The Journal of Systems and Software 84 (2011) 559–572

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

BDTEX: A GQM-based Bayesian approach for the detection of antipatterns

Foutse Khomh[a,b,*], Stephane Vaucher[b], Yann-Gaël Guéhéneuc[a], Houari Sahraoui[b]

[a] Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
[b] GEODES Lab., DIRO, Université de Montréal, Canada

ARTICLE INFO

Article history:
Received 11 January 2010
Received in revised form 28 July 2010
Accepted 25 November 2010
Available online 17 December 2010

Keywords:
Code smells
Antipatterns
Detection

ABSTRACT

The presence of antipatterns can have a negative impact on the quality of a program. Consequently, their efficient detection has drawn the attention of both researchers and practitioners. However, most aspects of antipatterns are loosely specified because quality assessment is ultimately a human-centric process that requires contextual data. Consequently, there is always a degree of uncertainty on whether a class in a program is an antipattern or not. None of the existing automatic detection approaches handle the inherent uncertainty of the detection process. First, we present BDTEX (Bayesian Detection Expert), a Goal Question Metric (GQM) based approach to build Bayesian Belief Networks (BBNs) from the definitions of antipatterns. We discuss the advantages of BBNs over rule-based models and illustrate BDTEX on the Blob antipattern. Second, we validate BDTEX with three antipatterns: Blob, Functional Decomposition, and Spaghetti code, and two open-source programs: GanttProject v1.10.2 and Xerces v2.7.0. We also compare the results of BDTEX with those of another approach, DECOR, in terms of precision, recall, and utility. Finally, we also show the applicability of our approach in an industrial context using Eclipse JDT and JHotDraw and introduce a novel classification of antipatterns depending on the effort needed to map their definitions to automatic detection approaches.

1. Context and problem

Software quality is important because of the complexity and pervasiveness of software systems. Moreover, the current trend in outsourcing development and maintenance requires means to measure quality with great details. Object-oriented quality is adversely impacted by antipatterns (Brown et al., 1998); their early detection and correction would ease development and maintenance.

Antipatterns are "poor" solutions to recurring implementation and design problems that impede the maintenance and evolution of programs. They are described using a template which describe their general forms, their symptoms, their consequences, and some refactored solutions. The symptoms are often code smells (Fowler, 1999). Even though a class in a program can present all symptoms of a given antipattern, it is not necessarily an antipattern. Moreover, when discussing antipatterns, we do not exclude that, in a particular context, an antipattern could be the best way to implement or design a (part of a) program. For example, automatically generated parsers present many symptoms of Spaghetti Code, i.e., very large classes with very long and complex methods. Only a quality analyst can evaluate the impact of antipatterns on their program in their context.

All aspects of an antipattern are loosely specified because quality assessment is ultimately a human-centric process that requires contextual data. Consequently, there is always a degree of uncertainty on whether a class in a program is an antipattern or not. Therefore, detection results should be reported with the degree of uncertainty of the detection process. This uncertainty accounts for the loose definitions and the similarity of classes with the antipattern.

There exist many approaches to specify and detect antipatterns. Some of these approaches are manual (Travassos et al., 1999), others are based on rules (Marinescu, 2004). Manual detection approaches avoid the problem of uncertainty, but do not scale up to the inspection of large systems. To the best of our knowledge, none of the existing automatic approaches provide a way to deal with the uncertainty of the detection. They provide quality analysts with an unsorted set of candidates classes with no indication of which one(s) should be inspected first for confirmation and correction.

This paper builds on our previous work (Khomh et al., 2009) in which we illustrated the use of a Bayesian Belief Network (BBN) to specify the Blob antipattern and to detect its occurrences in programs. A Blob, also called God class (Riel, 1996), is a class that centralises functionality and has too many responsibilities. Brown et al. (1998) characterise its structure as a large controller class that depends on data stored in several surrounding data classes. Table 1

* Corresponding author at: Department of Electrical and Computer Engineering, Queen's University, Canada. Tel.: +1 613 533 6000x75542.
E-mail address: foutse.khomh@queensu.ca (F. Khomh).

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.11.921

Table 1
List of detected antipatterns.

The *Blob* (called also God class (Riel, 1996)) corresponds to a large controller class that depends on data stored in surrounded data classes. A large class declares many fields and methods with a low cohesion. A controller class monopolises most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes. We identify controller classes using suspicious names such as 'Process', 'Control', 'Manage', 'System', and so on. A data class contains only data and performs no processing on these data. It is composed of highly cohesive fields and accessors
The *Functional Decomposition* antipattern may occur if experienced procedural developers with little knowledge of object-orientation implement an object-oriented system. Brown describes this antipattern as "a 'main' routine that calls numerous subroutines". The Functional Decomposition design defect consists of a main class, i.e., a class with a procedural name, such as 'Compute' or 'Display', in which inheritance and polymorphism are scarcely used, that is associated with small classes, which declare many private fields and implement only few methods
The *Spaghetti Code* is an antipattern that is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring long methods with no parameters, and utilising global variables for processing. Names of classes and methods may suggest procedural programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms, polymorphism and inheritance

summarises its definition. The output of the BBN was a probability that a class is a Blob.

The BBN could handle uncertainty by evaluating the probability that an input, e.g., a large class, causes an observed output, e.g., a Blob. Consequently, such a BBN allowed quality analysts to prioritise the inspection of candidate classes. Furthermore, the Bayesian theory that underlies BBNs could be used to calibrate a BBN using past detection results by learning the relations between different inputs and their combined effects on the output. We could thus improve the performance of a BBN for a given context, e.g., organisation or type of program. A quality analyst can also encode her judgement into a BBN when data is unavailable.

Building on this previous work, this paper presents a Goal Question Metric (GQM) based approach, BDTEX (Bayesian Detection Expert), to systematically build BBNs to detect antipatterns from their definitions instead of an intermediate representation based on rules. It also provides an extensive discussion on the challenges, strengths, and weaknesses of a BBN-based approach with a systematic criteria-based comparison of the state-of-the-art. In addition to the Blob, we apply the approach to two additional antipatterns: the Functional Decomposition and the Spaghetti code, whose definitions are presented in Table 1, and to two extra systems, Eclipse JDT and JHotDraw, to show that BBNs perform well on "good" programs and scale up. This paper also shows that our approach provides BBNs whose results minimise the quality analysts' effort when compared to the state-of-the-art rule-based approach, DECOR (Moha et al., 2010a).

Section 2 presents a systematic criteria-based review of previous work. Section 3 recalls the basics of BBNs and describes the different steps of our GQM-based approach, BDTEX. Section 4 reports experiments on the use of BDTEX on two open source systems: GanttProject v1.10.2, Xerces v2.7.0 and three antipatterns: Blob, Functional Decomposition, and Spaghetti Code. It also compares our approach with DECOR in terms of precision, recall, and utility. Section 5 discusses the applicability of our approach in an industrial context using Eclipse JDT v3.1.2 and JHotDraw v5.3, and propose a classification of antipatterns into three types depending on the effort needed to map their definitions to automatic detection approaches in general, and BDTEX in particular. Section 6 concludes with future work.

2. Related work

In this section, we first recall the first work on antipatterns, then define criteria to compare previous work. We perform a comparison of previous work using these criteria and, finally, we discuss the benefits of BBNs.

2.1. The origin of antipatterns

The first book on "antipatterns" in object-oriented development was written by Webster (1995); his contribution covers conceptual, political, coding, and quality-assurance problems. Later Riel (1996), defined 61 heuristics characterising good object-oriented programming to assess software quality manually and improve design and implementation. Since then, antipatterns have been the center topic of many books. Fowler (1999) defined 22 code smells, symptoms of antipatterns, suggesting where developers should apply refactorings. Mantyla (2003) and Wake (2003) proposed classifications of code smells. Brown et al. (1998) described 40 antipatterns, including the well-known Blob, Functional Decomposition, and Spaghetti Code. These books provide in-depth views on heuristics, code smells, and antipatterns aimed at a wide academic and industrial audience. We build upon this previous work to propose an approach to detect antipatterns while taking into account their loose definitions and the inherent uncertainty of the detection process.

2.2. Issues with detection approaches

Dhambri et al. (2008) report six issues from their experience with industrial partners, which must be addressed to make effective an antipattern-detection process. These issues concern the loose definitions of antipatterns, which use code structures, developers/designers' intents, and evolution trends.

These issues also pertain to the effort required by the quality analysts to obtain, classify, and use the detected candidate antipatterns. On the one hand, a completely manual approach is not concerned by issues 1, 3, 4, and 6 but would require a great amount of effort because the quality analysts would need to assess manually each and every classes in a program every other classes, issues 2 and 5. On the other hand, (semi-)automated approaches face all these issues because they attempt to reduce the quality analysts' efforts by automating part of their decision process.

We now recall and discuss these issues before comparing previous work using them as criteria.

Issue 1: Difficulty of deciding when an antipattern candidate is actually a *real* antipattern. Antipatterns are detected by their symptoms, i.e., code smells. However, a class might exhibit all symptoms of an antipattern without being one of its instance. Therefore, human intervention is needed to validate detection results because the detection process is uncertain. Consequently, a detection approach should report the uncertainty of the candidate classes.

Issue 2: Uselessness of long lists of candidates. The objective of a detection process is to guide a manual review by a quality analyst. Too many false positives in the result set can lead to a rejection of the detection approach. Consequently, a detection approach should provide a list of its candidates ranked by confidence and/or severity.

