2009 Ninth International Conference on Quality Software

A Bayesian Approach for the Detection of Code and Design Smells

Foutse Khomh[1,2], Stéphane Vaucher[2],
Yann-Gaël Guéhéneuc[1], and Houari Sahraoui[2]

[1]Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
[2] GEODES, DIRO, Université de Montréal, Canada

E-mail: {`foutsekh,vauchers,guehene,sahraouh`}`@iro.umontreal.ca`

Abstract

The presence of code and design smells can have a severe impact on the quality of a program. Consequently, their detection and correction have drawn the attention of both researchers and practitioners who have proposed various approaches to detect code and design smells in programs. However, none of these approaches handle the inherent uncertainty of the detection process. We propose a Bayesian approach to manage this uncertainty. First, we present a systematic process to convert existing state-of-the-art detection rules into a probabilistic model. We illustrate this process by generating a model to detect occurrences of the Blob antipattern. Second, we present results of the validation of the model: we built this model on two open-source programs, GanttProject v1.10.2 and Xerces v2.7.0, and measured its accuracy. Third, we compare our model with another approach to show that it returns the same candidate classes while ordering them to minimise the quality analysts' effort. Finally, we show that when past detection results are available, our model can be calibrated using machine learning techniques to offer an improved, context-specific detection.

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 code and design smells [4]; their early detection and correction would benefit the development and maintenance processes.

Code and design smells are "bad" solutions to recurring implementation and design problems that impede the maintenance and evolution of programs. Code smells [11] are usually symptoms of larger design smells, *e.g.*, antipatterns [4]. When studying smells we do not exclude that, in a particular context, a smell could be the best way to actually implement or design a (part of a) program. For example, automatically-generated parsers are often Spaghetti Code, *i.e.*, very large classes with very long methods. Only quality analysts can evaluate their impact in their context.

The definitions of smells are often loosely specified because quality assessment is a human-centric process that requires contextual data. Consequently, there is always a degree of uncertainty on whether a class in a program is a smell or not. Therefore, detection results should be reported with a probability corresponding to the degree of uncertainty of the detection process. This uncertainty accounts for the loose definitions and the similarity of the class with the smell.

There exist many approaches to specify and detect smells. Most of these approaches are manual [28] or based on rules [2, 17, 18, 19]. Although these approaches improved the state of the art and of the practice in smell detection, to the best of our knowledge, none is able to handle the inherent uncertainty of the detection. They provide quality analysts with an unsorted set of "bad" candidates classes with no indication of which one(s) should be inspected first for confirmation and correction.

We present an approach to support uncertainty in smell detection. We based this approach on Bayesian belief networks (BBNs) to specify smells and to detect them in programs. We describe the steps required to build a BBN for the detection of a smell. The output of a BBN is a probability that a class is part of a design smell. A BBN is able to handle uncertainty by evaluating the probability that an input, *e.g.*, a large class,

1550-6002/09 $26.00 © 2009 IEEE
DOI 10.1109/QSIC.2009.47

causes an observed output, *e.g.*, the Blob antipattern. Consequently, a BBN allows quality analysts to prioritise the inspection of "bad" candidate classes. Furthermore, the Bayesian theory that underlies BBNs can 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 can thus improve the performance of a BBN for a given context, *e.g.*, organisation or type of program.

We illustrate our approach on the Blob antipattern. Moha *et al.* [18] presented precise rules to detect this antipattern. We transformed these rules into a BBN systematically and evaluated its performance on two open-source programs. We found that the BBN provides better results than the original rules. We also show that it can be calibrated using historical data both from a similar and from a different context.

Section 2 presents previous work. Section 3 gives a short background on BBNs. Section 4 describes the different steps of our approach to build a BBN for the detection of a smell from existing rules. Section 5 reports experiments on the use of the resulting BBN on GanttProject v1.10.2 and Xerces v2.7.0. It also discusses the approach and its results. Section 7 concludes with future work.

2 Related Work

Webster [31] wrote the first book on "antipatterns" in object-oriented development; his contribution covers conceptual, political, coding, and quality-assurance problems. Riel [23] defined 61 heuristics characterising good object-oriented programming to assess software quality manually and improve design and implementation. Beck [11] defined 22 code smells, suggesting where developers should apply refactorings. Mäntylä [16] and Wake [30] proposed classifications of code smells. Brown *et al.* [4] described 40 antipatterns, including the well-known Blob. These books provide in-depth views on heuristics, code smells, and antipatterns aimed at a wide academic and industrial audience. We build upon this work to propose an approach to detect smells while taking into account their loose definitions and the inherent uncertainty of the results.

