison of pattern detectors, whereas such resources have been developed for natural language parsers (see, e.g., Buchholz and Marsi [6]).

Sim et al. observe that the development of benchmarks in computer science disciplines is often accompanied by technical progress and community building [7]. The lack of such benchmarks in turn makes it hard to further develop a field by adopting the successful and forgetting the less promising approaches. In the long run, it may be an obstacle to exploiting design pattern detection methods in applications.

Establishing standard benchmarks for comparing accuracy of design pattern occurrence detection is not an easy task, partly because the exact delimitation of particular patterns is seldom or never completely clear-cut. Nevertheless, experience from the natural language processing community clearly shows that this complexity can be mastered, cf., Section 3.1.

Sim et al. also give several prerequisites for the establishment of benchmarks as an important driving force in research, e.g., the existence of a research community, a minimum maturity of the discipline, and general ethos of collaboration in the community [7]. The pattern detection community consists of several groups around the world. Besides our group, there are, e.g., the Pattern Trace Identification, Detection, and Enhancement in Java (Ptidej) team led by Yann-Gaël Guéhéneuc at the Université de Montréal [8], [9], [10], [11], Jing Dong and Yajing Zhao at the

University of Texas at Dallas [12], [13], [14], and the Software Engineering Group at the University of Paderborn [2], [15], [16], [17], [18], [19]. Both a minimum maturity of the discipline and general ethos of collaboration are indicated by the communication of progress in conferences like the Conference on Software Maintenance and Reengineering (CSMR) [9], [10], [20], [21], the International Conference on Program Comprehension (ICPC) [16], [22], [23], [24], and the discussion-oriented Working Conference on Reverse Engineering (WCRE) [25], [26]. Altogether, the pattern detection community is apparently ripe for and could profit from standard benchmarks and accuracy assessments.

As a basis for a creating a well-accepted and methodologically profound benchmark, this paper presents the results of a survey of accuracy evaluations of design pattern occurrence detection methods covering 14 papers [3], [9], [15], [21], [22], [23], [25], [27], [28], [29], [30], [31], [32], [33].[1] All papers perform analysis and evaluation on either C++, Java, or Smalltalk (although some approaches may have the possibility of analyzing several similar languages because of an intermediate representation, e.g., [22], [29]). We found several methodological problems in these accuracy evaluations, implying at least a limited generality of the conclusions drawn. Moreover, we found several variables differing in the accuracy evaluations making them to a large extent incomparable. In particular, we focus on the following problems and variables:

Problem 1 (Design Patterns and Variants). *We, the design pattern occurrence detection community, use different design patterns in the evaluations. Furthermore, we allow different implementation variants of these design patterns. The difficulty of detection may depend on the pattern to detect and even its implementation variant. Hence, accuracy evaluations may be incomparable when they vary the patterns and their implementation variants.*

Problem 2 (Systems). *We evaluate our detection approaches on different systems with different characteristics. In particular, systems of small size (i.e., a few hundred classes) are often used. However, accuracy results in small systems may not necessarily scale to larger systems.*

Problem 3 (Gold Standard). *In the evaluations, we often construct the set of actual occurrences of particular design patterns in specific software systems in an ad hoc manner. (This reference set is denoted by a* gold standard, *cf. Section 2 for a definition.) This implies that the correctness of the accuracy results is at least questionable. Furthermore, we do not use the same gold standards in different evaluations. Obviously, using exactly the same gold standards is essential for comparable accuracy results.*

Problem 4 (Precision and Recall). *We do not always evaluate accuracy using both* precision*—measuring what fraction of detected pattern occurrences are real—and* recall*—measuring what fraction of real pattern occurrences are detected. This can be problematic since an increase in precision is often related to a decrease in recall (and vice versa).*

Problem 5 (Pattern Occurrence Type). *A pattern occurrence is represented as a tuple of the participating classes, interfaces, and possibly methods. We use different tuple types representing occurrences. This may largely influence the calculated precision and recall measurements.*

Some of these problems and variables have been discussed to some extent before (e.g., [3], [9]). However, they are all still open issues that need to be resolved before creating accuracy benchmarks for comparing design pattern occurrence detection methods and tools. Therefore, this paper goes beyond the discussion of the problems and also suggests approaches to overcome them. In particular, it makes the following contributions:

1. We present a detailed survey of design pattern detection papers that analyze accuracy and highlight problems and variables in the accuracy evaluation.
2. We complement the survey results with controlled experiments showing that certain problems and variables discussed are indeed a threat to generalizability and comparability of the results.
3. We propose solutions to the problems, including the following:
   a. a benchmark suite varying the variables in a controlled manner (e.g., selection of design patterns and software systems as well as the selection of pattern occurrence types),
   b. the introduction of the weighted F-Score metric to ease comparisons of pattern occurrence detectors, which have either high recall or high precision, and
   c. the partial match metrics, which are a more fine-grained complement to the exact match metrics currently used.

The remainder of the paper is structured as follows: Section 2 recapitulates some necessary definitions for accuracy evaluation. Section 3 relates to other surveys in the field of design pattern detection. It also discusses methods for accuracy evaluation used in related branches of computer science, e.g., natural language processing and information retrieval. Sections 4-8 discuss each of the problems and variables introduced. They define the problem in more detail, report the specific survey results, give experimental support for our claims, and propose a solution to the problem. Section 9 introduces the partial match metrics. Section 10 contains the conclusions of the paper and points to directions for future work. The Appendix describes the experimental setup used to give experimental support for the claims in Sections 4-8.

2 Basic Concepts and Definitions

A design pattern names, abstracts, and identifies the key aspects of a common *design* that makes it useful for creating a reusable object-oriented design [1]. An intrinsic property of design patterns is that they can be *implemented* very differently (e.g., depending on program language, personal style, or other design decisions). Hence, a design pattern can be seen as a set of software patterns. A software pattern provides a specification for one of the implementation variants. A software pattern consists of a *structural*

1. We investigated five additional papers [16], [17], [20], [34], [35] describing design pattern detection methods. They were excluded from the survey since they did not contain accuracy evaluations.

description and a *behavioral protocol*. The structural description usually introduces *roles* of program entities, like classes and methods and their relations. The behavioral protocol adds constraints on the behavior of runtime incarnations of the program entities like the order in which objects are created and methods invoked. The description of a software pattern can contain information that is hard or impossible to analyze automatically.

For example, one software pattern for the *Observer Design Pattern* can be defined as follows: This *Observer Software Pattern* contains nine roles of entities: concrete/abstract observer and subject classes, concrete/abstract update-method, as well as attach, detach, and notify observer methods. The structural description is the following: The methods should be structurally contained in the corresponding classes. Furthermore, both the *attach* and *detach* methods should take a parameter of the (abstract) observer class. Finally, the concrete *update* method should override the abstract *update* method. The behavioral protocol additionally constrains the behavior of the runtime instances, i.e., of observer and subject instances: A subject instance must be used as a parameter in an *attach* method before it can be detached or updated. Furthermore, the *notify* method must perform atomic updates of all attached observer instances.

The detection of a software pattern in a program system delivers a set of *pattern occurrences*, i.e., tuples of program entities in the different pattern roles. They need to conform to the constraints of the structural description and their runtime instances need to conform to the behavioral protocol of the software pattern. This can only be approximated since completely analyzing the behavior is undecidable, in general.

Sometimes software pattern detection only returns entities in the key roles of a pattern, e.g., the subject classes of the Observer pattern, even though more roles are actually used in the detection. We refer to the tuple of key roles of a detection approach as its *occurrence type* and to the number of key roles as its *occurrence size*. A possible pattern occurrence type for the Observer pattern described earlier is the quadruple $(attach, detach, notify, update)$. Another possible occurrence type is $(subject)$. The sizes are 4 and 1, respectively.

A *gold standard* defines all actual occurrences of a particular design pattern in a specific software system. When comparing the output of a pattern detector to the gold standard, a pattern occurrence can be either a true positive, a false positive, a true negative, or a false negative. The set of *True Positives* TP contains all pattern occurrences proposed by a pattern detector that are also contained in the gold standard. The set of *False Positives* FP contains all pattern occurrences proposed by a pattern detector but not contained in the gold standard. The set of *True Negatives* TN contains all pattern occurrences not proposed by a pattern detector and not contained in the gold standard either. The set of *False Negatives* FN contains all pattern occurrences not proposed by a pattern detector but contained in the gold standard.

Two metrics used to evaluate the accuracy of retrieved information are *precision*, i.e., how large is the fraction of true positives in the retrieved patterns, and *recall*, i.e., how large is the fraction of true positives in all actual patterns. The precision P and recall R are defined as $P = t_p/(t_p + f_p)$ and $R = t_p/(t_p + f_n)$, respectively, where $t_p = |TP|$, $f_p = |FP|$, and $f_n = |FN|$.

3 Related Work

In this section, we discuss accuracy evaluation methods in other domains of computer science and related surveys of design/software pattern detection approaches.

3.1 Methods in Related Domains

The problems encountered in evaluating and comparing the accuracy of different methods are not unique to pattern detection, and it may therefore be instructive to consider some of the steps taken to come to terms with these problems in other branches of computer science.

One relevant comparison is the evaluation of natural language processing systems, in particular, natural language parsers. The task of such a parser is to provide a syntactic analysis, e.g., a context-free parse tree, for every sentence occurring in a natural language text. Since the correct syntactic analysis can currently be established only by human experts, evaluating the accuracy of such a parser presupposes that samples of text in the relevant language have been manually annotated with the correct analysis for each sentence. Large collections of syntactically annotated samples of this kind are commonly referred to as *treebanks* [36], the most well-known example being the Penn Treebank of American English [37]. Establishing large manually validated gold standards in this way addresses Problem 2 (i.e., systems) and Problem 3 (i.e., gold standards).

