Software Qual J (2016) 24:947–965
DOI 10.1007/s11219-015-9271-9

On the use of design defect examples to detect model refactoring opportunities

Adnane Ghannem · Ghizlane El Boussaidi · Marouane Kessentini

Published online: 10 March 2015

Abstract Design defects are symptoms of design decay, which can lead to several maintenance problems. To detect these defects, most of existing research is based on the definition of rules that represent a combination of software metrics. These rules are sometimes not enough to detect design defects since it is difficult to find the best threshold values; the rules do not take into consideration the programming context, and it is challenging to find the best combination of metrics. As an alternative, we propose in this paper to identify design defects using a genetic algorithm based on the similarity/distance between the system under study and a set of defect examples without the need to define detection rules. We tested our approach on four open-source systems to identify three potential design defects. The results of our experiments confirm the effectiveness of the proposed approach.

Keywords Search-based software engineering · Design defects · Detection by example · Genetic algorithm

1 Introduction

Many studies reported that these software maintenance activities consume up to 90 % of the total cost of a typical software project. Model maintenance is defined as different modifications made on a model in order to improve its quality, adding new functionalities,

A. Ghannem (✉) · G. El Boussaidi
Software Engineering and IT Department, École de Technologie Supérieure, Montreal, QC, Canada
e-mail: Adnane.ghannem.1@ens.etsmtl.ca

G. El Boussaidi
e-mail: ghizlane.elboussaidi@etsmtl.ca

M. Kessentini
SBSE Research Lab, CIS Department, University of Michigan, Ann Arbor, MI, USA
e-mail: marouane@umich.edu

detecting bad designed fragments, correcting them, and modifying the model, etc. (Marinescu 2004). Due to the high cost related to model maintenance activities, automated solutions to improve model quality are a must.

To support maintenance and improve the quality of software, several approaches were proposed in the literature (e.g., Bois et al. 2004; El Boussaidi et al. 2011; Marinescu 2004; Mens et al. 2007; Moha et al. 2008; Van Der Straeten et al. 2007; Van Kempen et al. 2005; Zhang et al. 2005). Most of these approaches focus on detecting and correcting design defects. To do so, they rely on declarative rules that are manually defined; these rules are specified using metrics that embody the symptoms related to the design defect. For example, the design defect called blob (Brown et al. 1998) is characterized by symptoms like a high number of methods, attributes, and relations to many data classes. Nevertheless, there is no consensus on what makes a particular design fragment a design defect. Furthermore, for most common design defects, defining appropriate threshold values for the related metrics is not obvious. For example, a rule that detects blob classes involves metrics related to the class size (e.g., number of methods). Although we can easily measure these metrics, appropriate threshold values are not trivial to define. In addition, existing work has, for the most part, focused on detecting and correcting (refactorings) design defects at the source code level. Very few approaches tackled this problem at the model level (e.g., El Boussaidi et al. 2011; Mens et al. 2007; Zhang et al. 2005). Most of the model-based approaches are based on rules that can be expressed as assertions (i.e., invariants, pre-and post-condition) (Van Der Straeten et al. 2007; Van Kempen et al. 2005), or graph transformations targeting refactoring operations in general (e.g., (Bois et al. 2004; El Boussaidi et al. 2011)), or refactorings related to design patterns' applications (e.g., (El Boussaidi et al. 2011). However, a complete specification of defect detection and correction requires an important number of rules, and these rules must be complete, consistent, non-redundant, and correct.

In this work, we start from two main observations: (1) design defect detection rules are difficult to define and (2) they do not capitalize on defect repositories that may be available in many companies where defects in projects under development are manually identified, corrected, and documented. Based on these observations, we propose a by-example approach that exploits existing examples of defects to overcome the problems related to explicitly defining detection rules. Our approach takes as inputs an initial model and a base of defect examples and takes as controlling parameters a set of software metrics, and it generates a design defects set detected in the initial model. To this end, we used a population-based meta-heuristic search based on genetic algorithms (GA) (Goldberg 1989). In the context of this paper, we focus on detecting defects in UML class diagrams. Our approach is evaluated on four large open-source systems and aimed at investigating to what extent the use of the base of examples of design defects improves the automation of detection.

The primary contributions of the paper can be summarized as follows:

- We introduce a detection approach based on the use of design defect examples. Our proposal does not require an explicit definition of detection rules, and thus, it does not require a specification of the metrics to use or their related threshold values.
- We report the results of an evaluation of our approach; we used design defect examples extracted from four object-oriented open-source projects. We applied a fourfold cross-validation procedure. For each fold, one open-source project is evaluated by using the remaining three systems as bases of examples. The average values of precision and recall computed from 31 executions on each project are 95 and 76 %, respectively,

which allows us to say that the obtained results are promising. The effectiveness of our approach is also assessed using a comparative study between our approach and two other approaches.

The remainder of this paper develops our proposals and details how they are achieved. Therefore, the paper is structured as follows. Section 2 is dedicated to the background and problem statement related to our approach. Section 3 presents the overall approach and the details of our adaptation of the GA to the problem of detecting design defects in UML class diagrams. Section 4 reports on the experimental settings and results. Related works are discussed in Sect. 5, and we conclude and outline some future directions to our work in Sect. 6.