Several approaches to specify and detect smells have been proposed in the literature. Manual approaches were defined, for example, by Travassos *et al.* [28], who introduced manual inspections and reading techniques to identify code smells. Marinescu [17] presented a metric-based approach to detect smells with detection strategies, which capture deviations from good design principles and consist of combining metrics with set operators and comparing their values against absolute and relative thresholds.

Similarly to Marinescu, Munro [19] proposed metric-based heuristics to detect code smells; the detection heuristics are derived from 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.

Alikacem and Sahraoui [1] proposed a language to detect violations of quality principles and smells in programs. It allows the specification of rules using metrics, inheritance and association relations among classes. They use fuzzy logic to express the thresholds of rules conditions but do not handle the uncertainty of the detection results.

Moha *et al.* [18] proposed a domain-specific language to specify smells based on a literature review of existing work. They also proposed algorithms and a platform to automatically convert specifications into detection algorithms and apply these algorithms on any program. They showed that they obtain good precision and perfect recall while allowing quality analysts to easily adapt the specifications to their context.

Some visualisation techniques [7, 24] were used to find a compromise between fully-automatic detection techniques, which are efficient but lose track of the context, and manual inspections, which are slow and subjective [14]. Other approaches perform fully-automatic detection and use visualisation techniques to present the detection results [15, 29].

All these previous approaches are based on rules that make rigid Boolean decisions as to whether a class belongs to a smell or not. These rules do not provide a probability, *i.e.*, a degree of uncertainty, that a class is or is not part of a smell.

3 Bayesian Beliefs Networks

BBNs have been successfully used to model uncertainty in fields as diverse as risk management [6], medicine [27], and computer science [9]. This success makes them an interesting choice for smell detection. We propose to use BBNs to specify and detect smells.

3.1 Definition

A BBN is a directed, acyclic graph that represents a probability distribution [20]. 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 each node X_i in the network is 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 quality analyst needs two pieces of information to build 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. This structuring can be done using heuristics found in the literature [21]. The conditional probabilities can be learned using historical data or entered directly by the analysts when data is missing. The structure ensures the qualitative validity of the approach while appropriate conditional tables (learned or entered manually) ensure that the model is well-calibrated and is quantitatively valid.

3.2 Comparison with other Techniques

There are many techniques capable of modelling uncertainty. The two most popular groups of techniques are machine learning models and statistical models. Both groups rely on the availability of historical data to correctly predict a phenomenon with certainty. However, these types of models must be trained on large amounts of tagged data to be effective (each datum describing the inputs and correct outputs). In the context of smell detection, organisations rarely keep track of past detected smells and there are no public database containing instances of smells. Consequently, these techniques are not easily and directly applicable to smell detection. Furthermore, they use black-box processes not suitable for quality analysts who want to encode their knowledge in the process.

BBNs can work with missing data and allow quality analysts to specify explicitly the decision process. When data is unavailable or must be adapted to a different context, an analyst can encode her judgement into the model. In the context of smell detection, this structuring is important because there are usually only a few instances of smells in a program; hence, a database of smell instances would be generally too small for other types of models while the literature contains many analysts' judgements on smells, which can be used to structure BBNs.

3.3 Application to Smell Detection

In the context of smell detection, the BBN nodes correspond either to an input (*e.g.*, a metric value), to a decision step if there are incoming arcs (*e.g.*, is a class part of a smell given the values of its parent nodes?), or to an output node, which is a decision node without children. The structure of the BBN encodes the analysts' judgement (and that of her peers from the literature) on the detection process.

Smell detection can be viewed as a classification problem where there are two possible outputs for a given class: $C = \{smell, not\ a\ smell\}$, given an observation $(a_1, ..., a_d)$, a vector of inputs describing the class for example. A Bayesian classifier is a BBN applied to a classification problem. 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 computed as:

$$c = \arg max_{c_k} p(c_k|a_1, \ldots, a_d),$$

where c_k ranges over the set of classes in C and the observations a_i is written as a vector of dimension d: concrete metric values, in our context. 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 class c_k, all observations are conditionally independent, the BBN structure is drastically simplified. The BBN is then called a naive Bayes classifier and the following common form of *a posteriori* probability is obtained:

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

The $p(c_k)$ marginal probability [10] is the probability that a member of a class c_k is observed. The $p(a_j|c_k)$ prior-conditional probability is the probability that the j^{th} observation assumes a particular value m_j given the class c_k. These two prior probabilities determine the structure of the naive Bayes classifier. They are learned, *i.e.*, estimated, on a training set when building the classifier.