It is worth pointing out that when establishing treebanks for the evaluation of natural language parsing, there is sometimes disagreement even among experts as to what should count as the correct analysis of a given sentence or text. This is in many ways similar to the situation for pattern detection, where it can often be difficult to agree on the precise definition of a pattern and what exactly should count as an occurrence of a pattern. In the case of natural language parsing, this problem is usually handled by adopting an explicit set of guidelines that annotators can rely on in order to produce a consistently annotated data set. In one way, this can be seen as a way of creating an artificial sense of precision, but treebanks have nevertheless proved to be a very valuable resource in driving and focusing research efforts on natural language parsing. Moreover, anyone is free to propose alternative annotation decisions for a given data set as long as these are consistently applied and explicitly documented.

Moreover, using both precision and recall—and well-defined integrations of these two metrics, e.g., the so-called F-Score—is common practice for the evaluation of accuracy in natural language processing. This practice addresses Problem 4 (i.e., precision and recall).

However, since the accuracy of current natural language parsers, when measured by the proportion of sentences that are assigned a completely correct parse tree, is still rather low (well below 50 percent except under very favorable conditions), there has been a need for more fine-grained evaluation metrics that also consider partially correct parse trees. The most well-known evaluation suite is the so-called PARSEVAL metrics [38], [39], containing metrics that

consider both precision and recall at the level of subtrees (or constituents) as well as the number of conflicts in bracketing between the parser output and the gold standard, i.e., the treebank annotation. Considering subtrees in natural language processing corresponds to considering occurrence types and sizes smaller than the full pattern (size); it is therefore relevant to Problem 5 (i.e., pattern occurrence types). We will also return to this discussion in Section 9.

In information retrieval, there is also a long tradition of evaluating the quality of systems by measuring both precision and recall, i.e., how many of the documents retrieved are relevant and how many of the relevant documents are retrieved [40]. Recall can especially be very problematic to measure in settings where the number of documents that could potentially be retrieved is very large, as is the case, e.g., in document retrieval on the Web.

The series of Text Retrieval Conferences (TREC) has developed a methodology for dealing with incomplete information in this sense, by using a pooling process. The idea is that every system participating in the evaluation contributes a list of n top-ranked documents, and that all of the documents appearing in one of these lists are submitted to a manual relevance judgment. In this way, systems can be compared on precision with respect to the set of n top-ranked documents, and on recall with respect to the common pool of documents. Moreover, the established pools of relevance judged documents can later be reused for new evaluations, although this may require slightly different evaluation metrics unless we are willing to make the assumption that any document outside the pool is irrelevant [41]. This work can be taken as inspiration when addressing Problem 2 (i.e., system), Problem 3 (i.e., gold standard), and Problem 4 (i.e., precision and recall) since it provides a methodology to incorporate new systems into accuracy evaluations, to construct gold standards in an efficient but semi-automated way, and to adjust the definition of precision and recall accordingly.

Without going further into the details, we think the examples from natural language parsing and information retrieval show that it is possible to improve the quality and comparability of evaluation methods by community efforts focusing on the development of large, manually validated gold standards (treebanks, document pools) and batteries of evaluation metrics that go beyond simple precision and recall based on exact match. Design pattern detection resembles natural language parsing in that the objects retrieved have a complex internal structure, making partial match metrics relevant, and it resembles information retrieval in that the distribution of relevant objects normally is very sparse (i.e., heavily dominated by negative instances).

3.2 Related Surveys

This section relates to other studies of survey character in the field of design pattern detection. It just discusses survey papers, not individual approaches. Detailed information about the individual approaches is presented in the following sections as part of our own survey.

Guéhéneuc et al. [10] present a comparative framework for design recovery tools. Concerns like *context* (what is the context for the use of the recovery tool), *intent* (what is the recovery tool's purpose), and *users* (what is expected from the users) are discussed. Even though we also propose a comparative framework, our proposal is orthogonal to that of Guéhéneuc et al. Our focus is on improving comparability in accuracy evaluation of design pattern detection.

Fülöp et al. [42] present a comparison of three design pattern detection tools. They are compared with respect to the number of pattern hits (not accuracy), speed, and memory consumption. Again, the focus is not the same.

Pettersson et al. [43] performed a survey of pattern detection evaluations presented in the scientific literature. This survey revealed six major problems regarding comparability of accuracy between pattern detectors. The authors proposed to seek solutions to these problems in related domains, most notably the domain of natural language parsing. Some general guidelines regarding a common benchmark were also given. For instance, large systems should be included in the benchmark as well as a diverse set of patterns and variants. The current paper presents a continuation of this work.

Fülöp et al. [44] build on the works above and present an infrastructure for building design pattern detection benchmarks. It contains systems with pattern occurrences found by tools and it is extensible with more systems and their patterns occurrences. Moreover, it contains a human-approved gold standard for two C++ systems: NotePad++ and a C++ reference implementation of the standard GoF design patterns. Finally, it provides a Web interface for judging the instances' degree of correctness and uses the community decision for telling false from true positives. Finding additional false negatives cannot be naturally tool-supported.

Dong et al. [12] present a review of design pattern detection approaches. Naturally, the sets of pattern detection approaches investigated overlap. There is also some methodological overlap in the studies, mainly in that both identify the design patterns and software systems that are used in evaluating detection approaches. We corrected some minor issues in the survey results of [12].[2] The main difference between the two surveys is that ours focuses on the problem of comparing accuracy of detection approaches. It analyzes the surveyed approaches accordingly, supports findings with experiments, and proposes solutions to the problems. Dong et al. focus on complementary aspects like the type of intermediate representation used and whether or not behavior is considered.

4 Problem 1: Design Patterns and Variants

This and the following sections each deal with one of the five problems of comparing design pattern detection discussed in Section 1. They follow the same structure: First, we define the problem. Second, we show that it can be observed in today's practice of design pattern detection. Third, we give experimental evidence for having isolated a relevant question. Finally, we propose a solution to the problem.

2. If possible, we compared our results to those of Dong et al. and double-checked with the original papers again whenever they differed. For instance, Dong et al. missed that Balanyi and Ferenc [29] also use the Builder and the Iterator patterns in their evaluation. However, the iterator is excluded from the survey of Dong et al.

TABLE 1
GoF Design Patterns Used in Evaluations

	Abstract Factory	**Adapter**	Builder	**Bridge**	Chain of Responsibility	Command	**Composite**	**Decorator**	Facade	**Factory Method**	Flyweight	Interpreter	Iterator	Mediator	**Observer**	Prototype	**Proxy**	**Singleton**	Strategy	State	**Template Method**	Visitor
Brown [27]					x		x	x													x	
Krämer, Prechelt [25]		x		x			x	x									x					
Antoniol et al. [22]		x		x			x	x									x					
Keller et al. [28]				x						x											x	
Niere et al. [15]				x			x												x			
Heuzeroth et al. [23]					x		x							x	x							x
Balanyi et al. [29]	x	x	x	x	x			x		x			x			x	x	x	x		x	x
Philippow et al. [30]												x						x				
Pettersson [31]															x							
Vokac [3]								x		x					x			x			x	
Costagliola et al. [21]		x		x			x	x									x					
Kaczor et al. [9]	x						x															
Shi and Olsson [32]	x	x		x	x		x	x	x	x	x			x	x		x	x	x	x	x	x
Tsantalis et al. [33]		x				x	x	x		x					x	x		x	x	x	x	x

Patterns used in at least five evaluations are in bold face.

4.1 The Problem

The selected patterns in an evaluation have a crucial impact on accuracy since, in general, different patterns are not equally hard to detect. Being able to detect a certain pattern with high accuracy depends on several factors, e.g., the allowed implementation variants, distinctive and unique descriptions for the variants (e.g., structural, behavioral, and metrics based), and the possibility of expressing those descriptions using the particular detection method.

4.2 Survey

Table 1 shows the design patterns most commonly used in evaluations of the community. Philippow et al. [30] detect all patterns defined by Gamma et al. [1]. However, evaluation is only performed using the Singleton and Interpreter patterns.[3] In the evaluation of Balanyi and Ferenc [29], not a single occurrence was found for several patterns.

According to previous experiments, it has turned out that some patterns are easier to detect than others. Krämer and Prechelt [25] as well as Antoniol et al. [22] go as far as stating that structural design patterns are easier to detect than the behavioral and creational patterns. This is probably an overgeneralization since the Template Method is a behavioral pattern and also relatively easy to detect [28]. Besides the Template Method, "easier patterns" include, according to the literature, the Factory Method [28], while harder patterns include Observer and Bridge [3], [23], [28]. According to a recent study by Shi and Olsson [32], all patterns of Gamma et al. [1] can be detected by structural and/or behavioral aspects except the Builder, Memento, Command, and Interpreter patterns. The structurally detectable patterns (structure-driven) are Bridge, Composite, Adapter, Facade, Proxy, Template Method, and Visitor. The behaviorally detectable patterns (behavior-driven) are Singleton, Abstract Factory, Factory Method, Flyweight, Chain of Responsibility, Decorator, Strategy, State, Observer, and Mediator. It could seem nonintuitive that some

3. Precision and recall values (P = R = 100%) are only presented for the Singleton and the Interpreter.

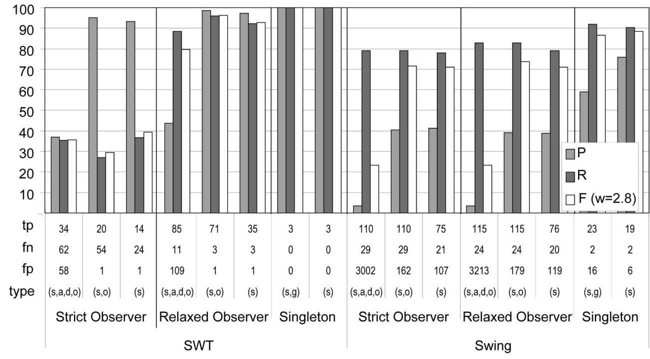

Fig. 1. P: Precision, R: Recall, F: F-score ($w = 2.8$) for different patterns and tuple types. tp: the number of true positives, fn: false negatives, fp: false positives, type: pattern occurrence type.

pairs of patterns are separated into different categories. For instance, the Proxy and Decorator are quite similar, but still separated into the structural and behavioral category, respectively. Despite the similarities, they differ in both intent and level of dynamism. The Proxy pattern statically hides information about an underlying object from a client. A Decorator pattern dynamically adds behavior to an underlying object (or chain of Decorators), which may also be accessed from a client without using the Decorator.