Issue 3: Difficulty of taking into account the quality analysts' judgement. The quality analysts' background and contextual knowledge affect their understanding of what is a

"good" or a "poor" design, given a certain set of symptoms.

Issue 4: Necessity to take into account the context of the detection. Classes that may be considered "good" by many quality analysts in a given context may still violate some design principles. The context can be related to an organisation or application domain.

Issue 5: Necessity to use thresholds when dealing with quantitative data. Many symptoms are defined using vague terms like "few" and "too many". Although there might be a concensus on extreme values, given their different backgrounds, quality analysts may interpret different values differently.

Issue 6: Difficulty of recovering and using semantic data. The key symptoms of some antipatterns, such as Functional Decomposition, relate to semantic data, such as a class implementing a single feature. An automated detection approach should take into account semantic data.

2.3. Review of previous detection approaches

We now revisit the literature on the detection of code smells and antipatterns using the six issues as criteria.

Manual approaches were defined, for example, by Travassos et al. (1999), who introduced manual inspections and reading techniques to identify antipatterns. By putting the analyst at the heart of the detection process, they avoid issues 1, 3, 4, and 6 but face issues 2 and 5. However, as the authors note, their approach does not scale to larger systems. Ciupke (2010) proposed an approach for analysing legacy code, specifying frequent design problems as queries and locating the occurrences of these problems in a model derived from source code. The majority of the detected problems were simple ones, i.e., simple conditions with fixed threshold values, such as "the depth of inheritance tree must not exceed six levels". He did not address complex antipatterns and thus avoided issues 3, 4, and 6. However, he used fixed thresholds and as such did not address issues 1, 2, and 5.

Marinescu (2004) presented a metric-based approach to detect antipatterns with "detection strategies" that capture deviations from good design principles and combine metrics with set operators and compare their values against absolute and relative thresholds. Similarly to Marinescu, Munro (2005) proposed metric-based heuristics to detect code smells; the detection heuristics are derived from a template similar to the one used for design patterns. He also performed an empirical study to justify the choice of metrics and thresholds for detecting code smells. This work addressed only the issues 5 and 6. It cannot handle uncertainty, issue 1. Their detection approaches include neither the context of the programs, nor the quality analysts' interpretations, issues 3 and 4. They return lists of candidates without any ranking, issue 2.

Alikacem and Sahraoui (2006) proposed an ad hoc domain-specific language (DSL) to detect violations of quality principles in programs. This language allows the specification of fuzzy-logic rules that include quantitative properties and relationships among classes. The thresholds for quantitative properties are replaced by fuzzy labels. The mapping between metric values and fuzzy labels is performed through membership functions that are obtained by fuzzy clustering. Although, fuzzy inference allows to explicitly handle the uncertainty of the detection process and rank the candidates, the author did not validate their approach on real programs and only addressed issues 5.

Rao and Reddy (2008) proposed the use of design change-propagation probability matrices (DCPP matrix) to detect the Shotgun Surgery and Divergent Change antipatterns. They used the change propagation between the design of artifacts to specify these two antipatterns. A probability matrix does model the uncertainty of the detection process, issue 1. Their detection approach takes into account neither the quality analysts' interpretations, nor the context of systems, respectively issues 3 and 4. Moreover, it does not use thresholds but compares candidates to fixed templates representing the antipatterns and thus cannot report ranked lists, issues 2 and 5. No semantic data is used, issue 6.

Moha et al. (2010a) proposed a DSL to specify antipatterns based on a literature review of existing work. The specification of antipatterns takes the form a rule cards, which we used in our previous work (Khomh et al., 2009). They also proposed algorithms and a platform to automatically convert rule cards into detection algorithms (Moha et al., 2010b). They were able to identify all existing occurrences of four antipatterns: Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, with 100% recall at the expense of precision, between 41% and 88%, average 60%. Their detection approach addressed issues 4, 5, 6 but does not handle the uncertainty for the results, issue 1. It does not provide ranked list of candidates, issue 2, and cannot learn from the quality analysts' interpretations, issue 3.

Some visualisation techniques (Dhambri et al., 2008; Simon et al., 2001) were used to find a compromise between fully automatic detection approaches, which are efficient but which lose track of the context, and manual inspections, which are slow and subjective (Langelier et al., 2005). However, none of them addressed issues 1 and 2. Other approaches perform fully automatic detection and use visualisation techniques to present the detection results (Lanza and Marinescu, 2006; van Emden and Moonen, 2002). These approaches do not handle uncertainty and long lists, issues 1 and 2. They also do not consider quality analysts' interpretations and thresholds, issues 3 and 4.

Most previous approaches are based on rules that make rigid Boolean decisions as to whether or not a class is an antipattern. These rules do not provide the uncertainty of their decisions, e.g., a probability. Furthermore, they do not provide a way to leverage past results to improve their detection performance in precision and recall. Regarding the six issues and previous approaches, Table 2 summarises their strengths and weaknesses.

Approaches		Issue 1	Issue 2	Issue 3	Issue 4	Issue 5	Issue 6
Manual	Travassos et al. (1999)	√		√	√		√
	Ciupke (2010)			√		√	√
	Marinescu (2004)					√	√
	Dhambri et al. (2008)			√	√	√	
	Munro (2005)					√	√
	Alikacem and Sahraoui (2006)					√	
Semi-automated	Rao and Reddy (2008)	√					
	Moha et al. (2010b)				√	√	√
	Simon et al. (2001)			√	√	√	
	Lanza and Marinescu (2006)				√		
	van Emden and Moonen (2002)				√		
	BDTEX	√	√		√	√	√

2.4. Discussion

BBNs have been successfully used to model uncertainty in fields as diverse as risk management (Cowell et al., 2007), medicine

Table 2
GQM applied to Spaghetti Code.

Goal: Identify Spaghetti Code	
Definition: Code that does not use appropriate structural mechanisms	
Q1 Does the class have long methods?	**M1** maximum LOCs in methods is high
Q2 Does the class have methods with no parameters?	**M2** # methods with no parameters is high
Q3 Does the class use global variables?	**M3** # global variable access is high
Q4 Does the class not use polymorphism?	**M4**% of non-polymorphic method calls is high
Q5 Are the names of the class indicative of procedural programming?	**M5** check names *Process*, *Init*, *Exec*, *Handle*, *Calculate*, *Make*... (terms used in Moha et al. (2010a))

(Szolovitz, 1995), and computer science (Fenton and Neil, 2007). We now discuss the benefits of BBNs over previous (semi-)automated approaches for antipattern detection.

First, previous approaches do not handle the uncertainty of the detection process and, therefore, miss borderline classes, i.e., classes similar to antipatterns but "surfacing" slightly above or "sinking" slightly below the antipatterns-related thresholds because of minor variations in the characteristics of these classes. Thus, quality analysts would not have any information about classes below the thresholds or too much information from classes just above the thresholds, which relate to issue 1. To help address issue 1, we propose the use of BBNs, whose outputs are the probabilities that classes exhibiting the symptoms of some antipatterns are really such antipatterns. With such probabilities, a quality analyst does not need to take "binary" decisions about classes by ranking classes according to their probabilities; hence, BBNs also address issue 2.

Second, a BBN can be trained on past results, which have been validated by quality analysts and obtained in the context of a given set of programs. They can also work with missing data and allow quality analysts to specify explicitly their decision process. When data is unavailable or must be adapted to a different context, an analyst can encode her judgement into the BBN. In the context of antipattern detection, this encoding is important because there are usually only a few antipattern instances in a program; hence, a database of instances would be generally too small for other techniques to learn automatically while the literature contains many quality analysts' judgements on antipatterns. Consequently, BBNs address issues 3, 4, and 5.

Although BBNs address most of the issues of previous approaches, there are other techniques capable of modelling uncertainty: machine learning techniques and statistical models. However, these techniques and models must be trained on large amounts of tagged data to be effective (each datum describing the inputs and a correct output). This reliance on tagged data limits the applicability of these approaches in an industrial context because (1) organisations rarely keep track of previously detected antipatterns, (2) there are no large public databases containing instances of antipatterns, and (3) antipatterns are relatively infrequent in programs, thus constituting a corpus is costly. Consequently, these techniques and models are not easily and directly applicable to antipattern detection. Furthermore, these types of techniques and models use black-box processes unsuitable for quality analysts who want to encode their knowledge in the process.

BBNs, by their very construction, also embed the uncertainty of the quality analysts that provided the instances of antipatterns on which to train the BBNs. Thus, as previous approaches, they depend on the quality analysts' understanding of the antipatterns and on the agreement of the analysts over a set of instances of the antipatterns. On the one hand, BBNs thus encode the peculiarities of a development context and provide occurrences that are more relevant to some analysts than a "generic" approach. On the other hand, the quality analysts could encode their particular knowledge that would lead the BBNs to report occurrences that would look erroneous to a third party. We carefully craft oracles in our experiments, as explained in Section 4.5, and make these oracles available on-line for further validation and reuse.

3. BDTEX

This section details what are BBNs and introduces BDTEX to build BBNs for antipattern detection without the support of rules from DECOR. It is illustrated using the example of the Blob.

3.1. Bayesian Belief Networks

A BBN is a directed, acyclic graph that represents a probability distribution (Pearl, 1988). In this graph, each random variable X_i is denoted by a node. A directed edge between two nodes indicates a probabilistic dependency from the variable denoted by the parent node to that of the child. Therefore, the structure of the network denotes the assumption that the values of each node X_i in the network are only conditionally dependent on its parents. Each node X_i in the network is associated with a conditional-probability table that specifies the probability distribution of all of its possible values, for every possible combination of values of its parent nodes.