A naive Bayes classifier is thus a simple structure [8] 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 naive Bayes classifier treats discrete and continuous observation in different ways [13]. For each discrete observation, $p(a_j|c_k)$ is a single real that represents the probability that the j^{th} observation assumes a particular value m_j when the class is c_k.

4 Approach

We now describe the conversion of previous detection rules by Moha *et al.* into the structures of BBNs. The conversion process is important because it shows that previous rules can be systematically converted into BBNs to benefit from previous expertise and to assign a probability to the results. We exemplify this process on the detection of the Blob antipattern.

Moha *et al.* [26] proposed the method DECOR. At the heart of DECOR is a domain-specific language describing code and design smells. They used this language to describe well-known smells. They also presented algorithms to parse rules and automatically generate detection algorithms [18]. They were able to identify all existing occurrences of four well-known antipatterns, Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, with 100% recall at the expense of precision, between 41% and 88% of precision, average 60%.

We chose to extend their work and transform their rules into BBNs because of their high recall. Thus, we want to achieve high precision and compare the obtained results by their precisions and the probabilities of each detected candidate class.

4.1 The Blob and its Rule Card

Listing 1 shows the specification of the Blob antipattern. The Blob is also called God class [23]. It is defined as a class that centralises functionality and has too many responsibilities. Brown *et al.* [4] characterise its structure as a large controller class that depends on data stored in several surrounding data classes.

DECOR implements the detection of the Blob by detecting classes that are controllers (ControllerClass rule), large (LargeClass rule), or weakly cohesive (LowCohesion rule), and associated to one or many data classes. Following Riel's heuristics, a controller class can be identified by its name or its method names, which must contain terms indicative of procedural programming: *Process, Control*, ... A weakly cohesive class is characterised using the LCOM5 metric [5]. Finally, a class is considered large if its number of declared methods and fields (measured using the NMD and NAD metrics) is very high *wrt.* the average size of classes in the program. Data classes are classes with more than 90% of accessor methods.

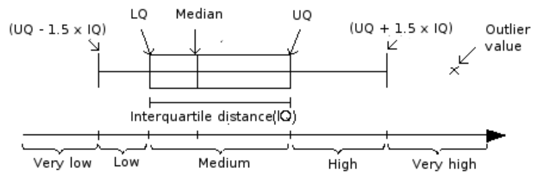


Figure 1. BoxPlot

The rule card defines three types of input properties: metric values, structural properties, and lexical properties. These properties are used to identify sets of candidate classes and, for complex smells, can be combined using set operators (intersections and unions) and using binary class relations. Structural and lexical properties evaluate to true or false to identify candidate classes. Metric values are discretised into five different levels: “very low”, “low”, “medium”, “high” and “very high”. Classes which metrics values belonging to a specified level are kept as candidate. Metric values are analysed using a box-plot to perform the discretisation. A box-plot, also know as a box-and-whisker plot, is used to single out statistical particularities of a distribution and allows for a simple identification of abnormally high or low values. Figure 1 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. The detection rules also offer some flexibility in the discretisation: the value of 20 in the LowCohesion rule indicates that values that are up to 20% below the upper outlier are equally acceptable.

4.2 Structuring BBNs from Rule Cards

We first structure the network to build a BBN using the rule card. This structuring is done in two steps. First, we transform the input properties (metrics, structural, and lexical) in the rule card into input nodes with probability distributions, and, second, we transform the set operators in the rule card into decision nodes with conditional probability tables.

4.2.1 Input Properties

For structural and lexical rules, probabilities are either 0 or 1 corresponding to the property being evaluated to true or false on a class. For metric values, the probabilities are calculated as follows: we use three groups (low, medium, high) and estimate the probability that an analyst would consider the metric value as belonging to each group. Limiting the number of groups to three

```
RULE_CARD : Blob {
   RULE : Blob { ASSOC: associated FROM: mainClass ONE TO: DataClass MANY };
   RULE : MainClass { UNION LargeClassLowCohesion ControllerClass };
   RULE : LargeClassLowCohesion { UNION LargeClass LowCohesion };
   RULE : LargeClass { (METRIC: NMD + NAD, VERY_HIGH, 0) };
   RULE : LowCohesion { (METRIC: LCOM5, VERY_HIGH, 20) };
   RULE : ControllerClass { UNION
            (SEMANTIC: METHODNAME, {Process, Control, Ctrl, Command, Cmd,
                                    Proc, UI, Manage, Drive})
            (SEMANTIC: CLASSNAME,  {Process, Control, Ctrl, Command, Cmd,
                                    Proc, UI, Manage, Drive, System, Subsystem}) };
   RULE : DataClass { (STRUCT: METHOD_ACCESSOR, 90) };
};
```

Listing 1. Specification of the Blob Antipattern

simplifies the interpretation of the detection results. What DECOR considers very low and very high values is interpreted as unambiguously low and high, *i.e.*, with a probability of 1. The probability of any other value is derived by calculating the relative distance between the value and its surrounding thresholds. The value is interpolated linearly as presented in Figure 2.