It is hard to compare the accuracy of two pattern detection approaches even if the same patterns are used in evaluations. One reason is that different implementation variants are used in evaluations. For instance, some approaches only consider a single implementation variant for each pattern, closely related to the design diagrams, e.g., [25]. Other approaches aim at detecting variants, e.g., [33]. Hence, not even the selection of a specific design pattern in the evaluation guarantees comparable accuracy results.

4.3 Experimental Support

To show the impact of the design patterns and their implementation variants on accuracy results, we detected the Singleton and the Observer patterns (in two variants) in the Swing and SWT libraries. For the complete experimental setup, we refer to the Appendix. We compared the accuracy of the Singleton and Observer patterns, as well as the accuracy between the two Observer variants. One variant is relaxed (knows nothing about concrete observers), whereas the other requires that the concrete update method in the concrete observer override the abstract update method of the observer, a direct superclass of the concrete observer (see also the Appendix). We count one pattern occurrence for a representative class of each pattern, i.e., a Singleton class and a Subject class count as one pattern occurrence, respectively.

The measurements are available in Fig. 1. The Singleton is detected in Swing with $P = 76\%$ and $R = 90.5\%$, whereas the strict variant of the Observer pattern is found with $P = 41\%$ and $R = 78\%$. The relaxed variant is detected with $P = 39\%$ and $R = 79\%$. Hence, the Singleton detection has significantly higher precision and recall than the Observer detections in Swing.

The situation is quite similar in SWT. The Singleton is detected with $P = 100\%$ and $R = 100\%$, whereas the strict

variant of the Observer pattern is found with $P = 93.3\%$ and $R = 37\%$. The relaxed variant is detected with $P = 97.2\%$ and $R = 92.1\%$. It shows that there can be large differences between the two variants of the Observer. The relaxed version has high precision and recall, whereas the strict version has high precision but a rather low recall. In this case, the reason is that the strict implementation variant requires a concrete observer role, which is not that common in the GUI library SWT. On the other hand, Swing has several concrete observers since we include the SwingSet2 demo application in the detection. In general, this supports our assumption that there are differences in difficulty of detection between different pattern implementation variants.

Altogether, the large differences in the accuracy of detecting the Singleton versus the Observer patterns confirm once more that different patterns are not equally hard to detect.

4.4 Proposed Solution

The solution to this problem is rather obvious: The comparison of detection methods requires standard pattern sets, including patterns that are both easier and harder to detect. A step in this direction can be seen in Antoniol et al. [22], where the authors detect the same patterns as Krämer and Prechelt [25].

The most commonly used design patterns are good candidates to be included in a benchmark.[4] The patterns used in five or more papers are shown in bold face letters in Table 1.

The classification of Shi and Olsson [32] is also interesting when defining appropriate benchmark patterns. We should include patterns requiring both structural detection and behavioral detection in our benchmark.

Finally, the selected patterns should be compatible with the systems selected for the benchmark. We return to this in Section 5 when we discuss the systems of the benchmark and in Section 6 when we discuss the construction of the gold standards.

Given the above reasons, we propose using the following patterns: first, the commonly used and structurally detectable patterns *Adapter*, *Bridge*, *Composite*, *Singleton*,[5] and *Template Method*; second, the commonly used patterns that need behavioral detection: *Decorator* and *Observer*. This core set of patterns can be extended if other patterns are commonly required by client applications, i.e., applications requiring pattern occurrences as input.

Also, we need to standardize the allowed implementation variants for the selected patterns. This can be defined either intentionally or extensionally. An intentional definition provides pattern specifications (e.g., diagrams showing allowed structure and behavior), whereas an extensional definition provides a gold standard enumerating all occurrences of the design patterns in a set of systems.

An advantage of the intentional definition is that it facilitates consistent identification of occurrences in the gold standard at markup time (independent of the software system used). It also makes it easier to discuss the allowed implementation variants.

However, an intentional specification requires a specification *language* that allows specifying pattern occurrences uniquely. Such a specification language may favor some approaches to detect the specified design pattern variants. As an indicator, we see that early approaches to design pattern occurrence detection used only structural information [25], maybe due to the unique specification of structural aspects in [1]. An intentional definition may also exclude some perfectly acceptable implementation variants of a pattern. For instance, the specification by example used for the behavioral aspects of pattern occurrences in [1] is therefore not sufficient.

An advantage of an extensional definition is that detection results can be compared to the gold standard automatically. Hence, we propose an extensional definition of allowed pattern variants since it is possible to enumerate all variants occurring in (finite) systems. Complementarily, an intentional definition of allowed pattern variants, well-defined and consistent with the occurrences listed extensionally, ought to be developed and used to discuss whether particular code structures are pattern occurrences or not. This is similar to the (extensional) markup in treebanks and the (intentional) rules for creating such markup as used in natural language processing; see Section 3.1.

5 Problem 2: Systems

5.1 The Problem

Approaches to pattern detection are evaluated by detecting pattern occurrences in some software systems. Since systems with different characteristics (e.g., size, number of pattern occurrences, programming language) are used, it is hard to compare the accuracy of the approaches.

On average, over many systems, we may assume that the number of pattern occurrences is proportional to the system size. However, the number of pattern occurrence *candidates* grows exponentially with system size, assuming that the number of entities suitable for each role of the pattern grows linearly with system size.[6] Pattern occurrence detection can restrict the number of candidates immediately due to the constraining relation between the entities in different roles. But the number of other entities that a given entity is related to is crucial for this restriction. In good object-oriented designs, we may assume that the number of classes reachable via "calls" and "aggregates" relations does not grow with the system size. This is obviously not the case for transitive relations like "is reachable from" or "is a descendant of." Therefore, especially for large pattern occurrence types and pattern variants without distinct and unique static footprints, the number of real pattern occurrences often grows significantly slower with system size than the number of occurrence *candidates*. This leads, for the same pattern and detection approach, to a lower precision in large systems than in small systems.

4. On the other hand, there are also some patterns that are seldom used (e.g., Command, Facade, and Flyweight). These patterns are may be harder to detect, and therefore, interesting to include in a benchmark.

5. Even though classified as a pattern requiring behavioral detection by Shi and Olsson [32], the Singleton has a rather distinct and unique static structure, cf., also the results of our static detection in Fig. 1.

6. It is the number of combinations of system entities over (suitable) pattern roles.

TABLE 2
Software Systems Used in Evaluations

	AWT (Java)	ET++ (C++)	**galib** (C++)	groff (C++)	**JHotDraw** (Java)	Jikes (C++)	JRefactory (Java)	JUnit (Java)	Juzzle (Java)	**LEDA** (C++)	**libg++** (C++)	**Mec** (C++)	QuickUML (Java)	**socket** (C++)	StarOffice Calc (C++)	StarOffice Writer (C++)	**Swing** (Java)	zApp (C++)	Other
Brown [27]																			x
Krämer, Prechelt [25]										x								x	x
Antoniol et al. [22]			x	x						x	x	x		x					x
Keller et al. [28]		x																	x
Niere et al. [15]	x																		
Heuzeroth et al. [23]																	x		x
Balanyi et al. [29]						x				x					x	x			
Philippow et al. [30]																			x
Pettersson [31]																	x		
Vokac [3]																			x
Costagliola et al. [21]			x								x	x		x					
Kaczor et al. [9]					x				x				x						
Shi and Olsson [32]	x				x												x		x
Tsantalis et al. [33]					x		x	x											

Systems used in at least two evaluations are in bold face.

5.2 Survey

Table 2 lists the systems used in the evaluations in the papers included in our survey.[7] Besides the explicitly listed systems, there are even other systems used. These systems are not publicly available and often include the evaluated detection tool itself, student projects, and industrial systems with nondisclosure agreements.

One intrinsic obstacle to the comparison of detection approaches is that they are applied to software systems written in different programming languages. So far, the impact of the programming language on the difficulty of pattern detection has not been researched. Nevertheless, one can assume that it is harder for dynamically or weakly typed languages (like Perl, Python, PHP, and C) as the relations between entities—restricting the candidate sets—can only be statically approximated.

However, even within a programming language, there is little overlap in the software systems used in evaluations. For Java, only Swing, JHotDraw, and AWT are used more than once. For C++, the LEDA library is most widely used (although in different versions).

In addition to the different programming languages, the software systems used in evaluations also differ in their sizes. Fig. 2 lists the number of classes of the systems used in the evaluations. The systems range from 5 to 6,729 classes/interfaces (the largest being StarOffice Writer). The median of the largest system in each evaluation is 502 classes/interfaces.

5.3 Experimental Support

We compare the detection of the Singleton and Observer patterns in Swing and SWT. We keep the pattern fixed and compare the results when the system is exchanged. Swing consists of 3,104 classes and interfaces and 25,777 methods. In contrast, the SWT library consists of 568 classes and interfaces and 5,726 methods. There is one pattern and system pair to compare for each occurrence type.

The measurements are available in Fig. 1. For all pairs, precision is higher in the smaller system (SWT) than the larger one (Swing). Furthermore, the difference can be very large. For instance, Strict Observers (4-tuple) in SWT are found with $P = 40\%$ but only with $P = 3.5\%$ in Swing. Relaxed Observers (2-tuple) in SWT are found with $P = 98.6\%$ but only with $P = 39\%$ in Swing.

The recall is larger in the smaller system in five out of the eight pairs. For instance, the Singleton detection in SWT has a recall of 100 percent (2-tuple), while the detection in Swing only has a recall of 59 percent (2-tuple). The exceptions are the detection of Strict Observers in SWT and Swing (all three pairs), where the recall is very low for SWT but much higher in Swing. The reason for the low recall in SWT is that we, unlike for Swing, have not performed detection with any application using the library (i.e., few concrete observers). Hence, we expect many false negatives. Even so, there are a few concrete observers in SWT itself, e.g., the `TypedListener` (for the subjects `Display` and `Widget`). Most of the concrete observers are anonymous classes, e.g., created inside `AnimatedProgress` or `PopupList`.