2 Background and problem statement

2.1 Design defects

We focus in this paper on the detection of a specific type of design defect to improve model quality. Design defects, also called design anomalies, refer to design situations that adversely affect the development of models (Brown et al. 1998). Different types of defects, presenting a variety of symptoms, have been studied in the intent of facilitating their detection and suggesting improvement solutions. In Fowler and Beck (1999), they define a set of symptoms of common defects. These include large classes, feature envy, long parameter lists, and lazy classes. Each defect type is accompanied by refactoring

Table 1 Considered metrics in our approach

Ref	Metric description
NA	The total number of attributes per class
NPvA	The total number of private attributes per class
NPbA	The total number of public attributes per class
NProtA	The total number of protected attributes per class
NM	The total number of methods per class
NPvM	The total number of private methods per
NPbM	The total number of public methods per class
NPrtM	The total number of protected methods per class
NAss	The total number of associations
NAgg	The total number of aggregation relationships
NDep	The total number of dependency relationships
NGen	The total number of generalization relationships (each parent–child pair in a generalization relationship)
NAggH	The total number of aggregation hierarchies
NGenH	The total number of generalization hierarchies
DIT	The DIT value for a class within a generalization hierarchy is the longest path from the class to the root of the hierarchy
HAgg	The HAgg value for a class within an aggregation hierarchy is the longest path from the class to the leaves

suggestions to correct the defect. Brown et al. (1998) define another category of design defects that are documented in the literature and named anti-patterns.

In our approach, we focus on the detection of some defects that can appear at the model level and especially in class diagrams. We choose from Brown et al. (1998) three important defects that can be detected in class diagrams:

1. Blob, which is found in designs where one large class monopolizes the behavior of a system (or part of it), and other classes primarily encapsulate data.
2. Functional decomposition (FD): It occurs when a class is designed with the intent of performing a single function. This is found in class diagrams produced by non-experienced object-oriented developers.
3. Data class (DC): It encapsulates only data. The only methods that are defined by this class are the getters and the setters.

2.2 Software metrics

Software metrics provide useful information that helps assessing the level of conformance of a software system to a desired quality such as *evolvability* and *reusability* (Fenton and Pfleeger 1998). Metrics can also help detecting some similarities between software systems. The most widely used metrics for class diagrams are the ones defined by Genero et al. (2002). In the context of our approach, we used the eleven (11) metrics defined by Genero et al. (2002) to which we have added a set of simple metrics (e.g., number of private methods in a class, number of public methods in a class) that we have defined for our needs. The metrics configuration for the experiments reported here consisted of the sixteen software metrics described below in Table 1. All these metrics are related to the class entity, which is the main entity in a class diagram. Some of these metrics represent statistical information (e.g., number of methods, attributes, etc.), and others give information about the position of the class through its relationships with the other classes of the model (e.g., number of associations). All these metrics have a strong link with the design defects presented in the previous section.

These metrics considered by our approach will not be used to evaluate the quality of the model fragments, but to calculate the structural similarity between the models to evaluate and the base of examples. We selected these metrics based on previous work on the estimation of the similarity between models (Ben Fadhel et al. 2012).

2.3 Problem statement

A tool supporting the detection and correction of design defects at the model level may be of great value for novice designers as well as experimented ones when refactoring existing models. However, there are many open challenging issues that we must address when building such a tool. Some of these open issues were introduced by Kessentini et al. (2011). We summarize these issues in the following.

Most of existing studies focus on detecting design defects at the code level and not the model one. However, it is important to fix these defects as early as possible; thus, it will be useful to detect defects at the design step of the project. In addition, the detection of defects at the model level is more challenging since not all metrics applied at the code level can be used at the model one. Furthermore, it is easier for developers to identify, understand, and fix defects detected at the model level.

In the current state of art, there is no consensus on what makes a particular design fragment a bad design. Even if we detect some design form that we defined as "suspicious," we cannot say for sure that it is a defect (El Boussaidi et al. 2011). Asserting that a suspicious design fragment is actually a design defect depends on the context. For example, a "Log" class responsible for maintaining a log of events, used by a large number of classes, is a common and acceptable practice. However, from a strict defect definition, it can be considered as a class with abnormally large coupling. Furthermore, even for the design defects that are commonly recognized in the literature such as the blob, deciding which classes are blob candidates depends on the designer's interpretation. This also depends on the detection thresholds set by the designer when dealing with quantitative information. For example, the blob detection involves information such as class size. Although we can measure the size of a class, an appropriate threshold value is not trivial to define. A class considered large in a given context could be considered average in another. Another issue is related to the usefulness of detecting and returning long lists of defect candidates. In these cases, a designer needs to assess the defect candidates, select true-positives that must be fixed, and reject false-positives. This can be a fastidious task and not always profitable. In addition to these issues, manually defining the rules that detect all targeted design defects can be a time-consuming and an error-prone process. Finally, it is difficult to generalize the detection rules from a set of defect examples. In fact, the same type of defects can have different possible structures. The rule-based approaches try to maximize the coverage of defect examples; however, in our approach, we are looking mainly into the similarity scores between the input (model to evaluate) and the different examples. Thus, we can consider situations where there is a similarity between part of the model to evaluate and few examples. However, the rule-based approaches try to mainly consider the "common" similarities between the examples characterizing the same type of defect. Therefore, we argue that it is more efficient to rely on similarities between the software under analysis and existing defect examples to detect design defects in this software. This idea is the foundation of the approach proposed in this paper.