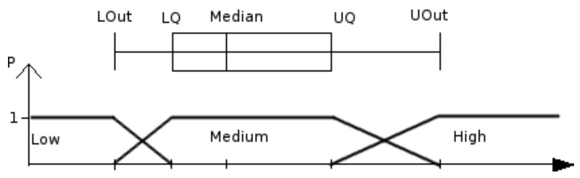
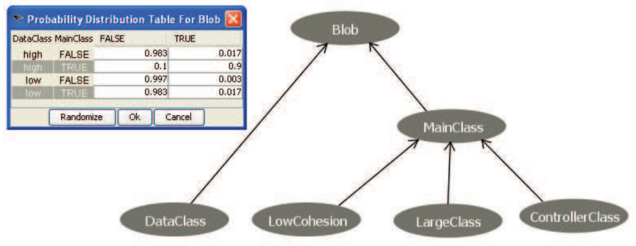


Figure 2. Metric Probabilistic Interpretation


Probability Distribution Table For Blob

DataClass	MainClass	FALSE	TRUE
high	FALSE	0.983	0.017
high	TRUE	0.1	0.9
low	FALSE	0.997	0.003
low	TRUE	0.983	0.017

Figure 3. Bayesian Network for the Blob

For structural properties, which for example determine if a class is associated to data classes, their probabilities are the number of such relations, *e.g.*, the number of data classes with which a class is associated. The rationale of these probabilities is that the more a class is associated to data classes, the more likely it is a Blob. The number of associations to data classes is evaluated using the 90% accessor ratio and is basically treated like a metric, called NoDC. To find the probability distribution of NoDC, its value is interpolated between 0 and N where N is the upper outlier value observed for NoDC in the program.

4.2.2 Set Operators

There are two operations available to combine information from the different rules: intersections and unions. For the rule card describing a Blob, a MainClass is identified as a union of classes with three different properties (high size, low cohesion, and specific name), and a Blob is the intersection of the set of MainClasses and classes associated to data classes. Set operations can be directly encoded into the BBN as decision nodes. The probability tables for these nodes will be learned from a corpus of validated data.

4.2.3 Output Node

Figure 3 presents the structure of the resulting BBN when applying the steps on the Blob rule card. For the sake of simplicity, the rule LargeClassLowCohesion was merged into the MainClass node because the union operator is associative. Such a merging could be performed systematically with any intersection operators.

DECOR outputs a binary categorisation of a class (Blob or not Blob). The output of the BBN is the probability of its output node, $p(Blob = true)$. The probability p can be used to sort classes in order of importance. This ranking could be used to guide manual inspection by allowing analysts to balance precision and recall as they see fit.

4.3 Learning BBNs Probability Tables

When there is previously classified and validated data available, a quality analyst could use Bayes' theorem to calibrate the model. The resulting BBN should be consequently superior to rule-based models, such as DECOR, because the conditional-probability table would then describe *real* smell occurrences.

If we consider the case of a MainClass, the DECOR rules state that any class that exhibits any one of three different symptoms (inputs) is a MainClass. If, however in a particular context, one of the inputs is noisy, *e.g.*, no clear cause-to-effect relation between the input and the next decision node, the detection process would produce incorrect results. By using historical data to learn the probability tables, the effect of the noisy input would be minimised, because in the past its value did not characterise known occurrences.

5 Experiments

The purpose of these experiments is to be a proof of concept demonstrating the applicability of a Bayesian approach for smell detection. Following the Goal-Question-Metric (GQM) approach [3], the *goal* of our study is to improve the quality of programs by improving the detection of smells. Our *purpose* is to provide an approach to support uncertainty in smells detection. The *quality focus* is to provide a sorted set of smell occurrences 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.

We present the results of some experiments aimed at answering the following research questions:

RQ1: To what extent a model built with a BBN based on an existing rule-based model is able detect smells in a program?

RQ2: Is a model built with a BBN better than a state-of-the-art approach, DECOR?

We used two programs on which Blobs were manually detected to form a corpus of known Blobs. Throughout the experiments, we divided the corpus in two groups: one is used to calibrate the BBN and the other is used to assess its results.