A BBN is a classification function (classifier) $f: R^d \mapsto C$ that assign a label from a finite set of classes $C=\{c_1, \ldots, c_q\}$ to observations $a \in R^d$. Antipattern detection can be viewed as a classification problem where there are two possible outputs for a given object-oriented class:$C=\{antipattern, not\ an\ antipattern\}$ given an observation $(a_1, \ldots, a_d)$, a vector of inputs describing a class.

For each classification, there is a probability that the detection result is correct, which corresponds to its degree of uncertainty. It classifies a d-dimensional observation a_i by determining its most probable class c:

$$c = argmax_{ck} p(c_k|a_1, \ldots, a_d)$$

where c_k ranges over C and the observations a_i are written as a vector of dimension d. By using *the rule of Bayes*, the probability $p(c_k|a_1, \ldots, a_d)$, called *a posteriori* probability, is rewritten as:

$$\frac{p(a_1, \ldots, a_d|c_k)}{\sum_{h=1}^{q} p(a_1, \ldots, a_d|c_h)p(c_h)} p(c_k).$$

When assuming that, given a c_k, all observations are conditionally independent, the BBN structure is drastically simplified. The BBN is then a naive Bayes classifier and its common form of *a posteriori* probability is:

$$p(c_k|a_1, \ldots, a_d) = \frac{\prod_{j=1}^{d} p(a_j|c_k)}{\sum_{h=1}^{q} \prod_{j=1}^{d} p(a_j|c_h)p(c_h)} p(c_k). \quad (1)$$

The $p(c_k)$ marginal probability (Fenton and Neil, 1999) is the probability that an object-oriented class be a $c_k \in C$. The $p(a_j|c_k)$ prior-conditional probability is the probability that the *jth* observation assumes a particular value m_j given c_k. These two prior probabilities determine the structure of the BBN. They are learned, i.e., estimated, on a training set when building the classifier.

The resulting BBN is thus a simple structure (Duda and Hart, 1973) that has (1) the classification node as the output node, to which is associated a distribution of marginal probabilities, and (2) the input nodes as leaves, each of them associated with q distributions of prior-conditional probabilities.

A quality analyst needs two pieces of information to build such a BBN: the structure of the network, in the form of nodes and arcs (causal relations), and the conditional-probability tables describing the decision processes between each node. By structuring the network, the quality analyst ensures that the decision process is valid. The conditional probabilities can be learned using historical data or entered directly by the quality analysts when data is missing. The structure ensures the qualitative validity of the approach while appropriate conditional tables ensure that the BBN is well-calibrated and is quantitatively valid. The structuring can be done systematically using our GQM methodology.

3.2. GQM methodology

In our previous work (Khomh et al., 2009), we depended on the rule cards provided by the DECOR approach to obtain BBNs to detect antipatterns. This approach had two limitations: the limited expressiveness of the rules cards (in particular the available operators) and the composition of sub-rules into rules, which led to too many intermediate nodes in the BBNs. These intermediate nodes put additional constraints on the corpus to calibrate the BBN. These two limitations impeded the practical applicability of the approach.

Consequently, we propose the use of the Goal Question Metric (GQM) methodology introduced by Basili and Weiss (1984) to extract information from the antipattern definitions and build BBNs for their systematic detection, without relying on rule cards. The GQM is a good fit for this problem because it is relatively simple to map different parts of antipatterns to goals, questions and metrics. Furthermore, it is the most widespread methodology and does not have a steep learning curve.

The GQM defines a measurement model on three levels:

Conceptual level/goal: A goal is defined for an object for some reasons, with respect to some quality models, from different points of view, and in a given context. In this paper, our goal is to retrieve occurrences of some antipattern (one per BBN) in some programs, developed within a given context.

Operational level/question: A set of questions is used to define models of the object and then characterise the assessment or achievement of a specific goal. In this paper, our questions relate to the symptoms of the antipatterns to be detected, i.e., code smells.

Quantitative level/metric: A set of metrics is associated and used to answer every question in a measurable way. We derive from our questions the metrics necessary to answer them. In this paper, we focus on the metrics and combinations thereof required to detect classes with the symptoms identified in the questions.

After completing these three first steps for an antipattern, we obtain a BBN for that antipattern. The input nodes of this BBN correspond to the symptoms obtained at the operational level while the output node is the probability of a class being an antipattern (goal). Metric extraction is done independently from the BBN.

The context of a program can have an effect on how an antipattern will manifest itself. Therefore, BDTEX introduces two additional steps to adapt a BBN to a given context. First, some questions might be useless for a given context; they can be identified manually by a quality analyst or by applying a statistical test. This step is discussed in Section 4. Second, a BBN can be calibrated using past data when available using Bayes' theorem.

3.3. Running example

The application of BDTEX to build a BBN to detect occurrences of the Blob antipattern is illustrated in Fig. 1. Following BDTEX, we first identified the measurement goal at the conceptual level: our goal is to recover occurrences of the Blob antipattern. To operationalise this goal, we asked questions that could characterise the design of the program analysed and identify symptoms of the Blob like "Is the class a large controller?". Finally, at the quantitative level, we identified metrics that can provide answers to the questions. For example, large classes are measured using the NMD and NAD metrics (Number of Methods Declared and Attributes, respectively). Following Riel's heuristic, controller classes are detected by their names or those of their methods, which must contain terms indicative of procedural programming, e.g., *Process* or *Control*. BDTEX led naturally to a hierarchical tree structure for the BBN to maintain the relations among goals, questions, and metrics, as shown in Fig. 2. Once the structure of the BBN and the metrics are obtained, we can compute the probability distributions of each node.

Input nodes: To compute the probability distributions of the input nodes (symptoms), we first discretised metrics values into three different levels: "low", "medium", and "high". We used a box-plot to perform the discretisation. A box-plot, also known as a box-and-whisker plot, is used to single out the statistical particularities of a distribution and allows for a simple identification of abnormally high or low values. Fig. 3 illustrates the box-plot and the thresholds that it defines: LQ and UQ correspond respectively to the lower and upper quartiles that define thresholds for outliers.

For each class in a system, and each symptom, the probability that the class presents the symptom is computed as follows:

- For symptoms captured by metric values, the probabilities are calculated as follows: we use three groups ("low", "medium", "high") and estimate the probability that a quality analyst would consider the metric values as belonging to each group. Limiting the number of groups to three simplifies the interpretation of the detection results. For each metric value, the probability is derived by calculating the relative distance between the value and its surrounding thresholds. The probability is interpolated linearly as presented in Fig. 3.
- For symptoms characterised by lexical properties describing class names, probabilities are either 0 or 1, whether the name contains a term or not. For method names, we treated the number of methods containing the term as a metric and used the box-plot to interpolate a probability.
- For symptoms that determine the strength of relations, the probabilities are calculated using the numbers of such relations, (e.g., the number of data classes with which a class is associated). The more a class is associated to data classes, the more likely it is a Blob. Data classes are identified using the 90% accessor ratio. To convert this count to a probability distribution, its value is interpolated between 0 and N where N is the upper outlier value observed in the program.

Output nodes: The probability of the output node is inferred from the probabilities of input nodes using Bayes' theorem. Every

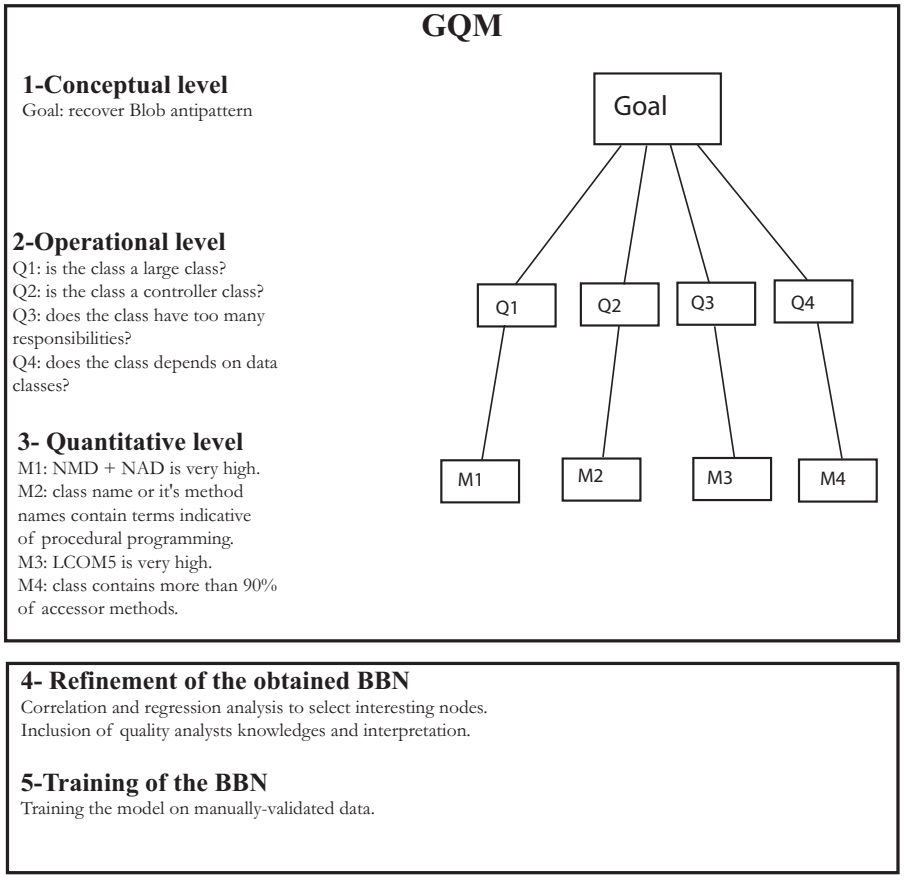

Fig. 1. BDTEX applied to the detection of the Blob antipattern.