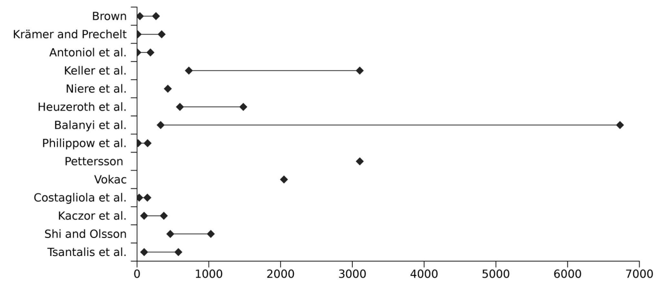

Fig. 2. The size interval of the software system used in evaluations (expressed in number of classes and/or interfaces). In Niere et al., the detection is evaluated on a subset of AWT. The whole AWT contains 429 classes but only 8,700 of 114,431 lines of code are included in the evaluation. For Shi and Olsson, it is not explicit what packages of Swing that are analyzed.

5.4 Proposed Solution

In the area of natural language parsing, the characteristics of texts matter in evaluations. Characteristics include the sentence length and the text type, e.g., scientific text versus newspaper articles versus personal correspondence. For instance, parsing accuracy is sensitive to the size of the entities being analyzed in the sense that longer sentences tend to have lower accuracy than shorter sentences. Hence, it can be misleading to compare two test sets, where the average sentence lengths differ.

Similarly, the characteristics of the software systems used in pattern detection evaluations can have an impact on the accuracy measurements. Hence, it is important to agree on a set of software systems for evaluation. The set of software systems used in a benchmark should fulfill the following criteria:

- systems in different languages (it is not expected that all detection tools/approaches are able to analyze all systems),

7. AWT is the Abstract Window Toolkit, Java's original widget toolkit. ET++ is a C++ class library. galib is a C++ genetic algorithm library to solve optimization problems. groff is the GNU version of the troff utility. JHotDraw is a framework for technical and structured graphics. Jikes is a Java compiler. JRefactory is a refactoring tool for Java. JUnit is a unit testing framework for Java. Juzzle is a puzzle game. LEDA is the Library of Efficient Data Types and Algorithms. libg++ is a GNU C++ library. Mec is a trace-and-reply program. QuickUML is a UML class-diagram editor. socket is a library for interprocess communication. StarOffice Calc is the spreadsheet in StarOffice. StarOffice Writer is the word processor in StarOffice. Swing is a newer GUI for Java than AWT. zApp is a class library.

- different characteristics of the systems including system sizes, e.g., ranging from some hundred to at least a few thousand classes,
- compatibility with the selected set of design patterns in the benchmark, i.e., benchmark systems ought to contain a number of known occurrences of the benchmark patterns.

A step in this direction can be seen in Costagliola et al. [21], where the authors select some of the systems previously used by Antoniol et al. [22]. The most commonly used systems in Java and C++ are good candidates to be included in a benchmark. The systems used in at least two papers are shown in bold face letters in Table 2. For Java, we propose using the JHotDraw application v5.1 and the Swing library v1.4.2. Instead of using AWT, which is a subset of Swing and therefore has similar characteristics, we suggest selecting SWT v3.1.0. It is another GUI library with approximately the same size as AWT. For C++, one strong candidate for a benchmark is the LEDA library used by at least three research groups.

For SWT, all files in all packages could be included in the benchmark. For Swing, we suggest selecting all classes from the `com.sun.java.swing.*`, `javax.swing.*`, `java.awt.*`, and `sun.awt.*` packages. We also suggest including the *SwingSet2* example program to get more concrete observers into the system. The SwingSet2 application is an application that tests many features of the Swing API. The number of classes and interfaces is about 250, 550, and 3,100, for JHotDraw, SWT, and Swing, respectively.

Regarding compatibility with the selected patterns, this requires the presence of known pattern occurrences for each pattern and system pair. This is not the case yet but will be further discussed in the following section.

6 Problem 3: Gold Standard

6.1 The Problem

Well annotated gold standards for pairs of patterns and applications are essential for comparable accuracy metrics. A fundamental problem is that such gold standards for design pattern detection are not yet available and take much time to construct correctly (no garbage, i.e., false positives) and completely (no missing patterns occurrences, i.e., false negatives).

For large systems, gold standards are typically approximated, which may affect the correctness of accuracy evaluation. Compared to a correct gold standard, both under- and overapproximations are possible. Underapproximations exclude real pattern occurrences. Overapproximations include pattern occurrences that do not represent a design pattern. However, neither under nor overapproximations allow establishing correct lower or upper bounds on accuracy.

For small systems, exact gold standards can be constructed manually. However, small gold standards cause uncertainty about the reliability of the accuracy metrics.

6.2 Survey

Many approaches only verify if the *retrieved* pattern occurrences are true occurrences of the particular design pattern (i.e., they only measure precision). This also includes the conservative approaches with a recall of 100 percent (by definition of conservative). We will return to this in the next section.

Other papers are explicit about the problems of building correct and complete gold standards. For instance, Costagliola et al. [21] mention problems of building a gold standard even for a system with 144 classes (they build gold standards only for the smaller systems).

Several methods are used to validate the correctness of occurrences (retrieved or not), for instance, manual inspection of sources [15], [21], [32], [33], manual inspection of available documentation [25], [32], [33], application knowledge [30], and naming conventions [23], [25].

Various research groups do not always classify the same pattern occurrences as correct. For instance, Krämer and Prechelt [25] find 10 real Bridge occurrences in LEDA 3.0, whereas Antoniol et al. [22] do not find any real Bridge at all in LEDA 3.4. Even though a possible explanation for this result is a removal of Bridges between versions 3.0 and 3.4, it is more likely that the two groups applied different constraints. For example, Antoniol et al. may not have included exclusively necessary constraints, or the gold standard could have been overapproximated by Krämer and Prechelt. Another example is the results of Shi and Olsson [32], [45] on Swing compared to our own results [24], [43], [46]: We consider the (JMenu, addMenuListener, removeMenuListener, MenuListener) as an Observer occurrence in Swing, but no such occurrence is reported by Shi and Olsson.

An underapproximated gold standard can be constructed by using the pattern detection tool itself and relaxing some constraints of the evaluated pattern specifications. This gives an occurrence set that is a superset of the original detection result, which is then used to count false negatives of the original pattern specification. This approach is used by Krämer and Prechelt [25]. In their case, the relaxation revealed no false negatives at all. This is reported as a recall of 100 percent. However, it is possible that the relaxed constraints are still too restrictive. Hence, the result may still have false negatives, and thus, less than full recall. Hence, a (possibly) underapproximated gold standard is used. Relying on application knowledge and naming conventions are other possibilities that may lead to underapproximated gold standards.

Overapproximations are also used. For instance, Pettersson guarantees that a subset of the roles of the pattern occurrences is correct in the gold standard, but there may be garbage in some roles, leading to many tuples that do not really represent any real pattern occurrence [31].

In many cases, the gold standards are small. Even when only precision is measured, the retrieved lists of patterns are often small. In Krämer and Prechelt [25], e.g., the maximum number of true positives for the Bridge in any evaluated system—and also the number of occurrences in the gold standard due to its construction—is 10 (for LEDA 3.0). There are no true positives for the Composite, Decorator, and Proxy patterns in any of the four software systems. In total, there are 16 of 20 pattern and system pairs without any true positive at all. In Niere et al. [15], the gold standards also contain few occurrences. The Strategy has only two occurrences, the Composite and Bridge have only

one occurrence each. In the evaluation performed by Costagliola et al. [21], the number of real occurrences is in most cases only 1.

6.3 Support

In general, ad hoc gold standard approximations cannot be used to establish lower or upper bounds on the actual accuracy. Assume that we have a correct gold standard with m occurrences of a pattern. We exclude x occurrences to create a strictly underapproximated gold standard of size $\check{m} = m - x$. Let t_p, f_p, and f_n denote the number of true positives, false positives, and false negatives when detection result is compared to the correct gold standard. Analogously, let $\check{t_p}$, $\check{f_p}$, and $\check{f_n}$ denote the number of true positives, false positives, and false negatives when compared to the underapproximated gold standard.

Suppose that all x occurrences are found by a pattern detector tool. Since these x occurrences are not classified as correct in the underapproximated gold standard (but correct in the real gold standard), we have $\check{t_p} = t_p - x$ and $\check{f_p} = f_p + x$. The number of actual occurrences in the gold standard that are not found (i.e., false negatives) will still be the same when classified using the underapproximated gold standard, i.e., $\check{f_n} = f_n$. We obtain

$$\begin{aligned}\check{R} &= \check{t_p}/(\check{t_p} + \check{f_n}) = 1 - \check{f_n}/(\check{t_p} + \check{f_n}) = 1 - f_n/(\check{t_p} + f_n)\\ &\leq 1 - f_n/(t_p + f_n) = R \text{ and } \check{P} = \check{t_p}/(\check{t_p} + \check{f_p})\\ &= (t_p - x)/(t_p - x + f_p + x) = t_p/(t_p + f_p) - x/(t_p + f_p)\\ &= P - x/(t_p + f_p) \leq P.\end{aligned}$$

Hence, both recall and precision calculated using the underapproximated gold standard are lower than using a more correct gold standard.

Instead, suppose that none of the x excluded occurrences is found by the pattern detector tool. Then, no found occurrence is reclassified from true positive to false positive. Hence, $\check{t_p} = t_p$ and $\check{f_p} = f_p$. A false negative is an occurrence in the gold standard not found by detection. Since the underapproximated gold standard contains x less correct occurrences and none of them are found, the false negatives are decreased accordingly, i.e., $\check{f_n} = f_n - x$. We obtain $\check{R} = \check{t_p}/(\check{t_p} + \check{f_n}) = t_p/(t_p + \check{f_n}) \geq t_p/(t_p + f_n) = R$ and $\check{P} = \check{t_p}/(\check{t_p} + \check{f_p}) = t_p/(t_p + f_p) = P$. Hence, the recall calculated using the underapproximated gold standard is higher than using a more correct gold standard. The precision is unaffected under the assumption that the detection method and query are unaffected by the contents of the gold standard. However, precision can also increase using underapproximated gold standards. This may be accomplished since it is easier to target the detection toward a smaller set of correct pattern occurrences (and possible pattern variants), i.e., use a stronger structure than otherwise possible. Together, this shows that accuracy, calculated using underapproximations of the gold standard, cannot establish lower or upper bounds on the actual accuracy.