To answer RQ1, we studied the accuracy of our BBN in two scenarios. First, we assumed that there is historical data (*i.e.*, correctly identified Blobs) available for a given program. This data was used to calibrate the BBN, which was then applied on the same program. Second, we studied the accuracy of our BBN using heterogeneous data: we calibrated the BBN using known Blobs from one program and applied it on the other program. To answer RQ2, we showed that our BBN outperformed DECOR while being more flexible.

We used two open-source Java projects to perform our experiments: GanttProject v1.10.2 and Xerces v2.7.0. Table 1 summarises facts on these programs. 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 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 these programs because they are medium sized open-source projects, yet small enough to manually locate smells. All metrics and properties required to detect smells are extracted using the POM framework [12].

Systems	**♯ classes**	**KLOC**	**♯ Blobs**
GanttProjectv1.10.2	188	31	4
Xerces v2.7.0	589	240	15
Total	777	271	19

Table 1. Program Statistics

5.1 Building the Oracle

Prior to the construction of the BBN, we built a corpus of manually validated instances of the Blob antipatterns on the two programs to serve as oracle in the experiments. To the best of our knowledge, this corpus is one of the few existing and we make it available on-line[3] to help other researchers interested in smell detection.

To build the corpus, we asked two undergraduate students and two graduate students to detect occurrences of the Blob in the two programs. The pair of undergraduate students performed the task together to follow previous results [22] hinting that, on maintenance activities, the performance of a pair of undergraduate students is about the same as that of one graduate student. Prior to their manual detection of Blobs, the students were presented with several antipatterns and the Blob in particular. Then, each student/pair analysed every class of the two programs systematically to answer the boolean question: "Is this class a Blob?" We independently combined their votes such that iff at least two of the three students/pair considered a class a Blob, then we tagged it as a true occurrence. The number of detected Blobs is reported in Table 1. In both programs, only 2-3% of the classes participate in the Blob antipattern. The skewed data can have a negative effect on predictive models, we will discuss that in Section 6.3.

[1]`http://ganttproject.biz/index.php`
[2]`http://xerces.apache.org/`
[3]`http://www.ptidej.net/downloads/experiments/qsic09/`

5.2 Calibrating the BBN

In our study, we used the Bayes' theorem to calibrate the BBN. Using the known instances of Blobs, we learned the conditional probabilities using the Weka machine learning framework [32]. The framework finds, for every combination of inputs, the probabilities of a given output. To calibrate a BBN, Weka needs nominal inputs (where probabilities of inputs are 100%). We considered any input with a probability of 1 as "high" whereas any other value is "low" to obtain the nominal training dataset.

5.3 Scenario 1: Using Local Data to Calibrate a Model

In this first scenario, we studied how knowledge of previously detected Blobs in Xerces could predict the presence of Blobs in the same program. We divided the classes of Xerces v2.7.0 in three subsets with 5 instances of Blobs in each subset. Then, we trained the BBN on two of the subsets and applied it on the third subset in a 3-fold cross-validation. We did not use GanttProject because it contains too few instance of Blob.

Model	Number of Classes	Number of Blobs
BBN1	6	5
BBN2	7	5
BBN3	9	5

Table 2. Result of the 3-fold validation.

Table 2 shows the number of candidate classes that must be inspected before detecting the five known Blob. The order of inspection depends on the probability that a candidate class is a smell. It shows that if a quality analyst focuses her attention on the most probable smells first, she would waste 8% to 44% of her time (average 32%) to detect the five Blob classes.

5.4 Scenario 2: Using External Data to Calibrate the Model

In this second scenario, we assumed that a quality analyst has access to historical data from another program. She would therefore calibrate our BBN using this data and apply the BBN her other program. We show the accuracy of our BBN in this scenario by performing an inter-program validation.

We trained our BBN on GanttProject v1.10.2 and applied it on Xerces v2.7.0 and vice-versa. Figures 5 and 4 show the results: a quality analyst would focus on the n first classes returned by the BBN. The figures plot the precision (% true positives among candidates) and recall (% true positives candidates among all candidates) for every possible value of n, where the maximum value of n corresponds to the number of known Blob instances.

When analysing Xerces, Figure 4 shows that the quality analyst can quickly find the majority of Blobs: over 50% of the 15 instances of Blob are within the 13 first candidate classes. Furthermore, precision hovers around 60%. A recall of 100% is reached after inspecting 34 classes.

When analysing GanttProject, all the instances of Blobs were located among the first eight candidate classes. However, the two first candidate classes are false positives. An analysis of GanttProject shows that the use of specific terms in the names of its classes and methods, such as *Process, Control*..., has no particular link to the nature of the corresponding classes and methods.