Fig. 1 High-level pseudo-code for GA adaptation to our problem

```
Input: Initial Model (IM)
Input: Set of quality metrics
Input: Set of Design Defects examples
Output: A set of Design Defects (DD) detected in IM
1: I:= set_of (CIM, CBE, DD detected on  CBE)
2: P:= set_of (I)
3: initial_population (P, Max_size)
4: Repeat
5:    for all I P do
7:       Fitness(I) : = {f1(I) + f2(I)}/2
8:    end for
9:    best_solution := best_fitness(I);
10:  P := generate_new_population(P)
11:  it:=it+1;
12: Until it = max_it or fitness(best_solution) =1;
13: return best_solution
```

3 A search-based approach to detecting design defects

The approach proposed in this paper exploits examples of design defects and a heuristic search technique to automatically detect design defects on a given model and specifically in class diagrams. Our detection approach takes as inputs an initial model and a base (i.e., a set) of defect examples and takes as controlling parameters a set of software metrics. These metrics were presented above in Table 1, and their expressiveness and usefulness were discussed in the literature (Genero et al. 2002). The approach generates a set of design defects detected in the initial model. In the following subsection, we describe in details how we encoded the design defect detection problem using the GA (Koza 1992).

3.1 Adaptation of the genetic algorithm to design defect detection

GA is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution (Koza 1992). A high-level view of our adaptation of GA to the design defect detection problem is given in Fig. 1. As this figure shows, the algorithm takes as input an initial model, a set of software metrics, and a set of design defects examples. The output is the set of design defects that were detected in the initial model.

Lines 1–3 construct an initial population, which is a set of individuals that stand for possible solutions, representing a set of design defects that may be detected in the classes of the initial model. An individual is a set of triplets; a triplet is called a block, and it contains a class of the initial model denoted as CIM, a class of the base of examples denoted as CBE, and a design defect (DD) detected in the CBE. To generate an initial population, we start by defining the maximum individual size in terms of a maximum number of blocks composing an individual. This parameter can be specified either by the user or randomly. Thus, the individuals have different sizes. Then, for each individual, the blocks are randomly built, i.e., a block is composed by the triplet (CIM, CBE, DD), where a class (CIM) from the initial model is randomly matched to a class (CBE) from the base of examples and a design defect (DD) present in the CBE.

Individuals' representation is explained in more detail in Sect. 3.2. Lines 4–12 encode the main GA loop, which explores the search space and constructs new individuals by changing the matched pairs (CIM, CBE) in blocks. During each iteration, we evaluate the quality of each individual in the population. To do so, we use a fitness function defined as an average of two functions f1 and f2. f1 computes the similarities between the classes CMI and CBE of each block composing the individual, while f2 computes the ratio of the individual size by the maximum individual size (line 7). Computation of these two functions and the fitness function of an individual is described in more detail in Sect. 3.4. Then we save the individual having the best fitness (line 9). In line 10, we generate a new population (p + 1) of individuals from the current population by selecting 50 % of the best fitted individuals from population p and generating the other 50 % of the new population by applying the crossover operator to the selected individuals, i.e., each pair of selected individuals, called parents, produces two children (new solutions). Then we apply the mutation operator, with a probability, for both parents and children to ensure the solution diversity; this produces the population for the next generation. The mutation probability specifies how often parts of an individual will mutate. Selection, crossover, and mutation are described in details in Sect. 3.3.

The algorithm stops when the termination criterion is met (Line 12) and returns the best solution found during all iterations (Line 13). The termination criteria can be a maximum number of iterations or the best fitness function value. However, the best fitness function value is difficult to predict, and sometimes, it takes very long time to converge toward this value. Hence, our algorithm is set to stop when it reaches the maximum iteration number or the best fitness function value. In the following subsections, we describe in details our adaption of GA to the design defect detection problem.

3.2 Individual representation

An individual is a set of blocks. A block contains three parts as shown in Fig. 2: The first part contains the class CIM chosen from the initial model (model under analysis), the second part contains the class CBE from the base of examples that was matched to CIM, and finally, the third part contains the design defect detected on CBE. An example of a solution (i.e., an individual) is given in Fig. 3.

3.3 Genetic operators

3.3.1 Selection

We used the stochastic universal sampling (SUS) (Koza 1992) to select individuals that will undergo the crossover and mutation operators to produce a new population from the current one. In the SUS, the probability of selecting an individual is directly proportional to its relative fitness in the population. For each iteration, we use SUS to select 50 % of individuals from population p for the new population p + 1. These (population_size/2) selected individuals will be transmitted from the current generation to the new generation, and they will "give birth" to other (population_size/2) new individuals using crossover operator.

3.3.2 Crossover

For each crossover, two individuals are selected by applying the *SUS* selection (Koza, 1992). Even though individuals are selected, the crossover happens only with a certain probability. The crossover operator allows creating two offspring P'_1 and P'_2 from the two selected parents P_1 and P_2. It is defined as follows: A random position, k, is selected. The first k blocks of P_1 become the first k blocks of P'_2. Similarly, the first k blocks of P_2 become the first k blocks of P'_1. The rest of blocks (from position k + 1 until the end of the set) in each parent P_1 and P_2 are kept. For instance, Fig. 4 illustrates the crossover operator applied to two individuals (parents) P_1 and P_2. The position k takes the value 2. The first two blocks of P_1 become the first two blocks of P'_2. Similarly, the first two blocks of P_2 become the first *two* blocks of P'_1.