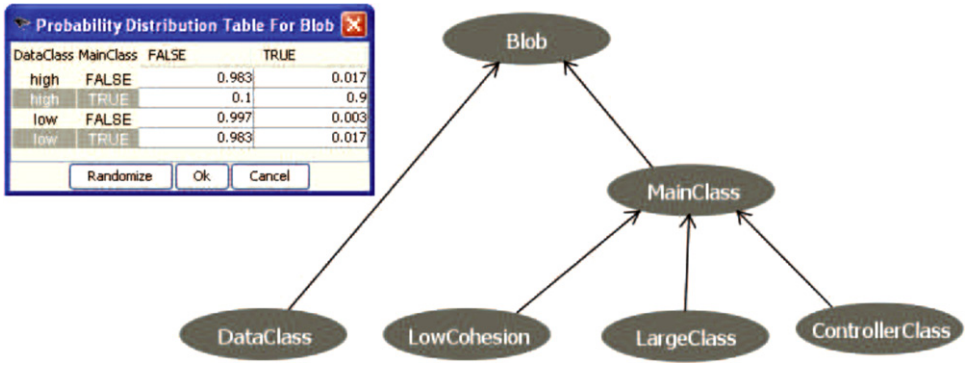

Fig. 2. Bayesian Belief Network for the detection of the Blob antipattern.

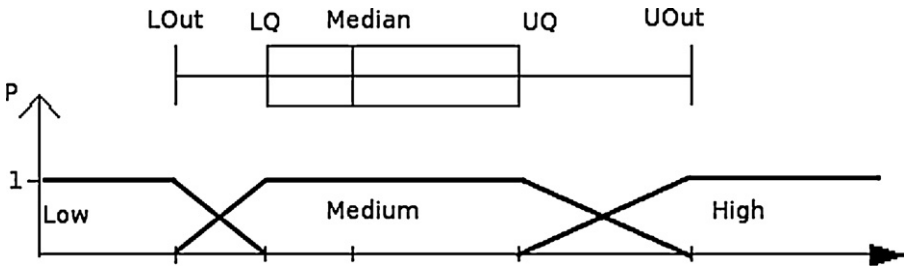

Fig. 3. Probability interpolation for metrics.

Table 3
GQM applied to Functional Decomposition.

Goal: Identify Functional Decomposition	
Definition: Object-oriented code that is structured as function calls	
Q1 Does the class use functional names?	**M1** same as **Q5**, in Table 2
Q2 Does the class use object-oriented mechanisms no parameters?	**M2** *NMO* > 0 or *DIT* > 1
Q3 Does the class use classes with functional names?	**M3**% of invocations to classes/methods with functional names (like **Q1**)
Q4 Does the class declare a single method?	**M4** *NMD* = 1
Q5 Are all the class attributes private?	**M5**% of private attributes = 100%

output node has a conditional probability table to describe the decision given a set of inputs. In our example, the probability of a class being a Blob depends directly on the two symptoms: *MainClass* and *DataClass*. We can use previously tagged data to fill the conditional probability table with $P(Blob|MainClass, DataClass)$, $P(Blob|MainClass, \neg DataClass)$, $P(Blob|\neg MainClass, DataClass)$, and $P(Blob|\neg MainClass, \neg DataClass)$. The node *MainClass* is an output node depending on three input nodes: *LowCohesion*, *LClass*, and *ControllerClass*.

4. Experiments

We now perform experiments to validate some BBNs obtained from BDTEX and to answer the two following research questions:

RQ1: To what extent a BBN built using our approach is able detect antipatterns in a program?
RQ2: Are the results of a BBN built using our approach better than those of a state-of-the-art approach, DECOR?

4.1. Preliminary

The *goal* of our experiments (Basili and Weiss, 1984) is to improve the quality of programs by improving the detection of antipatterns. Our *purpose* is to provide an approach to support uncertainty in antipattern detection. The *quality focus* is to provide a sorted set of occurrences of antipatterns that prioritise the most probable candidate classes. The *perspective* is that of quality analysts, who perform evaluation activities and are interested in locating parts of a program that need improvements with the least possible efforts. The *context* of our study is both development and maintenance.

4.2. Answering the research questions

To answer **RQ1**, we studied the accuracy of our BBNs in two scenarios. First, we assumed that there is historical data available for a given program (e.g., manually validated instances of Blob). This data was used to calibrate a BBN, which was then applied on the same program and the returned occurrences compared with the expected instances. Second, we studied the accuracy of our BBNs using heterogeneous data: we calibrated the BBNs using antipattern instances from one program and applied them on another program. This setup was used to answer **RQ2**, we show that our BBNs produced equivalent or better results than DECOR while being more flexible.

4.3. Building the BBNs

We focus on three antipatterns: Blob, Functional Decomposition, and Spaghetti Code to answer our research questions, whose definitions are recalled in Table 1. We chose these antipatterns because they are well-known and have been used in the literature to perform other experiments, in particular in the previous work by Moha et al. (2010a) against which we will compare our approach. We also choose them because it is possible to detect their occurrences using structural data, unlike other antipatterns, such as Poltergiest, which describes short-lived objects in the execution of programs. Finally, we select only three antipatterns because building oracles for all structural antipatterns would require a tremendous amount of manual work and is, therefore, future work.

We now present the BBNs corresponding to the chosen antipatterns obtained by applying BDTEX. We detail the Blob as our running example:

- **Conceptual level (goal):** Identify occurrences of the Blob antipattern, defined as classes that do or know too much.
- **Operational level (question):** Following the definition of the Blob in the literature (Brown et al., 1998), the symptoms to look for when identifying Blobs are: large, controller classes, with too many responsibilities and depending on data stored in surrounding data classes. These symptoms yield the following questions: **Q1** Is the class a large class? **Q2** Is the class a controller class? **Q3** Does the class have too many responsibilities? **Q4** Does the class depend on data classes?
- **Quantitative level (metric):** To answer **Q1**, we consider that a large class is a class that declares a very large number of fields and methods; structural property captured by the NMD and NAD metrics (Number of Methods/Attributes Declared). To identify a controller class, **Q2**, we look for a class that monopolises most of the processing done by a (part of a) program, takes most of the decisions, and directs the processing of other classes (Wirfs-Brock and McKean, 2002). We also use Riel's heuristic to identify controller classes using names of classes or methods, such as *Process*, *Control*, *Manage*, *System*. We use weak cohesion measured using the LCOM5 metric to answer **Q3**. To answer **Q4**, we count the number of associations to data classes as identified by the detection strategy defined in DECOR (Moha et al., 2010a) and include this count in the form of a dedicated metric, NoDC.

We summarise in Tables 2 and 3 the application of BDTEX to the Functional Decomposition and Spaghetti Code antipatterns. The questions describe the symptoms presented in Brown et al. (1998). Symptoms that were not detectable directly from the code were omitted. In the tables, NMD, NMO, and DIT respectively stand for the numbers of methods declared, of methods overridden, and the depth of the inheritance tree.

4.4. Choosing the programs

We used two open-source Java programs to perform our experiments: GanttProject v1.10.2, Xerces v2.7.0, presented in Table 4.

GanttProject[1] is a tool for creating project schedules by means of Gantt charts and resource-load charts. GanttProject enables breaking down projects into tasks and establishing dependencies

[1] AAA http://ganttproject.biz/index.php.

Table 4
Program statistics.

Programs	#Classes	KLOCs
GanttProject v1.10.2	188	31
Xerces v2.7.0	589	240
Total	777	271

Table 5
Manual identification of instances of the antipatterns: # of classes per vote count.

Programs	# Blobs		# S.C.		# F.D.	
	3–4 votes	5 votes	3–4 votes	5 votes	3–4 votes	5 votes
GanttProject v1.10.2	4	4	7	4	13	5
Xerces v2.7.0	29	15	23	18	15	0

between these tasks. Xerces[2] is a family of software packages for parsing and manipulating XML. It implements a number of standard API for XML parsing, including DOM, SAX, and SAX2. Other implementations are available for C++, and Perl.

We chose GanttProject and Xerces because they are of medium size, yet are small enough to manually locate antipatterns. Instance of the three antipatterns were manually detected to form an oracle of known antipatterns. This manual process is described in the following section. All metrics and properties required to detect antipatterns are extracted using the framework (Gueheneuc et al., 2004).

4.5. Building the oracles

Before applying the BBNs to answer the research questions, we built oracles of manually validated instances of the three antipatterns on the two programs to serve as oracles in the experiments. To the best of our knowledge, these oracles are among the few existing and we make them available on-line[3] to help other researchers interested in antipattern detection.

To build the oracles, we asked four undergraduate students and three graduate students to identify occurrences of the three antipatterns in the two programs. Undergraduate students performed the task in pairs to follow previous results (Ricca et al., 2008) hinting that the performance of a pair of undergraduate students is about the same as that of one graduate student, which gave us confidence on the instances of the antipatterns reported by the undergraduate students and, thus, freed us from redoing their work to confirm their findings.