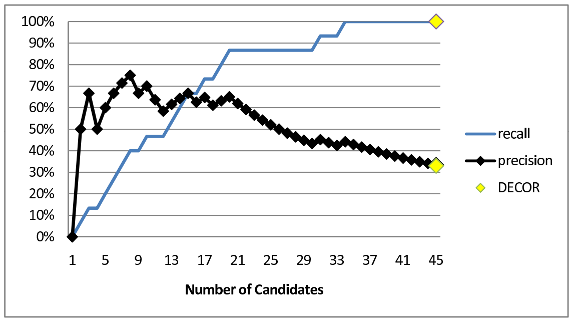
Therefore, using this knowledge, a quality analyst would adapt the BBN to these contextual particularities. To simulate and validate this adaptation, we trained our BBN on Xerces again, without the ControllerClass input node. Figure 5(b) shows the results of applying this new BBN to GanttProject. The two first candidate classes were then true positives and a recall of 100% was reached after inspecting seven candidate classes. When applying this same adaptation to Xerces, as shown in Figure 4(b), general precision was also improved.

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 be then adapted and applied in this company successfully.

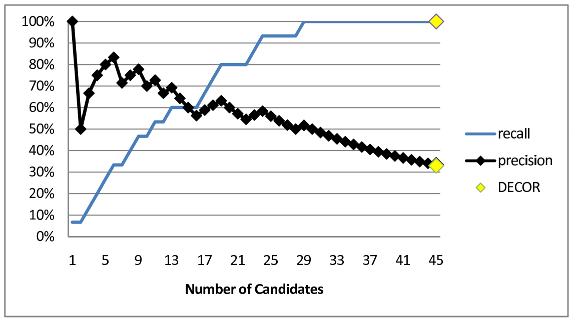
5.5 Comparison with DECOR

♯ classes	Bayesian Model			
	N_{FA}	P_{FA}	N_{TB}	P_{TB}
5	2	40%	3	20%
9	2	**22%**	6	40%
13	5	38%	8	53%
17	6	55%	11	72%
21	8	38%	13	87%
25	12	48%	13	87%
29	17	59%	13	87%
33	20	61%	14	93%
37	23	62%	15	**100%**
41	27	65%	15	100%
	DECOR			
45	30	67%	15	100%

Table 3. Comparison of the efficiency of BBN vs. DECOR on Xerces

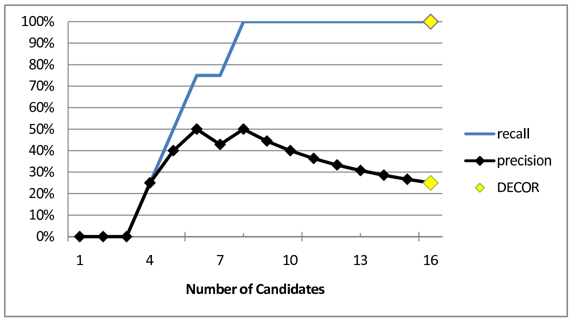


(a) Model calibrated with Gantt dataset

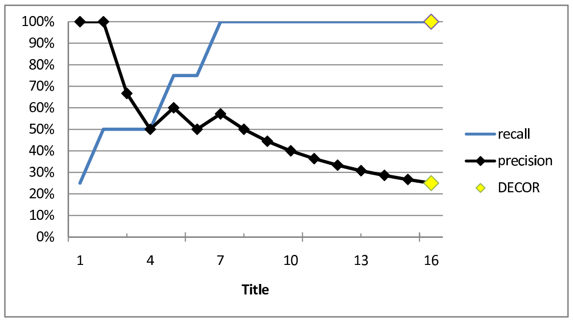


(b) Model calibrated with Gantt dataset, no Controller-Names

Figure 4. Xerces: Precision and Recall



(a) Model calibrated with Xerces dataset



(b) Model calibrated with Xerces dataset, no Controller-Names

Figure 5. Gantt: Precision and Recall

♯ classes	Bayesian Model			
	N_{FA}	P_{FA}	N_{TB}	P_{TB}
4	1	25%	3	75%
7	3	43%	4	100%
	DECOR			
16	12	75%	4	100%

Table 4. Comparison of the efficiency of BBN vs. DECOR on Gantt

The result of rule-based detection techniques is a set of candidate classes suggested to quality analysts for correction or improvement. A quality analyst would have to either validate the whole detection set or choose to inspect a subset without any indication of which classes to review first because there is no order in the results. BBNs provide a probability that a candidate class is smell which could be used to prioritise class inspections.