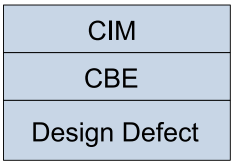
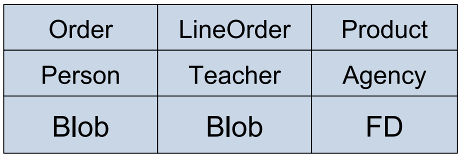


Fig. 2 Block representation



Order	LineOrder	Product
Person	Teacher	Agency
Blob	Blob	FD

Fig. 3 Individual representation

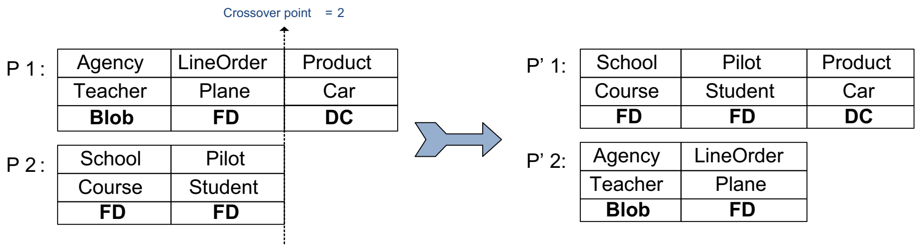


Fig. 4 Crossover operator

3.3.3 Mutation

The mutation operator consists of randomly changing one or more dimensions (i.e., blocks) in the solution. Hence, given a selected individual, the mutation operator first randomly selects some blocks in the individual. Then the CBE of the selected block is replaced by another CBE chosen randomly from the base of examples. Figure 5 illustrates the effect of a mutation that replaced the design defect blob detected in the class *Teacher* (initial model), which is extracted from the class *Agency* (base of examples) by the design defect data class (DC) extracted from the new matched class *Taxes* (base of examples).

3.4 Fitness function

The fitness function quantifies the quality of the generated individuals. The challenge is to define an efficient and simple fitness function in order to reduce the computational complexity. In our context, we want to exploit the similarities between the model under analysis and other existing models to infer the design defect that we must correct. Our intuition is that a candidate solution that displays a high similarity between the classes of the initial model and those chosen from the examples base should give the most accurate set of design defects. Hence, the fitness function aims to maximize the similarity between the classes of the model in comparison with the ones in the base of examples. In this context, we introduce first a similarity measure between two classes denoted by *Similarity* and defined by formulae 1 and 2.

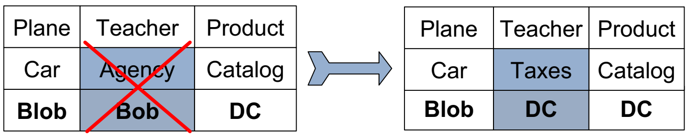


Fig. 5 Mutation operator

$$\text{Similarity }(\text{CIM}, \text{CBE}) = \frac{1}{m}\sum_{i=1}^{m} \text{Sim}(\text{CIM}_i, \text{CBE}_i) \tag{1}$$

$$\text{Sim }(\text{CIM}_i, \text{CBE}_i) = \begin{cases} 1 \quad \text{if CIM}_i = \text{CBE}_i \\ 0 \quad \text{if } (\text{CIM}_i = 0 \text{ and } \text{CBE}_i \neq 0) \text{ or } (\text{CIM}_i \neq 0 \text{ and CBE}_i = 0) \\ \dfrac{\text{CIM}_i}{\text{CBE}_i} \quad \text{if CIM}_i < \text{CBE}_i \\ \dfrac{\text{CBE}_i}{\text{CIM}_i} \quad \text{if CBE}_i < \text{CIM}_i \end{cases} \tag{2}$$

Where m is the number of metrics considered in this project. CIM_i is the ith metric value of the class CIM in the initial model, while CBE_i is the ith metric value of the class CBE in the base of examples. Using the similarity between classes, we define the first component (f1) of the fitness function of a solution defined by the formula 3. We also add a second component (f2) of the fitness function to ensure the completeness of the solution defined by the formula 4, i.e., f2 takes into consideration the size of an individual in terms of the number of blocks compared with the maximum individual size. As discussed in Sect. 3.1, the maximum individual size represents the maximum number of blocks composing an individual, and it can be specified either by the user or randomly.

$$f_1 = \frac{1}{n}\sum_{j=1}^{n} \text{Similarity}(\text{CIM}_{Bj}, \text{CBE}_{Bj}) \tag{3}$$

$$f_2 = \frac{\text{Individual size}}{\text{Maximum individual size}} \tag{4}$$

Where n is the number of blocks in the solution, and CIM_Bj and CBE_Bj are the classes composing the first two parts of the jth block of the solution. Finally, we define the fitness function of a solution, normalized in the range [0, 1], as denoted by the formula 5.

$$ff = \frac{f_1 + f_2}{2} \tag{5}$$

To illustrate how the fitness function is computed, consider as an example an individual I composed by two blocks. The first block matches the class Plane from the initial model to the class Catalog from the base of examples, while the second block matches the class Car from the initial model to the class Agency from the base of examples. In this example, the maximum individual size is set to 10 and we use five metrics. The values of these metrics are given for the classes composing the individual I in Table 2 (for classes from the initial model) and Table 3 (for classes from the base of examples).