The students were presented with examples of several antipatterns. Then, each student/pair analysed every class of the programs systematically to answer the boolean question: "Is this class a Blob (Functional Decomposition or Spaghetti Code, respectively)?". We independently combined their votes such that if at least three of the five students/pair considered a class an antipattern, then we tagged it as a true occurrence, i.e., an instance of the antipattern. The number of identified instances is reported in Table 5. In both programs, there is often conflicting opinions on what is an occurrence of an antipattern. These conflicts are most apparent for Functional Decomposition because the developers' intent is more important for that antipattern than for either the Blob or the Spaghetti Code.

4.6. Calibrating the BBNs

In our experiments, we divided the oracles in two groups: one was used to calibrate the BBNs and the other to assess their results. The calibration of a BBN is the evaluation of the conditional-probability tables that describe the probabilities of having an occurrence of an antipattern given a combination of input values.

Using the known instances of the three antipatterns, we computed the conditional probabilities using the Weka machine learning framework (Witten and Frank, 1999). The framework found, for every combination of inputs, the probabilities of a given output. To calibrate a BBN, Weka needs nominal inputs. We considered any input with a probability of 1 as "high" whereas any other value is "low" to obtain the nominal training dataset.

Given the relatively large number of inputs (symptoms) in our oracles and the small number of instances, we applied "bootstrapping" to improve the calibration. We counted every tagged antipattern proportionally to the number of votes in its favour. For a class considered as an antipattern by four out of five students/pairs, we duplicated the structure of this class four times in the corresponding oracle. Thus, a class from an oracle is considered three, four, and five times respectively according to the number of votes it received. This bootstrapping has the advantage of increasing the weight of classes with high consensus and also the size of positive data with which our BBNs are calibrated. Moreover, this increase in the weight of the structure of classes with higher numbers of votes should improve the likelihood that our BBNs return classes with a structure similar to the antipatterns with high probabilities. We completed the calibration using randomly selected classes without the corresponding antipatterns, a number of classes equal to the number of classes with the antipatterns. Having a balanced data set helps us avoid the negative effect skewed data has on predictive models (Provost, 2000).

4.7. Performing the experiments

Our experiments were divided in two scenarios, each corresponding to a research question. In different programs and scenarios, different symptoms might be more important than others. Therefore, for each scenario, we used the available data to adapt the BBNs. The adaptation will consist of identifying the symptoms that are not discriminating between classes that are antipatterns and those that are.

4.7.1. Scenario 1: using local data to calibrate a BBN

In this first scenario, we studied how instances of antipatterns in a program can be used to identify occurrences of these antipatterns in the same program. To avoid overfitting the BBNs, we performed a 3-fold cross-validation, which required a BBN to be executed three times using different parts of the oracles. The oracles are split into three equal-sized groups and every execution uses two groups for calibrating and one for testing.

For every input, we tested if the proportion of antipatterns containing the symptom was the same as that of non-antipatterns. When the symptom was discriminant and positively related to the antipattern, it was retained; otherwise, it was eliminated. The different symptoms with their respective p-values are presented in Table 6. Symptoms that are statistically but negatively related have their *p*-value struck through. Table 6 shows that only a few symptoms are relevant for every antipattern.

To determine whether or not a class is a Blob, we can rely only on its size (**Q1**) and on its use of data classes (**Q4**). We cannot rely on names as they seem to be context-specific and anecdotal. Finally, the Blobs were always relatively cohesive. The identification of the Functional Decomposition and Spaghetti Code antipatterns can rely only on a few symptoms. The influence of method size is useful for the Spaghetti Code as is naming and the use object-oriented mechanisms for Functional Decomposition. The importance of most other symptoms depends on the program. This dependence shows that a

[2] AAA http://xerces.apache.org/.
[3] AAA http://www.ptidej.net/downloads/experiments/jss10/.

Table 6
Intra project validation, salient symptom identification.

Antipatterns	Symptoms	% in GanttProject		p-Values	% in Xerces		p-Values
		AP	Not AP		AP	Not AP	
Blob	**Q1**	100%	4%	0*	94%	3%	0*
	Q2	0%	7%		5%	1%	53%
	Q3	0%	26%		0%	19%	
	Q4	72%	8%	0*	63%	48%	0.01*
S.C.	**Q1**	100%	6%	0*	89%	6%	0*
	Q2	49%	3%	0*	19%	10%	0.70
	Q3	38%	4%	0.01*	12%	2%	0.19
	Q4	0%	44%		27%	49%	
	Q5	0%	2%		36%	18%	0.01*
F.D.	**Q1**	75%	47%	0*	87%	18%	0*
	Q2	70%	39%	0*	60%	19%	0*
	Q3	6%	8%	0.93	67%	26%	0*
	Q4	63%	17%	0*	7%	8%	0.92
	Q5	6%	4%	87%	0	3%	
	Q6	75%	35%	0*	33%	57%	0.07

* p-values < 0.05.

general BBN could take into account the main symptoms and then be enhanced using context-specific symptoms.

All the results for the intra-project validation are presented in Fig. 4. Each sub-figure shows the average precision/recall curves for each combination of antipattern/program. The identification of Blob and Spaghetti Code is accurate with few false positives. We assume that this accuracy is due to the notions of size, which are very important symptoms. Size is one of the few concepts that can be measured precisely unlike the indirect symptoms used to identify the developers' intent, like object-oriented mechanisms, for Functional Decomposition.

4.7.2. Scenario 2: inter program validation

In this second scenario, we assumed that a quality analyst has access to historical data from another program. She would therefore calibrate the BBNs using this data and apply the BBNs on her other program.

As shown in Table 7, the usefulness of most symptoms does not cross program boundaries, we therefore used the intersection of valid symptoms. For the Spaghetti Code, this intersection is limited to one symptom, thus we also included the use of global variable to identify its occurrences.

The results of the BBNs are presented in Fig. 5. Unlike the intra-project validation were we computed precision and recall for all the corpus (3+ votes), we chose here to present results for unambiguous instances of antipatterns (5 votes) as these are the results that would most likely satisfy any quality analyst. The exception is for Functional Decomposition on Xerces because it had no instances with 5 votes. For that program, we computed the precision and recall for instances with 3+ votes.

The BBN of the Blob identified all the instances of this antipattern requiring the inspection of a candidate set half the size of that of DECOR. For Spaghetti Code, the inclusion of global variables adversely affected the BBN ability to identify occurrences of that antipattern in Xerces, but for GanttProject in which the instances found were the same. Finally, the Functional Decomposition BBN produced mixed results: good precision and recall for Xerces; the five top occurrences correctly for GanttProject.

These are promising results because they suggest that even in the absence of historical data on a specific program, a quality analyst can use a BBN calibrated on different programs and obtain acceptable precision and recall. These results also show that a BBN could be built using data external to a company and then be adapted and applied in this company successfully. These BBNs could also be enriched with locally salient symptoms to further improve their accuracy.

4.8. Comparing with DECOR

The results of rule-based detection approach is a set of candidate classes suggested to quality analysts for correction or improvement. A quality analyst would have to either validate the whole candidate set or choose to inspect a subset without any indication on which classes to review first, because there is no order in the results. BBNs provide probabilities that candidate classes are occurrences of some antipatterns, which can help prioritise inspections.

Therefore, we used the Rank-Biased Precision (RBP) metric introduced by Moffat and Zobel (2008) to measure and compare the effectiveness of our BBNs DECOR. RBP has been demonstrated to be a better measurement tool than precision or recall-based metrics (Moffat and Zobel, 2008). Moreover, this metric reflects the satisfaction of a quality analyst in absolute terms. RBP assumes that a quality analyst always starts by examining the top-ranked classes of a list of candidates classes, progressing from one class to the next with a probability p, and, conversely, end her examination of the ranking at a point with probability $1-p$. Each termination is decided independently of the current depth reached in the ranking, of previous decisions, and of whether or not the class just examined was relevant or not.

RBP also assumes that, as a quality analyst skims through candidate classes, they are willing to pay $1 for each true positive found, but nothing for false positives. This assumption leads to a notion of income for detection approaches, based on the utility gained by the quality analysts. As the quality analysts progress down the ranked lists of candidate classes, they are thus running up an account with the detection technique, or, equivalently, increasing their total utility every time they find a true positive. The total expected utility derived by the quality analyst (RBP), and the income payable to the detection technique, are given by:

$$\mathrm{RBP}(p) = (1-p)\sum_{i=1}^{d} r_i p^{(i-1)}$$

where $r_i \in \{0, 1\}$ is the relevance judgement of the *ith* ranked class (which is 1 if the class is a true positive and 0 otherwise), d is the length of the list, and the $(1-p)$ factor is used to scale the RBP within the range [0,1].

Table 7 presents the results of the comparison of our BBNs with DECOR. For all the three antipatterns: Blob, Spaghetti Code and Functional Decomposition, and for three categories of quality analysts: impatient ($p = 0.5$), patient ($p = 0.8$), and very patient ($p = 0.95$), the satisfaction of quality analysts appeared to be always higher for BBNs than for DECOR except for the Spaghetti Code on Xerces. The exception of the Spaghetti Code is caused by the adverse impact of global variables on the precision of the corresponding BBN, as discussed previously in Section 4.7.2.