We use a measure applied in previous work [25] to compare the effectiveness of our BBN *wrt.* DECOR. In the following, N_{FA} and P_{FA} are respectively the average number and proportion of incorrectly classified occurrences of Blob that a quality analyst must inspect for a given number of candidate classes. An effective detection should yield a low P_{FA}. N_{TB} is the number of occurrences of Blob correctly detected and P_{TB} is the proportion of detected Blob over the total number of known Blobs.

Tables 3 and 4 present the results of the comparison of our BBN and DECOR. They show that all Blobs can be found much quicker using BBN than with DECOR. First, BBN return less candidate classes: 37 vs. 45 in Xerces. Second, in both programs, there are very few false positives among the classes returned by our BBN, *i.e.*, a majority of occurrences of Blob are detected quickly. The lowest false positive rate, 22%, occurs with the nine candidate classes detected in Xerces where there are seven Blobs correctly detected with only two false positives. These low false positive rates are to be compared to the fixed ones obtained with DECOR, GanttProject: 75% and Xerces: 67%.

We also plotted the Relative Operating Curves (ROC) comparing the true and false positives rates of our BBN with those of DECOR on both GanttProject and Xerces. We cannot show these plots for lack of space but they show that our BBN performs always better than DECOR.

We conclude that the BBN, although built on the rules provided by DECOR, has a better precision and recall and provides a means for quality analysts to quickly inspect the most serious smells.

6 Discussion

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

6.1 Using BBNs in Industrial Settings

We showed that our BBN is able to efficiently prioritise candidate classes that should be inspected by a quality analyst. The BBN, built using detection rules and calibrated using external data, can successfully detect smells. The programs were of different nature (a parser library vs. a full-fledge GUI) and developed by different development teams, yet the model outperformed DECOR. Therefore, quality analysts in industrial settings could build a repository of smells on a set of programs and use the data to build BBN and detect smells in other programs. The customisation that we performed was minimal as to serve as a proof of concept, but in an industrial setting, customisation can be much more significant.

6.2 Estimating the Number of Smells

Knowing the number of smells in a program is important to stop investigating spurious candidate classes. A well-calibrated BBN should be able to estimate the number of Blobs in a program. This number can be estimated using the expectation of the returned probabilities. Since all Blobs are considered of equivalent weight, this expectation is the sum of the probabilities divided by the number of candidate classes.

However, using the expectation to estimate the number of Blobs is inaccurate because there are many classes with low probabilities of being a Blob. Table 5 illustrates this phenomenon for both previous scenarios. For scenario 1, when adding up the probabilities of every candidate classes, the total expected Blobs is highly overestimated (from 47% to 125%). For scenario 2, the results are highly volatile due to the small number of Blobs in the programs (5 and 4, respectively).

We will investigate in future work the use of heuristics to improve the estimation of number of smells. For example, we could 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.

Xerces (int.)	Xerces (ext.)	Gantt (ext.)
10 (200%)	22 (147%)	9 (225%)

Table 5. Expected Number of Blobs

6.3 Improving Smell Detection

There are different ways a BBN could be adapted to further improved its results. The first way 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 Blobs. This is 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 high probability. Another way would be to reconsider the method used to calibrate the BBN. Weka required nominal data, and consequently, the BBN was less accurate than if it could take as input continuous values. Although the training data was unbalanced, we obtained relatively good results. These results could be better with more true Blobs added to the training set.

6.4 Application to Other Antipatterns

Our approach 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) corpora of known occurrences of the 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 corpus of Blobs is the first of such freely available corpus.

7 Conclusion

In this paper, we showed that it is possible to use BBNs to detect code and design smells in programs. We exemplified our approach on the detection of the Blob antipattern. BBNs have two main benefits because they can work with missing data and can be tuned with analysts' knowledge. Indeed, an analyst can encode her judgement into a BBN when data is unavailable or when the BBN must be adapted to a different context.

BBNs provide a theoretically-sound approach to operationalise an abstract definition of a smell as an actual detection algorithm. The definition of the smells is provided by an analyst, thus ensuring that it is qualitatively sound. Calibration is done automatically using Bayes' theorem.

Candidate classes, *i.e.*, potential smells, are associated with probabilities, which indicate the degree of uncertainty that a class is indeed a smell. They can be used to focus manual inspection by ranking classes by their probabilities.