The fitness function of I is calculated as follows:

Table 2 Classes from the initial model and their metrics values

CMI	NPvA	NPbA	NPvM	NAss	NGen
Plane	4	1	1	1	1
Car	2	2	0	1	0

$$f_{1_I} = \frac{1}{2}\left[\frac{1}{5}\left[\left(\frac{4}{5}+1+0+1+0\right)+\left(1+\frac{1}{2}+1+\frac{1}{3}+1\right)\right]\right] = 0.66$$

$$f_{2_I} = \frac{2}{10} = 0.2$$

$$ff = \frac{f_{1_I}+f_{2_I}}{2} = \frac{0.66+0.2}{2} = 0.43$$

4 Validation of the approach

We implemented and tested the approach on four open-source projects. In this section, we describe our experimental setup and we present the results of our experiment. We specifically discuss the results of our GA algorithm in terms of precision and recall.

4.1 Research questions

The goal of the experiment was to evaluate the efficiency of our approach for the detection of design defects in UML class diagrams. Specifically, the experiment aimed at answering the following research questions:

RQ1 To what extent can the proposed approach detect design defects?
RQ2 What types of defects does it locate correctly?

To answer RQ1, we used an existing corpus of known design defects to evaluate the precision and recall of our approach. In the context of our study, the precision denotes the fraction of true design defects among the set of all detected defects. The recall indicates the fraction of correctly detected design defects among the set of expected defects (i.e., how many defects have not been missed) (see the formula 6 and 7).

$$\text{Precision} = \frac{\text{True design defects}}{\text{All detected design defects}} \tag{6}$$

$$\text{Recall} = \frac{\text{True design defects}}{\text{Expected defects}} \tag{7}$$

In general, the precision denotes the correctness of the approach (i.e., the probability that a detected defect is a true defect), and the recall denotes the completeness of the approach (i.e., the probability that an actually defect is detected). To answer RQ2, we investigated the type of defects that were detected by our approach.

Table 3 Classes from the base of examples and their metrics values

CBE	NPvA	NPbA	NPvM	NAss	NGen
Agency	2	1	0	3	0
Catalog	5	1	0	1	0

4.2 Experimental setup

We implemented our approach as an Eclipse plugin that takes as input the model under analysis, a list of metrics and a base of examples. It generates as output the optimal solution, i.e., a set of design defects detected in the analyzed model.

We used four open-source Java projects to perform our experiments:

- GanttProject (v0.10.2): A Java project that supports project management and scheduling.
- LOG4J v1.2.1: A logging library for Java.
- ArgoUML v0.18.1 An UML diagramming application written in Java.
- Xerces (v2.5): A set of parsers compatible with XML.

In fact, we selected these systems for our validation because they range from medium to large-sized open-source projects that have been actively developed over the past 10 years and include a large number of code-smells. In addition, these systems are well studied in the literature, and their code-smells have been detected and analyzed manually. We selected these three types of defects because they are the most studied ones in existing literature (data available) and the most important and frequent ones based on recent papers/studies in the area of defects detection and correction. In addition, the selected types of code-smells in our validation are not similar (unilateral), diversified, and cover different high-level design quality (maintainability) attributes such as reusability, flexibility, understandability, functionality, extendibility, and effectiveness. However, the proposed approach is generic and can be extend to other types of defects by adding new examples.

Table 4 provides some relevant information about these projects including the number of classes and the number of design defects (i.e., blob, functional decomposition and data class) existing in these projects. We used our parser to generate predicate models from the four selected projects. To build the base of examples, we completed the generated models by manually entering the design defects that were detected by related work (Kessentini et al. 2010; Moha et al. 2008) in these projects. In Moha et al. 2008, we asked three groups of students to analyze the libraries to tag instances of specific code-smells to validate their detection technique, DECOR. For replication purposes, they provided a corpus of describing instances of different code-smells. In Kessentini's previous work (Kessentini et al. 2011), they asked eighteen graduate students and four software engineers to extend the existing corpus proposed by Moha et al. To calculate the recall, the different participants analyzed different quality metrics based on the definition of code-smells. Based on this corpus, we were able to calculate the precision and recall scores. We used a fourfold cross-validation procedure. For each fold, one open-source project is evaluated by using the remaining three projects as base of examples. For example, Gantt is analyzed using

Table 4 Case study setting

Project	Number of classes	Number of blobs	Number of FDs	Number of DCs
GanttProject v1.10.2	245	10	17	10
LOG4J v1.2.1	227	3	11	5
ArgoUML v0.18.1	1267	29	37	41
Xerces v2.7	676	44	29	58

LOG4J, ArgoUML, and Xerces as base of examples and vice versa. We also performed multiple executions of the approach on each of the 4 projects to ensure that the results of our approach are stable.

We report the number of defects detected, the number of true-positives, the recall (number of true-positives over the number of design defects), and the precision (ratio of true-positives over the number detected); we determined the values of these indicators when using our algorithm for every defect in the four open-source projects.

To set the parameters of GA for the search strategies, we performed several tests and the final parameters' values were set to a minimum of 1000 iterations for the stopping criterion, to 30 as population size, to 2 as the minimum length of a solution in terms of number of block, and to size of the initial model as the maximum length of a solution. We also set the crossover probability to 0.9 and the mutation probability to 0.5. These values were obtained by trial-and-error method. We selected a high mutation rate because it allows the continuous diversification of the population, which discourages premature convergence to occur.

Finally, since we viewed the design defects detection problem as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. We executed our algorithm on a standard desktop computer (Intel i7 CPU running at 2.67 GHz with 8 GB of RAM). The execution time for detection defects with a number of iterations 1000 was <1 min. This indicates that our approach is reasonably scalable from the performance standpoint. However, the execution time depends on the number of used metrics and the size of the base of examples.