5. Discussion

We now discuss the experiments and the use of BDTEX by quality analysts based on the results of the experiments.

5.1. Using BBNs in an industrial context

We showed that our BBNs are able to efficiently prioritise candidate classes that should be inspected by a quality analyst, using the RBP metric. The BBNs, built using BDTEX and calibrated using external data, can successfully identify occurrences of antipatterns. The programs were of different nature (a parser library vs. a full-fledged

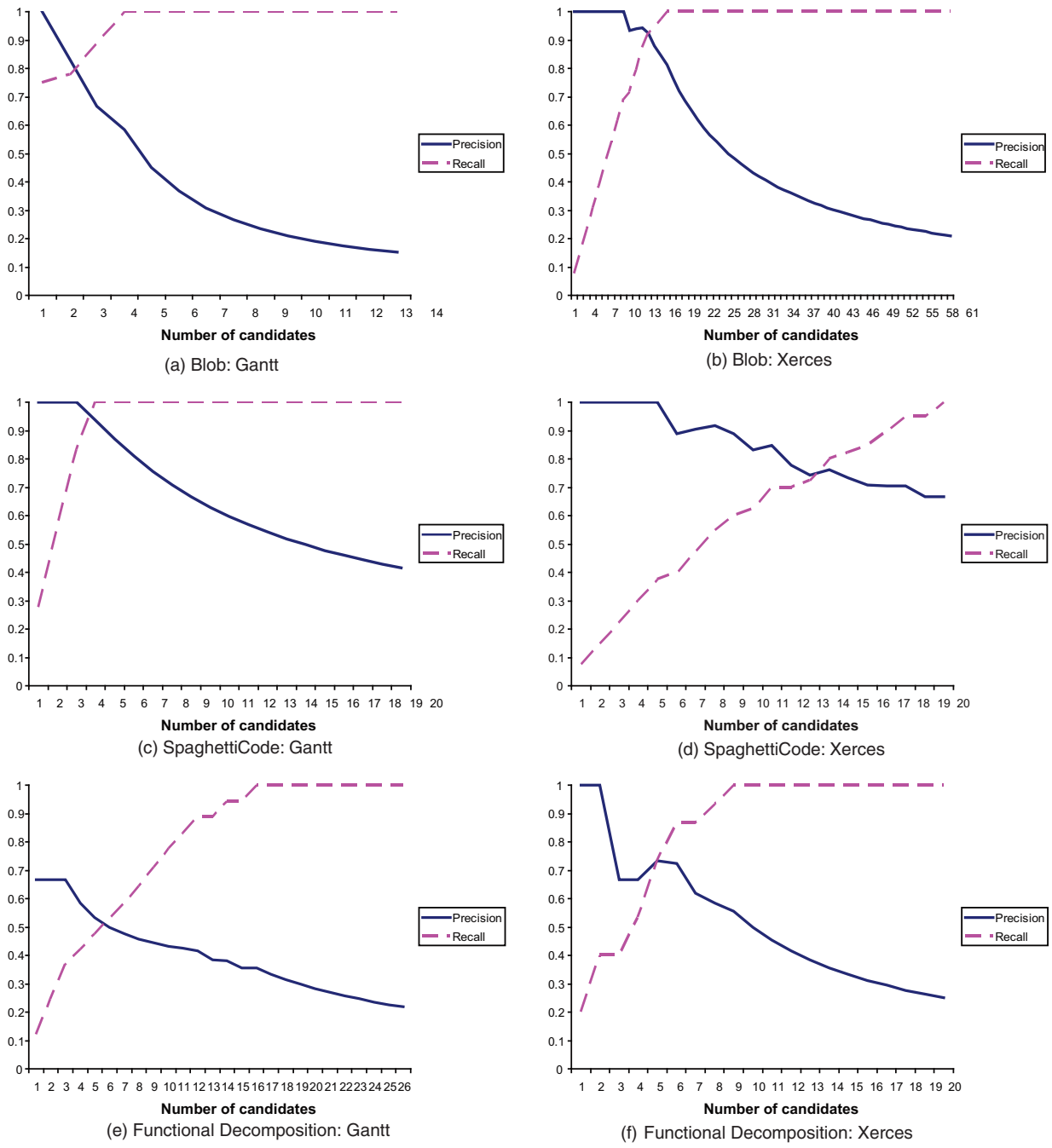

Fig. 4. Intra-project calibration: average precision and recall.

Table 7
Comparison of the utility of BBN vs. DECOR.

Programs	Antipatterns	$p = 0.5$		$p = 0.8$		$p = 0.95$	
		BBN	DECOR	BBN	DECOR	BBN	DECOR
GanttProject	Blob	0.36	0.18	0.4	0.26	0.17	0.15
	S.C.	0.38	0.1	0.34	0.16	0.13	0.1
	F.D.	0.48	0.12	0.55	0.2	0.26	0.13
Xerces	Blob	0.2	0.17	0.41	0.27	0.39	0.29
	S.C.	0.12	0.29*	0.18	0.47*	0.2	0.54*
	F.D.	0.23	0.21	0.44	0.33	0.33	0.29

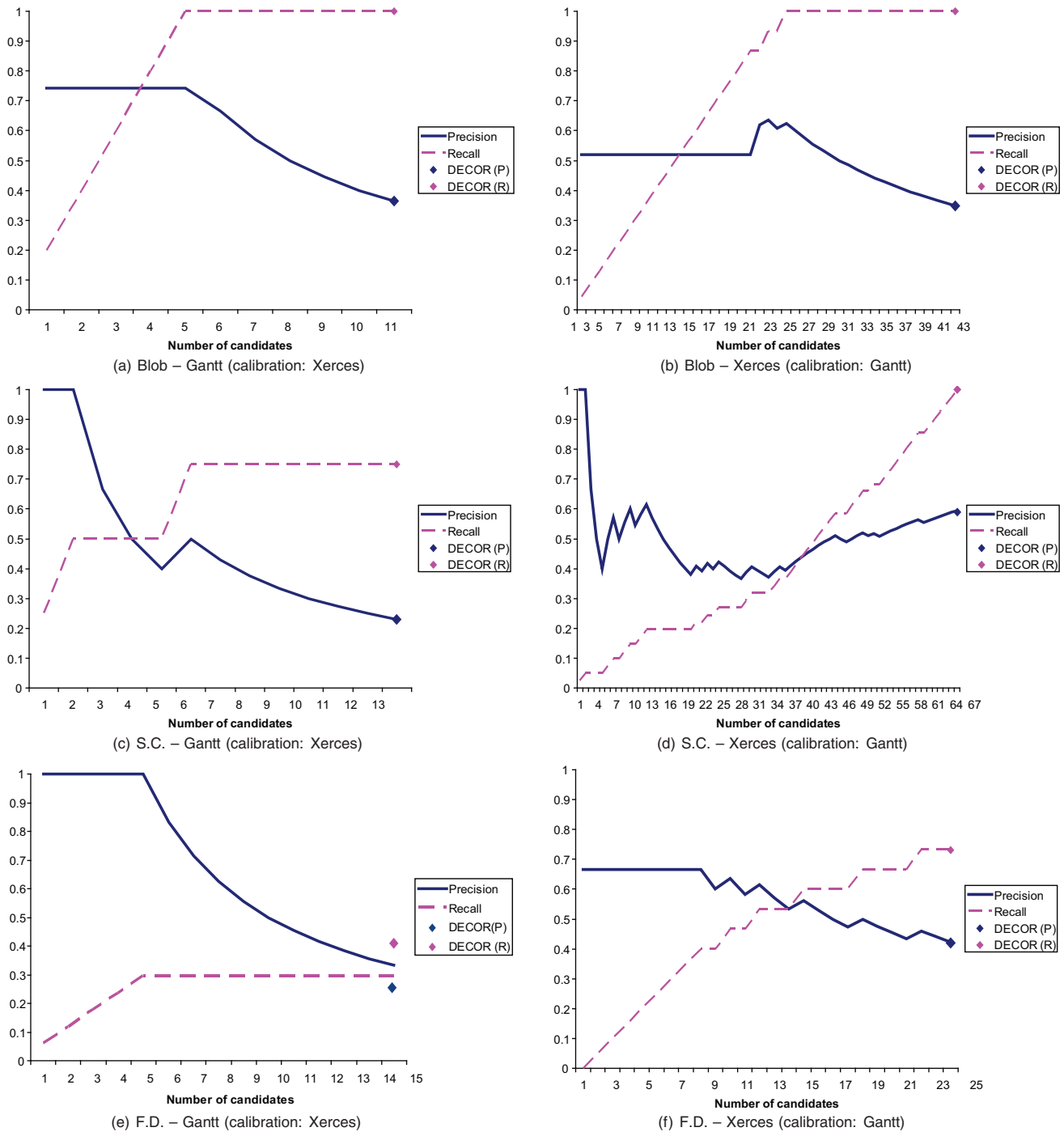

Fig. 5. Intra-project validation.

GUI) and developed by different development teams. Therefore, quality analysts in their industrial contexts could build a repository of antipatterns on a set of programs and use this data to calibrate BBNs and identify antipatterns in other programs. The customisation that we performed in the experiments was minimal, but in an industrial context, it could be much more important.

5.2. Estimating the number of occurrences of antipatterns

Estimating the number of occurrences of some antipatterns in a program is important to stop investigating spurious candidate classes. A well-calibrated BBN should be able to estimate this number in a program.