The situation is not better for overapproximations, which leads to the dual result. We include x extra occurrences into the correct gold standard to create a strict overapproximation. A pattern detection tool detecting all these garbage occurrences will have higher recall and precision using the overapproximated gold standard, compared to using the more correct gold standard. On the other hand, a pattern detection tool finding none of these extra occurrences (i.e., a more correct result) will have lower recall using the overapproximated gold standard compared to using the more correct gold standard. The precision is unaffected under the assumption that the detection method and query are unaffected by the contents of the gold standard. However, precision is easily decreased using overapproximated gold standards compared to more correct gold standards since it is harder to target the detection toward a larger set of pattern occurrences, i.e., use a weaker structure than otherwise possible.

Hence, neither under nor overapproximations can establish a lower or an upper bound compared to the actual accuracy.

6.4 Proposed Solution

The construction of a gold standard that is complete and correct for a large software system is inevitable even if it is an exhaustive semi-manual process. As we have seen, information about pattern occurrences can be obtained by several methods, including inspection of design documents and source code, knowledge of naming conventions, and other application knowledge. However, complete design information about design pattern occurrences in software systems is not always available [22], [23]. Furthermore, a single research group cannot afford to manually screen large software systems to find all cases where a class, interface, or method participates in a design pattern implementation, at least not for a large number of systems, which is needed to show generality of a particular detection method. The use of naming conventions is also a risk, since pattern occurrences not obeying the naming conventions will be excluded (i.e., leading to underapproximated gold standards). Finally, application knowledge relies on the developers to be aware of, and remember, all design pattern occurrences. As a result, the gold standards used so far use too small software systems or approximate the real pattern occurrences.

However, it is important to distinguish between ad hoc approximations and controlled underapproximations such as the pooling method used at the Text Retrieval Conferences (TREC). A controlled underapproximation allows comparison based on a set of pattern occurrences that the community agrees on being correct.

The initial gold standard containing the correct pattern occurrences can be constructed using a pooling process, similar to the one developed by the Text Retrieval Conferences, i.e., by combining the detection results from different approaches on the same pattern and system pair. For instance, both Kaczor et al. [9] and Tsantalis et al. [33] perform evaluation of the Composite in JHotDraw. Both Heuzeroth et al. [23] and Pettersson and Löwe [24], [31] perform evaluation of the Observer in Swing. Tsantalis et al. [33] perform evaluation of the Adapter, Composite, Template Method, Decorator, and Observer in JHotDraw. Kaczor et al. [9] perform evaluation of the Composite in JHotDraw. Heuzeroth et al. [23] perform evaluation of the Composite and Observer in Swing, and Pettersson and Löwe [24], [31] use Singleton and Observer in Swing and SWT.

TABLE 3
Metrics Used in Evaluations

	Precision	Recall	None given
Brown [27]	x		
Krämer and Prechelt [25]	avg	1	
Antoniol et al. [22]	x	1	
Keller et al. [28]			x
Niere et al. [15]	avg	avg	
Heuzeroth et al. [23]	x	~[8]	
Balanyi et al. [29]	x		
Philippow et al. [30]	x	x	
Pettersson [31]	x	1	
Vokac [3]	x	x	
Costagliola et al. [21]	x	~[9]	
Kaczor et al. [9]			x
Shi and Olsson [32]			x
Tsantalis et al. [33]	x	x	

1: (recall claimed to be) always 100 percent, "avg": average over several patterns.
8 Heuzeroth et al. assess recall only for one pattern (Observer).
9 Costagliola et al. measure recall only for the smaller systems.

Research groups should report real pattern occurrences for the selected design patterns and software systems to a publicly available moderated database like the *Design Pattern Detection Evaluation Suite*(DPDES) [47] or the infrastructure of Fülöp et al. [44]. The first revision of DPDES contains manually verified occurrences of the Observer and Singleton patterns for Swing and SWT. Over time, the gold standards will become more and more complete (research groups adding occurrences) and correct (errors can easily be found and corrected since all information is explicitly available). As discussed before, the selection of design patterns and software systems can also grow and change using the same process. Additional pattern types may also be added, e.g., antipatterns.

A natural continuation of this effort is to arrange pattern detection competitions (or shared tasks) inspired by the natural language parsing community [6]. Another continuation is to use statistics to estimate recall on large systems, which has been performed for small occurrence types [3]. The presence of gold standards can be of benefit to the pattern detection community as well as other areas relying on accurate knowledge of design patterns in systems (e.g., teaching design patterns).

7 Problem 4: Precision and Recall

7.1 The Problem

Evaluating accuracy without using *both* precision and recall is problematic since there is an intrinsic trade-off between high precision and high recall. By using a very specialized detection, we retrieve a *small* list of (mostly) correct pattern occurrences but exclude many real occurrences, i.e., get high precision and low recall. On the other hand, no constraints at all give 100 percent recall but nearly 0 percent precision since for each role of the pattern, all classes, interfaces, or methods will match. Very general detections give similar results. Hence, precision and recall must be assessed in combination, not individually. An integrated metric would be a valuable complement to enable easier comparison.

7.2 Survey

Table 3 shows the metrics used in evaluations in the community. About half of the evaluations in our survey only measure precision (and not recall) or do not present any of the metrics (i.e., only report the number of found occurrences). In some cases when both precision and recall are used, the results are only present for a subset of the patterns and systems that can be detected (e.g., [30]).

Several approaches calculate an average precision, e.g., by summing false positives/true positives for a set of patterns, rather than calculating precision per pattern and system pair. Average precision is not informative enough by itself since, for different patterns, it is differently hard to detect occurrences. Additionally, the number of real instances may differ for the patterns; there are even examples (e.g., [25]) where several evaluated patterns do not have a correct occurrence at all in some of the systems. All of this is hidden when using average precision.

7.3 Experimental Support

Using the same detection results as before in Section 5 (from detecting Singleton and Observer in Swing and SWT), it is evident that both precision and recall should be used.

The measurements are available in Fig. 1. For instance, the Strict Observer detection in SWT measured using (s, o) occurrence types gives a precision of 95.2 percent and a recall of 27 percent. Presenting only the precision of this pair is misleading. Presenting only recall is also questionable. For instance, the Relaxed Observer in Swing measured using (s, a, d, o) occurrence types gives a precision of 3.5 percent and a recall of 83 percent.

It should be noted that we have not used pattern specifications that deliberately aim for only high recall or high precision. We aim for a trade-off between precision and recall, with a slight bias toward higher recall.

7.4 Proposed Solution

Both precision and recall measurements should be assessed simultaneously for each pattern and system pair. To still keep the approaches comparable, precision and recall need to be integrated in a new common metrics.

One naive solution is to define such a metrics as the arithmetic mean of P and R, i.e., $(P + R)/2$. However, such a metric is still questionable: A precision of 0 percent with a recall of 100 percent—achievable by listing all combinations of classes and methods of the system in all patterns roles—still gives the mean of 50 percent, which could be interpreted as fair. Obviously, such a detection tool is quite useless.

A standard solution is to use the weighted harmonic mean of P and R (weighted F-Score). The weighted F-Score F_w, $w \in \mathbb{R}$, is defined as

$$F_w = \frac{(1 + w^2)PR}{w^2P + R}.$$

The traditional F-Score is obtained for $w = 1$, i.e., equally weights precision and recall. By letting w approach ∞ (0), the weighted F-Score approaches R (P). One important aspect of the F-Score is that higher F-Score is obtained if both precision and recall are reasonable high, compared to tools with only one of P or R high (in contrast to the arithmetic mean).

For pattern detection applications involving human clients, it is reasonable to consider recall more important than precision, i.e., one would tolerate checking a list of

TABLE 4
Pattern Occurrence Types (Classes, Interfaces, Methods, None Given) Used in Evaluations

	Classes/ Interfaces	Methods	None given
Brown [27]			x
Krämer and Prechelt [25]	x		
Antoniol et al. [22]			x
Keller et al. [28]			x
Niere et al. [15]	x		
Heuzeroth et al. [23]	x	x	
Balanyi et al. [29]	x		
Philippow et al. [30]			x
Pettersson [31]	x	x	
Vokac [3]	x		
Costagliola et al. [21]			x
Kaczor et al. [9]	x		
Shi and Olsson [32]	x		
Tsantalis et al. [33]			x

None given: Type not given explicitly nor derivable from the text.

pattern candidates manually if one could be sure that the list contains all pattern occurrences. A ratio of one real occurrence out of 10 candidates, i.e., $P = 10\%$ and $R = 100\%$, could be assumed as fair, i.e., such a detector should get a score of 50 percent. To achieve this, we suggest $w = 2\sqrt{2} \approx 2.8$.

With this weight, a precision of 90 percent with a recall of 10 percent will only give an F-Score of 11 percent. In contrast, a precision of 10 percent with a recall of 90 percent will give an F-Score of 48 percent. A precision of 60 percent and a recall of 95 percent give an F-Score of 89 percent. We propose to use this weighted F-Score metric as a complement to precision and recall.

8 Problem 5: Pattern Occurrence Type

8.1 The Problem

We use different occurrence types to represent design pattern occurrences (i.e., what roles of the pattern are represented in the list of matching occurrences). Smaller occurrence types contain less information and a user needs to manually find the missing parts. In general, precision and recall are affected, and as experiments show, the precision usually decreases with increased pattern occurrence size.

8.2 Survey

Table 4 shows the pattern occurrence types used in the evaluations of the survey. It is common to many papers that the pattern occurrence type is not exactly defined in the descriptions of the experimental setup.