4.3 Results and discussion

Figure 6 shows the precision results of multiple executions (31) of the approach for all the projects. The averages of precision are 93, 100, 94, and 94 % for Gantt, Log4J, ARGOUML, and Xerces, respectively. Similarly, Fig. 7 shows the recall results of the 31 executions for all the projects. The averages of the recall are 76, 76, 75, 76 % for Gantt, Log4J, ARGOUML, and Xerces, respectively. Generally, the high precision and recall average allow us to positively answer our first research question RQ1. Indeed, the precision average which is very high for all the projects (close to 100 %) proves that all the design

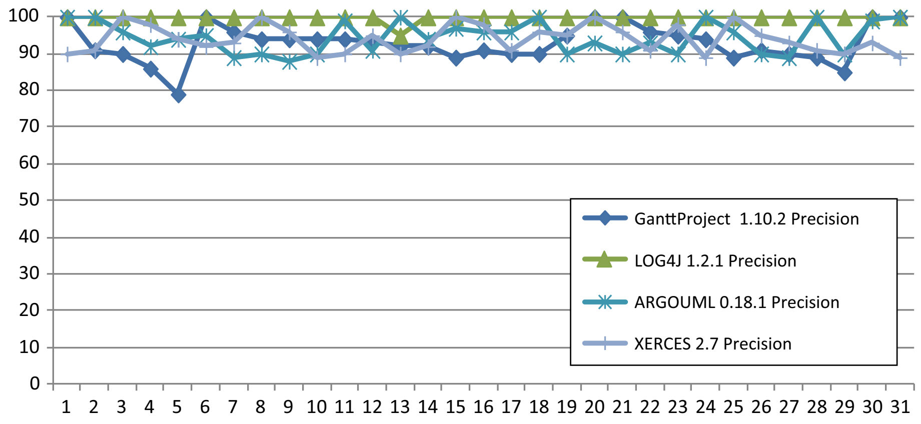


Fig. 6 Precision results of 4 multiple executions of our approach on the analyzed projects

defects detected by our approach were true existing defects in the analyzed projects (Fig. 8).

To further analyze the effectiveness of our approach, we conducted a comparative study between the results of our approach and the results of two other approaches, i.e., the approach proposed by Moha et al. called DECOR (Moha et al. 2008) and our previous work presented in Ghannem et al. (2011). We chose these two approaches because they analyzed the same projects and they considered two of the design defects that we are targeting in our experiments, namely the blob and functional decomposition (FD). We also considered these two approaches because one of them relies on a search-based approach, while the other does not. Indeed, DÉCOR (Moha et al. 2008) relies on the description of design defects to generate algorithms that detect these defects, while the approach in Ghannem et al. (2011) uses a search-based approach to automatically generate detection rules based on defects examples. Table 5 summarizes the results of our approach and the two other approaches in terms of precision for each of the blob and FD defects. The DC (data class) defect was not considered by the other two approaches. We noticed that, in general, our proposal has better precision values than those given by the two approaches for the 4 analyzed projects. Overall, the three approaches have very good results for the blob defect. However, for the FD defect, the precision of our approach is quite high compared with DECOR's precision: The gap reaches nearly double for Log4J and Xerces projects (i.e., 100 vs. 54.5 % and 90 vs. 51.7 %, respectively) and more than three times for GanttProject (88 vs. 26.7 %). On the other hand, our previous work (Ghannem et al. 2011) which is search based is more competitive than our current approach. These results can be explained by the fact that the FD defect is very difficult to describe in terms of rules and metrics, and a by-example approach is much more effective in this case.

To answer our second research question, we compared the results of our approach when applied to each of the three design defects we considered in this paper. Table 6 displays the average number of detected defects for each project and each defect, i.e., the values in Table 6 correspond to the average value of the recall per defect and project. We can notice that overall the majority of expected design defects are detected by our approach. For example, the recall varies from 86 to 100 % for the blob defect. Even in the case of the FD defect, the recall varies from 64 to 78 % with an average of 70 %. Accordingly, we can conclude that our approach is able to detect design defects regardless of their type. It is

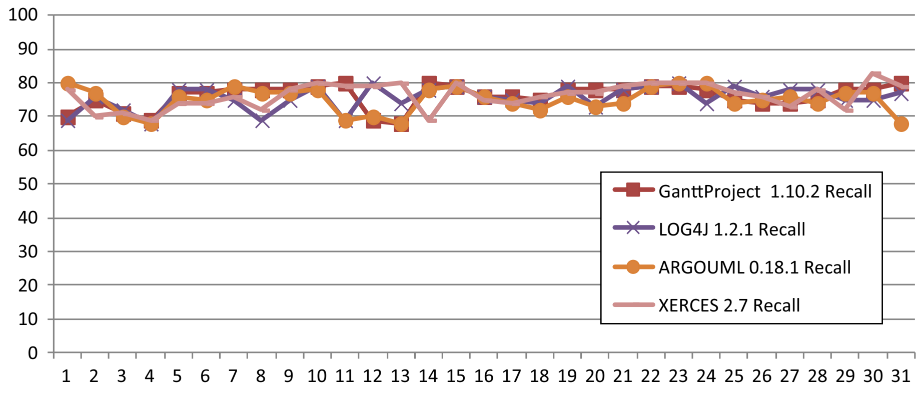


Fig. 7 Recall results of multiple execution of our approach on the analyzed projects