In practice, when quality analysts are confronted with long ranked lists of candidates classes, they employ ad hoc methods to reduce their efforts while improving their utility:

- They limit the manual validation to the top 20% of candidate classes because Pareto's law states that defects (in general) are located in only 20% of the code.
- They count the distance between every two true positive and stop inspecting candidate classes as soon as this distance is greater than the average distance of all previous two candidates.

We contribute to these effort-reduction methods by proposing a variant of RBP to compute a stop point at which the utility of a quality analyst is maximal. In the previous section, we choose to assign \$1 for each true positive and \$0 for false positives. We now assign a cost c for false positives and, thus, RBP is now defined as:

$$\mathrm{RBP}(p) = (1-p)\sum_{i=1}^{d}(r_i\lambda_i + (1-\lambda_i)(-c))p^{(i-1)}$$

where $r_i \in \{0, 1\}$ is the relevance judgement of the *ith* ranked class, λ_i is the probability provided by the BBN that the *ith* ranked class is a true positive, d is the length of the list, and the $(1-p)$ factor is used to scale the RBP within the range [0,1].

The optimum of this new definition of RBP is reached as soon as $\lambda_i \leq (c/c+1)$ at a rank i. Therefore, if a quality analyst values the cost of her reviewing a false candidate to −\$1, she should stop reviewing candidate classes as soon as the probability of the candidates is under 0.5, because her maximal utility would then be reached.

The value of the RBP metric depends on the precision of the BBN. Therefore, we suggest that quality analysts apply this effort-reduction method more than once. After reaching the stop point, quality analysts should re-calibrate the BBN to improve its precision as well as the estimation of the stop point.

5.3. Improving antipattern detection

There are different means with which a BBN could be adapted to a context to further improved its accuracy. The first means is by selecting a different interpolation technique for metric values. In our experiments, we found that a significant number of classes have equivalent probabilities of being an antipattern because $p(high) = 1$ as soon as a metric value is over the corresponding upper outlier value. In future work, we will investigate the use of higher thresholds and perform a more complex interpolation to ensure that any outlier value is flagged with a probability capturing its magnitude.

Another means would be to reconsider the method used to calibrate the BBNs. Weka required nominal data and, consequently, the BBNs were less accurate than if they could take as input continuous values. Building up a repository of instances of antipatterns will also help improve the size of the oracles and, thus, the precision of the BBNs. In future work, we also plan to investigate the use of negative rules as they could help improving the precision through their specification of false candidates to avoid.

5.4. Applying BDTEX to other antipatterns

BDTEX is general and can be applied to detect other antipatterns providing that (1) the characteristics of classes with these antipatterns can be measured using metrics and (2) oracles of known instances of these antipatterns are available. Any structural antipattern potentially satisfies the first condition. The second condition is more difficult to satisfy because, to the best of our knowledge, our oracles of instances of Blob, Functional Decomposition, and Spaghetti Code are the first of such freely available oracles.

5.5. Scaling up to industrial programs

An issue with other detection approaches concern their application to large programs. DECOR reports that it detects on Eclipse a significant part of its classes as antipattern candidates. Because DECOR results are not ranked, this approach does not scale up: a quality analyst would not know which are the worst of the hundreds of returned classes.

BDTEX produces ranked results and, at least in theory, should not be affected by such issues. We wanted to confirm this by applying our approach on the Eclipse JDT (over 3200 classes) and by comparing the number of top ranked candidate classes with the result set returned by DECOR. Eclipse JDT[4] is a Java plug-in to Eclipse providing a rich integrated development environment for Java.

Table 8
EclipseJDT: inspection sizes.

Antipatterns	DECOR	BDTEX
Blob	4%	0.2%
S.C.	14%	1.5%
F.D.	3%	2%

Table 9
JHotDraw: inspection size.

Antipatterns	# Inspected classes
Blob	1
S.C.	2
F.D.	0

There are significant differences in the number of returned classes by our BBNs and by DECOR, as shown in Table 8. Following our running example, we manually inspected the nine top-ranked Blobs and found that 5/9 classes were antipatterns. For 2/9 of the candidate classes, we could not reach a consensus as to whether or not these classes were Blobs. The other 2/9 candidate classes were clearly not Blobs. BDTEX is thus able to return only a limited number of candidate classes with a high-level of precision: 55% for unanimously classified Blob.

5.6. Dealing with good programs

Another issue with detection approach concerns their application to "good" programs and the number of false positives. Therefore, we applied the BBNs obtained from BDTEX on JHotDraw. JHotDraw[5] is a GUI framework for structured graphics (176 classes). It was designed as a design exercise by object-oriented experts. It is a good reference of a program that most likely does not contain any occurrence of antipatterns.

Table 9 presents the numbers of classes that present all symptoms of the three antipatterns. These numbers indicate that only a few candidate classes need to be inspected by quality analysts.

The reason why the BBNs return these false positives is the relative nature of our symptoms. For the detection of Blob and Spaghetti Code, size is an important symptom. Even in a program with generally small classes (like JHotDraw), the BBNs identify those that are relatively large.

It is possible for an organisation to impose global thresholds in the detection process. In fact, if we had used the relative thresholds values measured on Xerces and GanttProject to identify occurrences of the antipatterns in JHotDraw, there would have been no candidate classes corresponding to the highest level of probability.

5.7. Classifying antipatterns

From the results presented in Section 4 and our experience calibrating BBNs to detect antipatterns, we observed that their precision depends on the definition of the antipatterns. The BBNs yield better precision for antipatterns like the Blob, of which symptoms are precisely defined and can be easily quantified using metrics. Antipatterns like the Spaghetti Code, in the contrary, involve abstract concepts in their definition and require to understand the developers' intent. Automatically generated parsers for example

[4] AAA http://www.eclipse.org.
[5] AAA http://www.jhotdraw.org.

present many symptoms of Spaghetti Code, i.e., large classes with long and complex methods but only a quality analyst can accurately evaluate them in the context of her program.

Consequently, we propose a classification of antipatterns into three types depending on the effort to map their definitions to automatic detection approaches in general, and BDTEX in particular:

- **Type 1 antipatterns** includes antipatterns which definitions are precise enough to be expressed directly in terms of characteristics of the source code. Most all the data required to identify occurrences of that type of antipatterns can be found in the code. Blob and MisplacedClass are two examples of Type 1 Antipatterns.
- **Type 2 antipatterns** are antipatterns which definitions include the developers' intent. This intent cannot be found by analysing automatically the source code. Spaghetti Code and Functional Decomposition are examples of antipatterns belonging to this category. When considering if a class is an occurrence of Spaghetti Code, in addition to finding the antipattern symptoms (long methods with no parameters, use of global variables, etc.), the quality analysts' judgement is required to confirm potential candidates by separating those introduced by choice because they are the best solution to a particular problem.
- **Type 3 antipatterns** include antipatterns which presence can only be confirmed when considering the evolution of the program. This category stems from our experience specifying and detecting antipatterns, no antipattern studied in this paper is concerned by this category. ShotgunSurgery is such a Type 3 Antipattern: it requires analysing whether a change to a part of a program required many other changes elsewhere.

6. Conclusion

In this paper, we presented a GQM-based approach, BDTEX, to systematically build BBNs to detect occurrences of antipatterns in programs. BDTEX provides a theoretically sound approach to operationalise an abstract definition of an antipattern into a BBN for its detection. Symptoms specifying antipatterns are selected by a quality analyst, thus ensuring that the BBN is qualitatively sound. Calibration is done automatically using Bayes' theorem. Consequently, BBNs have two main benefits with respect to previous approaches: they work with missing data and can be tuned using quality analysts' knowledge. In addition, candidate classes, i.e., potential antipatterns, are associated with probabilities, which indicate the degree of uncertainty that a class is indeed an occurrence of some antipattern. These probabilities can help focus manual inspection by ranking the candidate classes.

To validate BDTEX, we built the BBNs of the Blob, Functional Decomposition, and Spaghetti Code, which are complex antipatterns requiring the evaluation of different sets of classes. We performed a population test to retain only the most useful symptoms characterising their instances. The resulting BBNs were calibrated and evaluated on two programs, GanttProject v1.10.2 and Xerces v2.7.0, showing high precision and recall; these BBNs successfully assigned high probabilities to candidate classes that were *indeed* occurrences of antipatterns. Finally, we also showed that, with one exception, the results of the BBNs obtained from BDTEX are superior to these of the state-of-the-art approach DECOR in terms of precision, recall, and quality analysts' utility.

We also provided an extensive discussion on the challenges, strengths, and weaknesses of a BBN-based approach with a systematic criteria-based comparison of the state-of-the-art. We also discussed the applicability and utility of BDTEX in an industrial context and proposed a classification of antipatterns into three types depending on the effort needed to map their definitions to automatic detection approaches.

In future work, we plan to extend the validation of BDTEX to other antipatterns from the literature. We will also improve the computation of the probability distributions of the input nodes by using continuous distributions and improving the interpolation. We also plan to investigate the use of negative rules and study other machine learning techniques, such as support vector machine. Finally, we will integrate in our quality Web portal, SQUANER,[6] a tool to support the usage of the proposed approach so that it can be applied on any program to obtain antipattern probabilities. This integration would also be useful to study the classification of antipatterns by different quality analysts, depending on their agreement on the definitions of antipatterns and on sets of reported occurrences.

Acknowledgments

This work has been partly funded by NSERC, in particular the Discovery Grant #293213 and the Canadian Research Chair Tier II in Software Patterns and Patterns of Software.