To validate our approach, we built the BBN of the Blob antipattern based on a previous rule-based specification. Blob is a complex antipattern that requires the evaluation of different sets of classes. The rules were systematically translated into a BBN and the resulting BBN was calibrated and evaluated on two test programs, showing its high precision and recall and its capability to assign high probabilities to candidate classes that are *indeed* smells. Finally, we also showed that the result of the BBN are superior to these of DECOR in terms of precision and recall.

In future work, we plan on automatically parsing the rule cards of DECOR to support all their code and design smells. 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 study other machine learning techniques, such as support vector machine.

Acknowledgment

This work has been partly funded by NSERC, in particular the Discovery Grant #293213.

References

[1] E. Alikacem and H. Sahraoui. Détection d'anomalies utilisant un langage de description de règle de qualité. In *actes du* 12^{e} *colloque Langages, Modèles, Objets*, pages 185–200. Hermès Science Publications, mars 2006.

[2] E. H. Alikacem and H. Sahraoui. Generic metric extraction framework. In *Proceedings of the* 16^{th} *International Workshop on Software Measurement and Metrik Kongress (IWS-M/MetriKon)*, pages 383–390, 2006.

[3] R. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. In *IEEE Transactions on Software Engineering*, 10(6):728–738, November 1984.

[4] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons, 1^{st} edition, March 1998.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. Technical Report E53-315, MIT Sloan School of Management, December 1993.

[6] R. G. Cowell, R. J. Verrall, and Y. K. Yoon. Modeling operational risk with bayesian networks. In *Journal of Risk and Insurance*, 74(4):795–827, december 2007.

[7] K. Dhambri, H. Sahraoui, and P. Poulin. Visual detection of design anomalies. In *Proceedings of the* 12^{th} *European Conference on Software Maintenance and Reengineering, Tampere, Finland*, pages 279–283. IEEE Computer Society, April 2008.

[8] R. Duda and P. Hart. *Pattern classification and scene analysis.* John Wiley and Sons, 1973.

[9] N. Fenton and M. Neil. Managing Risk in the Modern World : Applications of Bayesian Networks. Technical report, London Mathematical Society, November 2007.

[10] N. Fenton and M. Neil. A critique of software defect prediction models. In *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[11] M. Fowler. *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 1^{st} edition, June 1999.

[12] Y.-G. Guéhéneuc, H. Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Proceedings of the* 11^{th} *Working Conference on Reverse Engineering (WCRE)*, pages 172–181. IEEE Computer Society Press, November 2004. 10 pages.

[13] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, 1995.

[14] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the* 20^{th} *International Conference on Automated Software Engineering.* ACM Press, November 2005.

[15] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice.* Springer-Verlag, 2006.

[16] M. Mantyla. *Bad Smells in Software - a Taxonomy and an Empirical Study.* PhD thesis, Helsinki University of Technology, 2003.

[17] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the* 20^{th} *International Conference on Software Maintenance*, pages 350–359. IEEE Computer Society Press, 2004.

[18] N. Moha, Y.-G. Guéhéneuc, A.-F. L. Meur, L. Duchien, and A. Tiberghien. From a domain analysis to the specification and detection of code and design smells. In *Formal Aspects of Computing (FAC)*, 2009. Accepted for publication. 23 pages.

[19] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *Proceedings of the* 11^{th} *International Software Metrics Symposium.* IEEE Computer Society Press, September 2005.

[20] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1 edition, September 1988.

[21] J. Pearl. *Causality: models, reasoning, and inference.* Cambridge University Press, New York, NY, USA, 2000.

[22] F. Ricca, M. D. Penta, M. Torchiano, P. Tonella, M. Ceccato, and C. A. Visaggio. Are fit tables really talking?: a series of experiments to understand whether fit tables are useful during evolution tasks. In *Proceedings of the* 30^{th} *International Conference on Software Engineering*, pages 361–370. IEEE Computer Society Press, 2008.

[23] A. J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.

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

[25] Denys Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. In *Transactions on Software Engineering (TSE)*, 33(6):420–432, June 2007. 14 pages.

[26] Naouel Moha, Y.-G. Guéhéneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In *Proceedings of the* 21^{st} *Conference on Automated Software Engineering (ASE)*, pages 297–300. IEEE Computer Society Press, September 2006. Short paper. 4 pages.

[27] P. Szolovitz. Uncertainty and decisions in medical informatics. In *Methods of Information in Medicine*, 1995.

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

[29] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02).* IEEE Computer Society Press, Oct. 2002.

[30] W. C. Wake. *Refactoring Workbook.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[31] B. F. Webster. *Pitfalls of Object Oriented Development.* M & T Books, 1^{st} edition, February 1995.

[32] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1^{st} edition, October 1999.