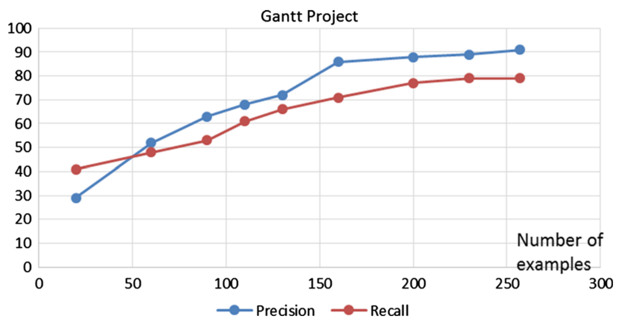


Fig. 8 The impact of the number of examples on the quality of the results of Gantt Project

Table 5 Comparison of the detection precision results of our approach to two other approaches

Project	Design defect	Precision of our approach (%)	Precision of Ghannem et al. (2011) (%)	Precision of DÉCOR Moha et al. (2008) (%)
GanttProject v1.10.2	Blob	100	100	90
	FD	88	88	26.7
LOG4J v1.2.1	Blob	100	66	100
	FD	100	82	54.5
ArgoUML v0.18.1	Blob	100	93	86.2
	FD	86	82	59.5
Xerces v2.7	Blob	100	97	88.6
	FD	90	88	51.7

worth pointing out that, as for all example-based approaches, the results of our technique depend largely on the quality of the base of examples.

The results are consistent since most of existing design defects symptoms are detected more using structural information/metrics rather than semantics ones. Thus, the detection results can be consistent even if the projects are different semantically (implementing different features). In addition, the different systems have similar sizes, and the base of examples is well diversified to cover several examples of different types of defects.

Table 6 Detected defects by our approach for each of the studied defects

Project	Average number of detected blobs	Average number of detected FD	Average number of detected DC
GanttProject v1.10.2	90 % (9/10)	70 % (12/17)	80 % (8/10)
LOG4J v1.2.1	100 % (3/3)	64 % (7/11)	80 % (4/5)
ArgoUML v0.18.1	86 % (25/29)	78 % (29/37)	88 % (36/41)
Xerces v2.7	91 % (40/44)	69 % (20/29)	86 % (50/58)

The reliability of the proposed approach requires an example set of code-smell examples. It can be argued that constituting such a set might require more work than identifying and adapting code-smell detection rules. In our study, we showed that by using open-source system defect examples directly, without any adaptation, our method can be used out of the box, and this will produce good detection results for the detection of code-smells for the studied systems. Figures 7 shows that around 200 code-smell examples can be used to obtain good precision and recall scores based on the Gantt Project.

4.4 Threats to validity

There are four types of threats that can affect the validity of our experiments. We consider each of these in the following paragraphs.

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We used the Wilcoxon rank sum test with a 95 % confidence level to test whether significant differences exist between the measurements for different treatments. This test makes no assumption that the data are normally distributed and is suitable for ordinal data, so we can be confident that the statistical relationships we observed are significant. In our comparison with the techniques not based on heuristic search, we considered the parameters provided with the tools. This can be considered as a threat that can be addressed in the future by evaluating the impact of different parameters on the quality of the results of DECOR.

Internal validity is concerned with the causal relationship between the treatment and the outcome. We consider the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on several independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95 % confidence level ($\alpha = 5$ %). However, the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error method, which is commonly used in the SBSE community. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution in order to provide the best possible performance.

Construct validity is concerned with the relationship between theory and what is observed. Most of what we measure in our experiments are standard metrics such as precision and recall that are widely accepted as good proxies for quality of code-smell detection solutions. In our future work, we plan to compare the performance of our approach using different meta-heuristics search algorithms. Another threat to construct validity arises because, although we considered 3 types of code-smells, we did not evaluate the detection of other types of code-smells. In future work, we plan to evaluate the performance of our proposal to detect some other types of code-smell. Another construct threat can be related to the corpus of manually detected code-smells since developers do not all agree whether a candidate is a code-smell or not. We will ask some new experts to extend the existing corpus and provide additional feedback regarding the detected code-smells.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on different widely used open-source systems belonging to different domains and with different sizes, as described in Table 4. However, we cannot assert that our results can be generalized to industrial Java applications, other programming languages, and to other practitioners. Future replications of this study are necessary to confirm our findings.

5 Related work

The proposal in this paper is related to work on detecting design defects in existing software. Existing work could be classified into two broad categories: non-search-based techniques and search-based techniques for detecting design defects. Most of the approaches in the first category are based on rule specification. Erni and Lewerentz (1996) use metrics to evaluate frameworks with the goal of improving them. They introduce the concept of multi-metrics, n-tuples of metrics expressing a quality criterion (e.g., modularity). Marinescu (2004) defined a list of rules relying on metrics to detect what he calls design flaws of OO design at method, class, and subsystem levels. The main limitation of these approaches is the difficulty to manually define threshold values for metrics in the rules. To circumvent this problem, Alikacem and Sahraoui (2006) express defect detection as fuzzy rules, with fuzzy labels for metrics, e.g., small, medium, large. When evaluating the rules, actual metric values are mapped to truth values for the labels by means of membership functions. Although no thresholds need to be defined, still, it is not obvious to determine the membership functions. Moha et al. (2008), in their DECOR approach, start by describing defect symptoms using an abstract rule language. These descriptions involve different notions, such as class roles and structures. The descriptions are later mapped to detection algorithms. In addition to the threshold problem, this approach uses heuristics to approximate some notions, which results in an important rate of false-positives. Khomh et al. (2009) extended DECOR to support uncertainty and to sort the defect candidates accordingly. Uncertainty is managed by Bayesian belief networks that implement the detection rules of DECOR. The detection outputs are probabilities that a class is an occurrence of a defect type. Away from rule-based metric, Budi et al. (2011) present a framework to detect design flaws in layered object-oriented architecture based on stereotypes. They tend to detect design rule violations associated with stereotypes and not real design defects.