References

Alikacem, E., Sahraoui, H., 2006. Détection d'anomalies utilisant un langage de description de règle de qualité. In: Rousseau, R., Urtado, C., Vauttier, S. (Eds.), actes du 12e colloque Langages, Modèles, Objets. Hermès Science Publications, pp. 185–200, http://objet.e-revues.com/article.jsp?articleId=7976.

Basili, R., Weiss, D.M., 1984. A methodology for collecting valid software engineering data. IEEE Transactions on Software Engineering 10 (6), 728–738, www.research.avayalabs.com/user/weiss/Publications.html.

Brown, W.J., Malveau, R.C., Brown, W.H., McCormick III, H.W., Mowbray, T.J., 1998. Anti Patterns: Refactoring Software, Architectures and Projects in Crisis, 1st ed. John Wiley and Sons, www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase_theanti patterngr/103-4749445-6141457.

Ciupke, O., 1999. Automatic detection of design problems in object-oriented reengineering. In: Firesmith, D. (Ed.), Proceeding of 30th Conference on Technology of Object-Oriented Languages and Systems. IEEE Computer Society Press, pp. 18–32, http://www.computer.org/proceedings/tools/0278/02780018abs.htm.

Cowell, R.G., Verrall, R.J., Yoon, Y.K., 2007. Modeling operational risk with bayesian networks. Journal of Risk and Insurance 74 (4), 795–827, doi:10.1111/j.1539-6975.2007.00235.x, http://ssrn.com/abstract=1030866.

Dhambri, K., Sahraoui, H., Poulin, P., 2008. Visual detection of design anomalies. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering, Tampere, Finland, IEEE Computer Society , pp. 279–283.

Duda, R., Hart, P., 1973. Pattern Classification and Scene Analysis. John Wiley and Sons.

Fenton, N., Neil, M., 1999. A critique of software defect prediction models. IEEE Transactions on Software Engineering 25 (5), 675–689.

Fenton, N., Neil, M., November 2007. Managing risk in the modern world—applications of bayesian networks. In: Tech. Rep. London Mathematical Society, http://www.agenarisk.com/resources/apps_bayesian_networks.pdf.

Fowler, M., 1999. Refactoring—Improving the Design of Existing Code, 1st ed. Addison–Wesley.

Guéhéneuc, Y.-G., Sahraoui, H., Farouk Zaidi, 2004. Fingerprinting design patterns. In: Stroulia, E., de Lucia, A. (Eds.), Proceedings of the 11th Working Conference on Reverse Engineering (WCRE). IEEE Computer Society Press, pp. 172–181, 10 pages, http://www-etud.iro.umontreal.ca/ptidej/Publications/Documents/WCRE04.doc.pdf.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H., 2009. A bayesian approach for the detection of code and design smells. In: Byoungju, C. (Ed.), Proceedings of the 9th International Conference on Quality Software (QSIC). IEEE Computer Society Press, p. 10, http://www-etud.iro.umontreal.ca/ptidej/Publications/Documents/QSIC09.doc.pdf.

Langelier, G., Sahraoui, H.A., Poulin, P., 2005. Visualization-based analysis of quality for large-scale software systems. In: Ellman, T., Zisma, A. (Eds.), Proceedings of the 20th International Conference on Automated Software Engineering. ACM Press, http://www.iro.umontreal.ca/labs/infographie/papers/Langelier-2005-VAQ/langelier-ase2005.pdf.

Lanza, M., Marinescu, R., 2006. Object-Oriented Metrics in Practice. Springer-Verlag, http://www.springer.com/alert/urltracking.do?id=5907042.

Mantyla, M., 2003. Bad smells in software—a taxonomy and an empirical study, Ph.D. Thesis, Helsinki University of Technology.

Marinescu, R., 2004. Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of the 20th International Conference on Software Maintenance ,. IEEE Computer Society Press, pp. 350–359.

[6] AAA http://www.ptidej.net/research/squaner/.

Moffat, A., Zobel, J., 2008. Rank-biased precision for measurement of retrieval effectiveness. ACM Transactions on Information Systems (TOIS) 27 (1), doi:10.1145/1416950.1416952, http://portal.acm.org/citation.cfm?id=1416952.

Moha, N., Guéhéneuc, Y.-G., Duchien, L., Meur, A.-F.L., 2010a. DECOR: a method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering, pp. 20–36.

Moha, N., Guéhéneuc, Y.-G., Meur, A.-F.L., Duchien, L., Alban Tiberghien, 2010b. From a domain analysis to the specification and detection of code and design smells. Formal Aspects of Computing (FAC) 22 (3–4), 345–361.

Munro, M.J., 2005. Product metrics for automatic identification of "bad smell" design problems in java source-code. In: Lanubile, F., Seaman, C. (Eds.), Proceedings of the 11th International Software Metrics Symposium. IEEE Computer Society Press, http://doi.ieeecomputersociety.org/10.1109/METRICS.2005.38.

Pearl, J., 1988. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, 1st ed. Morgan Kaufmann.

Provost, F., 2000. Machine learning from imbalanced datasets 101. In: Proceedings of AAAI Workshop on Imbalanced Data Sets , http://pages.stern.nyu.edu/fprovost/Papers/skew.PDF.

Rao, A.A., Reddy, K.N., 2008. Detecting bad smells in object oriented design using design change propagation probability matrix. In: Proceedings of the International MultiConference of Engineers and Computer Scientists.

Ricca, F., Penta, M.D., Torchiano, M., Tonella, P., Ceccato, M., Visaggio, C.A., 2008. Are fit tables really talking?: a series of experiments to understand whether fit tables are useful during evolution tasks. In: Wilhelm Schäfer, M.D., Gruhn, V. (Eds.), Proceedings of the 30th International Conference on Software Engineering. IEEE Computer Society Press, pp. 361–370.

Riel, A.J., 1996. Object-Oriented Design Heuristics. Addison–Wesley.

Simon, F., Steinbrückner, F., Lewerentz, C., 2001. Metrics based refactoring. In: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01, IEEE Computer Society) , Washington, DC, USA, p. 30.

Szolovitz, P., 1995. Uncertainty and decisions in medical informatics. Methods of Information in Medicine 34, 111–121.

Travassos, G., Shull, F., Fredericks, M., Basili, V.R., 1999. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In: Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications ,. ACM Press, pp. 47–56.

van Emden, E., Moonen, L., 2002. Java quality assurance by detecting code smells. In: Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02) ,. IEEE Computer Society Press, citeseer.ist.psu.edu/vanemden02java.html.

Wake, W.C., 2003. Refactoring Workbook. Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Webster, B.F., 1995. Pitfalls of Object Oriented Development, 1st ed. M & T Books, www.amazon.com/exec/obidos/ASIN/1558513973.

Wirfs-Brock, R., McKean, A., 2002. Object Design: Roles Responsibilities and Collaborations. Addison–Wesley Professional.

Witten, I.H., Frank, E., 1999. Data Mining: Practical Machine Learning Tools and Techniques with Implementations, 1st ed. Morgan Kaufmann.

Foutse Khomh is a Research Fellow at the Department of Electrical and Computer Engineering of Queen's University (Canada). In 2010 he received a Ph.D in Computer Science from the University of Montreal (Canada) under the supervision of Yann-Gaël Guéhéneuc. The primary focus of his Ph.D. thesis was to develop techniques and tools to assess the quality of the design and implementation of large software systems and to ensure the traceability of design choices during evolution. The final result of his thesis was a method to build quality models that take into account the quality of the design of large software systems and thus that provide both a more detailed and high-level view on quality. The quality models developed in his thesis are implemented and available online in the portal SQUANER. He also holds a Master's degree in Software Engineering from the National Advanced School of Engineering (Cameroon) and a D.E.A (Master's degree) in Mathematics from the University of Yaounde I (Cameroon). He has published several papers in international conferences and journals. He is a member of IEEE and IEEE Computer Society.

Stephane Vaucher is a member of the GEODES research team at the Université de Montréal where he completed his PhD in computer science in 2010 under the supervision of Houari Sahraoui. His thesis focused on the enhancement of software quality models using change-related information and subjective data. His interests include the detection of good and bad programming practices as well as the empirical evaluation of their effect on software quality. His industrial experience includes software design, architecture, and project management in the financial and telecommunication industry. He is a student member of the IEEE.

Yann-Gaël Guéhéneuc is associate professor at the Department of computing and software engineering of Ecole Polytechnique of Montreal where he leads the Ptidej team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, or architectural-levels. In 2009, he was awarded the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointe's supervision) since 2003 and an Engineering Diploma from École des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns. He was the first to use explanation-based constraint programming in the context of software engineering to identify occurrences of patterns. He is interested also in empirical software engineering; he uses eye-trackers to understand and to develop theories about program comprehension. He has published many papers in international conferences and journals. He is IEEE Senior Member since 2010.

Houari A. Sahraoui is Professor of Software Engineering in the Department of Computer Science and Operations Research of the University of Montreal. He obtained his Ph.D. degree in 1995 in computer science, with specialization in meta-modeling and model transformation, from the Pierre & Marie Curie University, Paris. His research interests include software visualization, object-oriented measurement and quality, and re-engineering. He has been on the program, steering, or organization committees of many international, IEEE and ACM conferences, and is member of the editorial board of three journals. He was the general chair of ASE 2003 and is the PC co-chair of VISSOFT 2011.