Even when the occurrence type is given (either explicitly or derivable from the text), there is no standard occurrence type for the patterns. Mostly, a tuple of classes/interfaces closely related to the roles described in the GoF book [1] is used. The differences in the occurrence types selected can still be very large. For instance, the pattern occurrence type can contain all roles—classes, interfaces, and methods—of the design pattern mentioned in the GoF descriptions [31]. At the other end of the spectrum, a single role, e.g., the subject in the Observer pattern, can determine a pattern occurrence type [3].

Kaczor et al. [9] as well as Vokac [3] address the problem of using different types of pattern occurrences. For instance, Kaczor et al. exemplify that the 22 occurrences of the Composite pattern found in their setup can also be seen as only two occurrences if the leaf role of the Composite pattern is omitted.

8.3 Experimental Support

The values for precision and recall are given in Fig. 1. Obviously, the results vary significantly with the occurrence types. For instance, the precision for relaxed Observer detection calculated using pattern occurrence type (s, a, d, o) is much lower than calculated using the (s) type. The observation is confirmed by the Singleton pattern detection in Swing, first calculated using the 2-tuple occurrence type (s, g), then the smaller type (s).

8.4 Proposed Solution

The occurrence types assessed need to be defined to make the results comparable. Larger tuple types are preferred by both automated and human client applications. But usually the precision decreases significantly with growing pattern occurrence size; the recall is less affected.

Therefore, we propose to use a set of relevant pattern occurrence types as a complement for the full types (cf., the PARSEVAL metrics discussed in Section 2). This is appropriate since a subset of entities of a pattern's occurrence can be meaningful by themselves, e.g., the (Subject, Observer) pair for the Observer pattern. Such a small occurrence may be enough for some aspects of program comprehension since the remainder of the pattern roles can be traced manually by the maintainers. A too low precision (cf., the (s, a, d, o)-occurrences of the Observer patterns in Swing) would deteriorate manual tracing even if larger tuple types may make it easier to rule out false positives. Besides, a gold standard is easier to construct for smaller pattern occurrence types.

In conclusion, it should be made explicit what occurrence types are considered when comparing the detection results to the Gold Standard. For the GoF patterns [1] in the benchmark, we propose to use the following pattern occurrence types: *Adapter*: (Adapter, Adaptee), *Bridge*: (Abstraction, RefinedAbstraction, Implementer, Concrete Implementer), *Composite*: (Composite, Component, Leaf), *Singleton*: (Singleton), *Template Method*: (Abstract Class, Concrete Class), *Decorator*: (Decorator, Concrete Decorator, Component, Concrete Component), and *Observer*: (Subject, Observer).

9 Partial Match Metric

9.1 Motivation and Definition

Currently, the community only uses exact match, where a retrieved pattern occurrence must *exactly match an occurrence of the gold standard* for *all* roles of the occurrence type to contribute to the precision, recall, and F-Score figures. However, if all roles of a retrieved pattern occurrence except a few are correct, the result is better for program comprehension than if all roles are falsely identified. For large occurrence types, it is likely that a few components are incorrect.

The basic idea (inspired by the PARSEVAL metrics [38], [39]) for the partial match metrics is to count how many of the roles of a pattern occurrence were correctly identified in the correct place. For instance, if the detector finds a

Composite pattern with the Composite and Component roles correct, but with an incorrect Leaf, the partial match metrics will still give points for the two correct roles.

Even if some aspects of a partial match can be realized by using smaller pattern occurrence types (suboccurrences) and exact match, there is one significant difference. When using a predefined suboccurrence and exact match, you still need to find that particular predefined set of components (which is what you want if exactly those components are meaningful). With partial match, one can find out if some components are correct, which is appropriate if there is no predefined set of meaningful components. We think that partial match can be a helpful complement for program comprehension since a developer can deduce the missing roles (if they are not too many), even if a missing role is part of the core of the pattern.

We use three counts, Std, Sys, and Cor [39]. The count Std is the number of components of the gold standard, i.e., the number of occurrences in the gold standard times the occurrence size. The count Sys is the number of components of the retrieved occurrences, i.e., the number of retrieved occurrences times the occurrence size. The count Cor is the number of correct components, calculated as follows: For each component of the tuples, we count (including duplicates) the number of correctly identified elements in the retrieved occurrences for this particular component (i.e., position in the tuple). This number is not allowed to be larger than the number of correct elements of the corresponding component of the gold standard. All of these numbers are added into the final count Cor. The partial recall is defined as $\tilde{R} = Cor/Std$ and the partial precision is defined as $\tilde{P} = Cor/Sys$.

As a concrete example, consider the following gold standard C and retrieved occurrences A:

$$C = \{(a,b,c),(a,b,d),(a,b,e),(f,b,c),(f,g,d)\},$$
$$A = \{(a,b,c),(a,b,d),(f,g,a),(f,g,b),(f,a,d),$$
$$(f,g,c),(c,b,d),(e,g,d),(b,c,a),(b,d,a)\}.$$

The number of components of the gold standard is $Std = 5 \cdot 3 = 15$ and the number of components of the retrieved occurrences is $Sys = 10 \cdot 3 = 30$. To determine the number of correct components Cor, we count the number of correctly identified components. The component a is found at position 1 of the retrieved tuple in two cases. Similarly, f is found at position 1 in four cases, but only two may be correctly identified since the gold standard contains only two fs at position 1. For position 2, we correctly identify b in three places, g in one place (although four are retrieved). Continuing in this manner gives $Cor = 2+2+3+1+2+2 = 12$, and therefore, $\tilde{R} = 12/15 = 80\%$ and $\tilde{P} = 12/30 = 40\%$.

If we instead calculated recall and precision using an exact match, we obtain $R = 2/5 = 40\%$ and $P = 2/10 = 20\%$. Hence, the partial match gives higher values expressing that some occurrences were partially accurate. Complementing exact match with partial match gives the possibility of a more fine-grained comparison.

Consider retrieved occurrences A' (using another detection approach):

$$A' = \{(a,b,c),(a,b,d)\}.$$

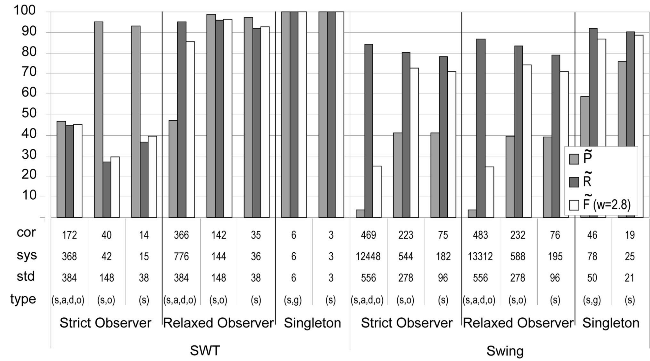

Fig. 3. Precision, Recall, and F-score of different patterns and tuple types scoring partial matches. std: the number of components in the gold standard, sys: number of components of the retrieved occurrences, cor: number of correctly retrieved components. For the 1-tuples, the scores are the same as for the exact matches.

This system would get $Sys = 2 \cdot 3 = 6$, $Cor = 2+2+1+1 = 6$, and therefore, $\tilde{R} = 6/15 = 40\%$ and $\tilde{P} = 6/6 = 100\%$, as if we calculated recall and precision using an exact match: $R = 2/5 = 40\%$ and $P = 2/2 = 100\%$. In fact, the partial match metrics (precision, recall, and F-score) are, by definition, always larger than the respective exact match metrics, by the factor $cor/(i \cdot t_p)$ (derivable from the definitions). The factor is not a constant, but rather a measure of how many retrieved design pattern occurrences are partially correct.

9.2 Experiments on Exact versus Partial Match

The accuracy scores using exact and partial matches for the detection of Singletons and Observers in SWT and Swing are given in Figs. 1 and 3, respectively. Comparing the figures confirms that the partial match metrics are indeed always by the above factor larger than the respective exact match metrics.

In case of the Singleton detection in SWT, there is no difference between exact and partial match simply because we already have $F = 100\%$ using exact match. There is no difference in Swing either. In this case, the reason is that there are no occurrences at all that are partially correct.

The case where partial match differs most from exact match is the detection of Strict Observers in SWT using (s,a,d,o)-tuples. The accuracy metrics are increased with over 26 percent, which corresponds to $cor - i \cdot t_p = 172 - 4 \cdot 34 = 36$ additional components found (and correct) besides those counted in an exact match. Using only exact match, it is not possible to distinguish between a detector finding these additional components in the correct roles, and a detector that does not.

The difference between exact and partial matches tends to be larger for large pattern occurrence types. This is also verified by our experiments. The four cases where the difference is the largest are also the four largest tuples, i.e., the 4-tuples.

10 Conclusion

We have highlighted issues related to accuracy evaluation of design pattern occurrences detection. To make competing approaches more comparable, we propose *Solutions* for the five *Problems* observed and discussed in the paper.

Solution 1 (Design Patterns and Variants). A common set of patterns, with different characteristics, with well-defined implementation variants given by gold standards and preferably also intentional markup rules.

Solution 2 (Systems). A common set of systems with different characteristics (e.g., size).

Solution 3 (Gold Standard). A community effort to construct well-annotated gold standards for the proposed patterns and systems. The gold standards may be controlled underapproximations of sufficient size.

Solution 4 (Precision and Recall). Present precision, recall, as well as the weighted F-Score metric ($w \approx 2.8$).

Solution 5 (Pattern Occurrence Type). A common set of explicit definitions of the pattern occurrence types for each pattern (variant) in the Gold Standard.

Finally, we propose partial matches as a complement to the exact match, especially for those patterns in the benchmark that lead to coarse-grained comparison results when based on exact matches.

The initial benchmark suite is available at DPDES [47]. The proposed GoF patterns and their pattern occurrence types are: *Adapter*: (Adapter, Adaptee), *Bridge*: (Abstraction, Refined Abstraction, Implementor, Concrete Implementor), *Composite*: (Composite, Component, Leaf), *Singleton*: (Singleton), *Template Method*: (Abstract Class, Concrete Class), *Decorator*: (Decorator, Concrete Decorator, Component, Concrete Component), and *Observer*: (Subject, Observer). The software systems proposed for Java in the benchmark are JHotDraw v5.1, SWT v3.1.0, and Swing v1.4.2 plus the SwingSet2 application.

Future work includes the construction of the benchmark collection through the DPDES site as well as the development of methods to facilitate accuracy comparisons for detectors operating on different programming languages.

Appendix A

Experimental Setup

A.1 Pattern Specification and Detection Method

We use first-order logic to specify design pattern occurrences. It is powerful, at least to express the structural pattern aspects. We use CrocoPat [26] for querying the relational facts extracted from the program structure and static semantics.

A.2 Design Patterns and Variants

We have focused on the Singleton and Observer patterns, which are different in nature. The Singleton has a rather distinct and unique structure (e.g., footprints in an AST), making it relatively easy to detect even without considering its relatively complex behavioral aspects (e.g., lazy initialization of the Singleton variable). The Observer pattern has a less distinctive and unique structure and defines even more behavioral constraints. Thus, it may be harder to detect with as high accuracy using structural detection only.

Fig. 4 defines the first-order logic expression used to detect Singletons. Most constraints for the Singleton are unary, e.g., *IsClass* and *IsConstructor*. There are a few binary relations, e.g., a method should have the singleton as return type. With our program model, we differentiate between *returns* and *returnsArrayOf*, which means that returning an array of singletons is prohibited. Furthermore, there should be a static containment of the singleton class itself. When all constraints are combined, the structure is very distinctive. Even so, several implementation variants fit this specific structural footprint.

$$
\begin{aligned}
&Singleton(s,g) = \\
&\quad isClass(s) \wedge && (1)\\
&\quad isMethod(g) \wedge && (2)\\
&\quad \neg isAbstract(s) \wedge && (3)\\
&\quad isStatic(g) \wedge && (4)\\
&\quad containsMethod(s,g) \wedge && (5)\\
&\quad returns(g,s) \wedge && (6)\\
&\quad staticcontainment(s,s) \wedge && (7)\\
&\quad \neg\exists c(containsMethod(s,c) \wedge isConstructor(c) \wedge isPublic(c)) && (8)
\end{aligned}
$$

Fig. 4. First-order logic expression for singletons involving two participants: the singleton class (s) and the *getInstance* method (g).

The Observer pattern has several possible implementation variants, e.g., occurrences with and without a broker class. We distinguish a strict version and a relaxed version. The relaxed observer version is exactly as the strict, except that it does not know anything about concrete observers. The relaxed version is suitable for libraries, where the concrete observers are supplied by an application.

The first-order logic expression to detect the Observer pattern (complete variant) is depicted in Fig. 5. Most of the query in Fig. 5 consists of unary constraints that cut down the search space considerably compared to allowing each class, interface, or method in each possible role (see lines 1-23). Besides these unary constraints, there are several binary relations holding in many variants (see lines 24-31). For instance, the Subject should contain the *attach*, *detach*, and *notifyObservers* methods (see lines 25-27). The *attach* method should take the Observer to attach as a parameter (see line 29). The same holds for the *detach* method (see line 30). Also, the concrete *update* method should override the corresponding *update* method in the Observer, which is one of the more restrictive constraints used (see line 31).

A.3 Software Systems

We use SWT (3.1.0) and Swing (1.4.2), which are known to contain a number of Singleton and Observer occurrences. The sizes of the systems differ: Swing consists of 3,104 classes and interfaces, and 25,777 methods. In contrast, the SWT library consists of 568 classes and interfaces, and 5,726 methods.

For SWT, all files in all packages were analyzed. For Swing, we analyze all classes from `com.sun.java.swing.*`, `javax.swing.*`, `java.awt.*`, and sun.awt.*. For Swing, we also included the Sun SwingSet2 example program to get concrete observers into the application.

A.4 Precision and Recall

We will measure both precision and recall and compute a weighted F-score ($w = 2\sqrt{2} \approx 2.8$).

A.5 Gold Standard Construction

Since large manually validated gold standards are not available to date, we have generated our own gold standard

$$\begin{aligned}
&Observer(s, a, d, no, o, u, co, cu) = \\
&\quad isClass(s) \wedge && (1)\\
&\quad isClass(co) \wedge && (2)\\
&\quad isMethod(a) \wedge && (3)\\
&\quad isMethod(d) \wedge && (4)\\
&\quad isMethod(no) \wedge && (5)\\
&\quad isMethod(u) \wedge && (6)\\
&\quad isMethod(cu) \wedge && (7)\\
&\quad isInterface(o) \wedge && (8)\\
&\quad isPublic(a) \wedge && (9)\\
&\quad isPublic(d) \wedge && (10)\\
&\quad isPublic(u) \wedge && (11)\\
&\quad isPublic(cu) \wedge && (12)\\
&\quad \neg isAbstract(a) \wedge && (13)\\
&\quad \neg isAbstract(d) \wedge && (14)\\
&\quad \neg isAbstract(no) \wedge && (15)\\
&\quad isAbstract(u) \wedge && (16)\\
&\quad \neg isAbstract(cu) \wedge && (17)\\
&\quad \neg isAbstract(co) \wedge && (18)\\
&\quad \neg isConstructor(a) \wedge && (19)\\
&\quad \neg isConstructor(d) \wedge && (20)\\
&\quad \neg isConstructor(no) \wedge && (21)\\
&\quad \neg isConstructor(u) \wedge && (22)\\
&\quad \neg isConstructor(cu) \wedge && (23)\\
&\quad containsMethod(o, u) \wedge && (24)\\
&\quad containsMethod(s, a) \wedge && (25)\\
&\quad containsMethod(s, d) \wedge && (26)\\
&\quad containsMethod(s, no) \wedge && (27)\\
&\quad containsMethod(co, cu) \wedge && (28)\\
&\quad containsParam(a, o) \wedge && (29)\\
&\quad containsParam(d, o) \wedge && (30)\\
&\quad overridesMethod(cu, u) \wedge && (31)\\
&\quad s \neq co \wedge && (32)\\
&\quad a \neq d \wedge && (33)\\
&\quad a \neq no \wedge && (34)\\
&\quad d \neq no && (35)
\end{aligned}$$

Fig. 5. Static observer specification involving eight participants: the subject class (s), the *attach* method (a), the *detach* method (d), the *notify* method (no), the observer interface (o), the *update* method (u), the concrete observer class (co), and finally, the *concrete update* method (cu).

with our best effort. The construction relies on naming conventions, comments, and a set of plausible implementation variants to extract the gold standards. Manual inspection is used to verify the constructed sets.

The gold standard for the *Singleton pattern* was constructed as follows: To obtain a set of *getInstance* methods, we extracted all methods with both *get* and *instance* in the name (to find textbook implementations, but also variants such as *getSingleInstance*). Also, we extracted all methods with both *shared* and *instance* but without *set* in the name. This includes methods like *getSharedInstance* and *sharedInstance* but excludes *setSharedInstance*. This convention is used to some extent in Swing. The source code was manually inspected in the context of each found method to exclude non-Singleton occurrences. We also performed detection using plausible implementation variants to find patterns violating the naming conventions. These were also manually inspected.

For SWT, we found three occurrences using naming conventions. All three were correct according to comments in the source code. Furthermore, no additional Singleton exists according to the same kind of comment. The very same three patterns, and no others, were also found using various implementation variants. Hence, we kept this set as the gold standard for SWT.

For Swing, we found 42 occurrences using naming conventions. We excluded 21 of them. In most cases, the reason was that the occurrence was a Factory. By performing pattern detection with a set of plausible implementation variants, we added four Singleton occurrences with deviant naming.

The gold standard for the *Observer pattern* was constructed as follows: The *attach* methods in both Swing and SWT are called `addXListener`, where X is an observer type. The *detach* methods follow a similar pattern. Both the *attach* and *detach* methods must contain the observer as a parameter. This creates a list of (s, a, d, o) tuples. The list was manually inspected since other parameters than observer X could be passed to the *attach* and *detach* methods, which are "false" Observers. We removed all such tuples manually. We are thus confident that the 4-tuples (s, a, d, o) of the gold standard are correct. However, there may exist Observers pattern occurrences not obeying the naming conventions; they would not be included in the gold standard.

In SWT, there were 100 4-tuples originally. We removed four tuples having incorrect Observer types, e.g., integer types.

In Swing, there were 209 4-tuples originally. We removed 15 tuples having incorrect Observer types. We removed 53 additional tuples corresponding to three Observer classes declared outside the Swing packages, i.e., in `java.beans` and `java.util`. Furthermore, we removed two tuples where the subject was an anonymous inner class with an empty implementation. According to comments in the code, they are only used to support garbage collection. This results in a gold standard of 139 pattern occurrences.

A.6 Measured Pattern Occurrence Types

For the Singleton pattern, we use two pattern occurrence types: the 2-tuple Singleton class and *getInstance* method as well as the 1-tuple Singleton class. For the Observer pattern, we measure three occurrence types: (s, a, d, o), (s, o), and (s), where s, a, d, and o denote subject, attach method, detach method, and observer, respectively.

Acknowledgments

The authors thank the anonymous reviewers of the *IEEE Transactions on Software Engineering* for their valuable comments. They also thank the participants of the First International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE) for suggestions on the first version of the paper. This research is partially funded by the Swedish National Graduate School in Computer Science (CUGS) and the Swedish Agency for Innovation Systems (Vinnova), Project No. P22479-1 A.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[2] M. Meyer, "Pattern-Based Reengineering of Software Systems," *Proc. 13th Working Conf. Reverse Eng.*, pp. 305-306, Oct. 2006.

[3] M. Vokac, "An Efficient Tool for Recovering Design Patterns from C++ Code," *J. Object Technology*, vol. 5, no. 1, pp. 139-157, Jan./ Feb. 2006.

[4] K. Driesen and U. Hölzle, "The Direct Cost of Virtual Function Calls in C++," *Proc. Conf. Object Oriented Programming Systems Languages and Applications*, pp. 306-323, Oct. 1996.

[5] U.P. Schultz, J.L. Lawall, and C. Consel, "Specialization Patterns," *Proc. 15th Int'l Conf. Automated Software Eng.*, pp. 197-206, 2000.

[6] S. Buchholz and E. Marsi, "CoNLL-X Shared Task on Multilingual Dependency Parsing," *Proc. 10th Conf. Computational Natural Language Learning*, http://www.aclweb.org/anthology/ W/W06/W06-2920, pp. 149-164, June 2006.

[7] S.E. Sim, S. Easterbrook, and R.C. Holt, "Using Benchmarking to Advance Research: A Challenge to Software Engineering," *Proc. 25th Int'l Conf. Software Eng.*, pp. 74-83, 2003.

[8] Y.-G. Guéhéneuc and H. Albin-Amiot, "Recovering Binary Class Relationships: Putting Icing on the UML Cake," *Proc. Conf. Object Oriented Programming Systems Languages and Applications*, J.M. Vlissides and D.C. Schmidt, eds., pp. 301-314, 2004.

[9] O. Kaczor, Y.-G. Guéhéneuc, and S. Hamel, "Efficient Identification of Design Patterns with Bit-Vector Algorithm," *Proc. Conf. Software Maintenance and Reeng.*, Mar. 2006.

[10] Y.-G. Guéhéneuc, K. Mens, and R. Wuyts, "A Comparative Framework for Design Recovery Tools," *Proc. 10th Conf. Software Maintenance and Reeng.*, pp. 123-134, Mar. 2006.

[11] Y.-G. Guéhéneuc and G. Antoniol, "Demima: A Multi-Layered Approach for Design Pattern Identification," *IEEE Trans. Software Eng.*, vol. 34, no. 5, pp. 667-684, Sept./Oct. 2008.

[12] J. Dong, Y. Zhao, and T. Peng, "Architecture and Design Pattern Discovery Techniques—A Review," *Proc. Int'l Workshop System/ Software Architectures*, June 2007.

[13] T. Peng, J. Dong, and Y. Zhao, "Verifying Behavioral Correctness of Design Pattern Implementation," *Proc. Int'l Conf. Software Eng. and Knowledge Eng.*, pp. 454-459, 2008.

[14] J. Dong, Y. Sun, and Y. Zhao, "Design Pattern Detection by Template Matching," *Proc. ACM Symp. Applied Computing*, pp. 765-769, 2008.

[15] J. Niere, W. Schafer, J. Wadsack, L. Wendehals, and J. Welsh, "Towards Pattern-Based Design Recovery," *Proc. 24th Int'l Conf. Software Eng.*, pp. 338-348, 2002.

[16] J. Niere, J.P. Wedsack, and L. Wendehals, "Handling Large Search Space in Pattern-Based Reverse Engineering," *Proc. 11th IEEE Int'l Workshop Program Comprehension*, May 2003.

[17] L. Wendehals, "Improving Design Pattern Instance Recognition by Dynamic Analysis," *Proc. Int'l Conf. Software Eng. 2003 Workshop Dynamic Analysis*, May 2003.

[18] L. Wendehals, M. Meyer, and A. Elsner, "Selective Tracing of Java Programs," *Proc. Second Int'l Fujaba Days*, vol. tr-ri-04-253, pp. 7-10, Sept. 2004.

[19] L. Wendehals and A. Orso, "Recognizing Behavioral Patterns at Runtime Using Finite Automata," *Proc. Workshop Dynamic Analysis*, May 2006.

[20] G. Costagliola, A.D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design Pattern Recovery by Visual Language Parsing," *Proc. Conf. Software Maintenance and Reeng.*, Mar. 2005.

[21] G. Costagliola, A.D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Case Studies of Visual Language Based Design Patterns Recovery," *Proc. Conf. Software Maintenance and Reeng.*, Mar. 2006.

[22] G. Antoniol, R. Fiutem, and L. Christoforetti, "Design Pattern Recovery in Object-Oriented Software," *Proc. Sixth Int'l Workshop Program Comprehension*, pp. 153-160, June 1998.

[23] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic Design Pattern Detection," *Proc. 11th Int'l Workshop Program Comprehension*, May 2003.

[24] N. Pettersson and W. Löwe, "A Non-Conservative Approach to Software Pattern Detection," *Proc. 15th IEEE Int'l Conf. Program Comprehension*, K. Wong, E. Stroulia, and P. Tonella, eds., pp. 189-197, June 2007.

[25] C. Krämer and L. Prechelt, "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software," *Proc. Working Conf. Reverse Eng.*, pp. 208-215, Nov. 1996.

[26] D. Beyer, A. Noack, and C. Lewerentz, "Simple and Efficient Relational Querying of Software Structures," *Proc. 10th IEEE Working Conf. Reverse Eng.*, pp. 216-225, Nov. 2003.

[27] K. Brown, "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk," Technical Report TR-96-07, North Carolina State Univ., June 1996.

[28] R.K. Keller, R. Schauer, S. Robitaille, and P. Page, "Pattern-Based Reverse-Engineering of Design Components," *Proc. Int'l Conf. Software Eng.*, pp. 226-235, May 1999.

[29] Z. Balanyi and R. Ferenc, "Mining Design Patterns from C++ Code," *Proc. IEEE Int'l Conf. Software Maintenance*, Sept. 2003.

[30] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann, "An Approach for Reverse Engineering of Design Patterns," *Software and Systems Modeling*, vol. 4, no. 1, pp. 55-70, Feb. 2005.

[31] N. Pettersson, "Measuring Precision for Static and Dynamic Design Pattern Recognition as a Function of Coverage," *Proc. Int'l Conf. Software Eng. Workshop Dynamic Analysis*, May 2005.

[32] N. Shi and R.A. Olsson, "Reverse Engineering of Design Patterns from Java Source Code," *Proc. Int'l Conf. Automated Software Eng.*, Sept. 2006.

[33] 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, Nov. 2006.

[34] J. Seemann and J.W. von Gudenberg, "Pattern-Based Design Recovery of Java Software," *Proc. ACM SIGSOFT*, pp. 10-16, Nov. 1998.

[35] J.M. Smith and D. Stotts, "SPQR: Flexible Automated Design Pattern Extraction from Source Code," *Proc. IEEE Int'l Conf. Automated Software Eng.*, Oct. 2003.

[36] *Treebanks: Building and Using Parsed Corpora*, A. Abeillé, ed. Kluwer Academic Publishers, 2003.

[37] M.P. Marcus, B. Santorini, and M.A. Marcinkiewicz, "Building a Large Annotated Corpus of English: The Penn Treebank," *Computational Linguistics*, vol. 19, pp. 313-330, 1993.

[38] E. Black, S. Abney, D. Flickinger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, S. Roukos, B. Santorini, and T. Strzalkowski, "A Procedure for Quantitatively Comparing the Syntactic Coverage of English Grammars," *Proc. DARPA Speech and Natural Language Workshop*, pp. 306-311, 1991.

[39] R. Grishman, C. Macleod, and J. Sterling, "Evaluating Parsing Strategies Using Standardized Parse Files," *Proc. Third ACL Conf. Applied Natural Language Processing*, pp. 156-161, 1992.

[40] E.M. Vorhees, "The Philosophy of Information Retrieval Evaluation," *Proc. 12th Text Retrieval Conf.*, pp. 355-370, 2002.

[41] P. Ahlgren and L. Grönqvist, "Measuring Retrieval Effectiveness with Incomplete Relevance Data," *Proc. Conf. Current Research in Information Sciences and Technology*, 2006.

[42] L. Fülöp, T. Gyovai, and R. Ferenc, "Evaluating C++ Design Pattern Miner Tools," *Proc. Sixth IEEE Int'l Workshop Source Code Analysis and Manipulation*, Sept. 2006.

[43] N. Pettersson, W. Löwe, and J. Nivre, "On Evaluation of Accuracy in Pattern Detection," *Proc. First Int'l Workshop Design Pattern Detection for Reverse Eng.*, Oct. 2006.

[44] L.J. Fülöp, R. Ferenc, and T. Gyimothy, "Towards a Benchmark for Evaluating Design Pattern Miner Tools," *Proc. 12th European Conf. Software Maintenance and Reeng.*, pp. 143-152, Apr. 2008.

[45] "pinot," http://www.cs.ucdavis.edu/shini/research/pinot/, Jan. 2008.

[46] N. Pettersson, "Towards Pattern Detection in Software," licentiate thesis, School of Math. and Systems Eng., Växjö Univ., May 2006.

[47] N. Pettersson, "Design Pattern Detection Evaluation Suite (DPDES)," http://w3.msi.vxu.se/users/npe/DPDES, May 2009.

Niklas Pettersson received the master's degree from Växjö University, Sweden, in April 2002, and the licentiate degree from the Swedish National Graduate School in Computer Science (CUGS) in May 2006. He is currently working toward the PhD degree at CUGS. His research interests include program analysis, program comprehension, and software design. His primary occupation is software development.

Welf Löwe received the PhD degree from the University Karlsruhe, Germany, in 1996. He studied computer science at the Technical University Dresden, Germany, from 1987 to 1992. After a postdoctoral visit to the International Computer Science Institute, Berkeley, California, he returned to Karlsruhe as a senior researcher. He has been a professor at Växjö University, Sweden, and the head of the Software Technology Group since 2002. Since 2006, he has also been the chief executive officer of Applied Research in System Analysis (ARISA) AB.

Joakim Nivre received the PhD degrees in general linguistics and computer science from the University of Gothenburg and Växjö University, respectively. He has been a professor of computational linguistics at Uppsala University since 2008 and Växjö University since 2002. His research focuses on data-driven methods for natural language processing, in particular for syntactic and semantic analysis. He is one of the main developers of the transition-based approach to data-driven dependency parsing, described in his 2006 book *Inductive Dependency Parsing* and implemented in the MaltParser system. He is the secretary of the European Association for Computational Linguistics and deputy director of the Swedish Graduate School in Language Technology.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.