Our approach is inspired by the approaches in the second category of work, which uses search-based techniques to suggest refactorings (e.g., Harman and Tratt 2007; Jensen and Cheng 2010; Kessentini et al. 2008; O'Keeffe 2008; Seng et al. 2006). In these approaches, design defects are not detected explicitly as the focus is put on detecting elements to change to improve the global quality. For example, a heuristic-based approach is presented in Harman and Tratt (2007; O'Keeffe 2008; Seng et al. 2006) in which various software metrics are used as indicators for the need of a certain refactoring. In Seng et al. (2006), a GA is used to suggest refactorings to improve the class structure of a system. The algorithm uses a fitness function that relies on a set of existing object-oriented metrics. Harman and Tratt (2007) propose to use the Pareto optimality concept to improve search-based refactoring approaches when the evaluation function is based on a weighted sum of metrics. Both the approaches in Seng et al. (2006) and Harman and Tratt (2007) were limited to the move method refactoring operation. In O'Keeffe (2008), the authors present a comparative study of four heuristic search techniques applied to the refactoring problem. The fitness function used in this study was based on a set of 11 metrics. The results of the experiments on five open-source systems showed that hill-climbing performs better than the other algorithms. In Kessentini et al. (2010) and Kessentini et al. (2014), Kessentini et al. proposed an approach, based on search-based techniques, for the automatic detection of potential design defects in code. The detection is based on the notion that the more the code deviates from good practices, the more likely it is bad. In both Ghannem et al. (2011) and Ouni et al. (2013), a search-based approach is used to generate rules that detect design defects in existing code. Contrary to these two approaches, our current proposal does not generate detection rules; it uses defect examples to identify potential defects. Also, the results from our experiments proved that our current proposal yields better results than our previous work. Recently, Palomba et al. (2014) used the history of changes to detect code-smells instead of the use of quality metrics. For example, classes that change frequently are considered as a blob. A comparison with DECOR confirms the outperformance of their approach and the benefits of the use of the history of changes for the detection of code-smells.

In our approach, we tackled the defect detection problem at the model level specifically in class diagrams. We circumvent the above-mentioned problems related to the use of rules, metrics, symptoms, and the manual adaptation/calibration effort by identifying directly the defect based on defects examples.

6 Conclusion

In this paper, we presented a novel search-based approach to improve the automation of design defect detection. We proposed an algorithm that is an adaptation of GA to exploit an existing corpus of known design defects and detect design defects in class diagrams. The proposed fitness function aims to maximize (1) the structural similarity between the model under analysis (i.e., class diagram) and the models in the base of examples and (2) the number of detected defects. We tested the approach on four open-source projects targeting the detection of three design defects. The results of our experiment have shown that the approach is stable regarding its correctness and completeness. The approach has also significantly increased the average precision and recall when compared to other approaches.

As part of future work, we plan first to cover all design defects potentially detectable in class diagrams. We plan also to extend our base of examples with additional badly designed models in order to take into consideration more programming contexts. We also want to study and analyze the impact of using domain-specific examples on the effectiveness of the approach. Actually, we kept the random aspect that characterizes GAs even in the choice of the projects used in the base of examples without prioritizing one or more specific projects on others to detect defects in the one under analysis. Finally, we want to apply the approach on other open-source projects and further analyze the type of defects that are correctly detected when using examples.

References

Alikacem, H., & Sahraoui, H. (2006). Détection d'anomalies utilisant un langage de description de règle de qualité. In *actes du 12e colloque LMO*.

Ben Fadhel, A., Kessentini M., Langer, P., & Wimmer, M. (2012). Search-based detection of high-level model changes. ICSM, pp 212–221.

Bois, B. D., Demeyer, S., & Verelst, J. (2004). Refactoring improving coupling and cohesion of existing code. In *Proceedings of the 11th working conference on reverse engineering*, pp. 144–151. IEEE Computer Society.

Brown, J. W., Raphael, C. M., Hays, W., & Thomas, J. M. (1998). *AntiPatterns: Refactoring software, architectures, and projects in crisis* (p. 336). New York, NY: Wiley.

Budi, A., Lucia, Lo, D., Jiang, L., & Wang, S. (2011). Automated detection of likely design flaws in N-tier architectures. *Software Engineering and Knowledge Engineering (SEKE)*, pp. 613–618.

El Boussaidi, G., & Mili, H. (2011). Understanding design patterns—What is the problem? *Software: Practice and Experience*. doi:10.1002/spe.1145.

Erni, K., & Lewerentz, C. (1996). Applying design-metrics to object-oriented frameworks. In *Proceedings of the 3rd international software metrics symposium*, pp. 64–74.

Fenton, N. E., & Pfleeger, A. S. L. (1998). *Software metrics: A rigorous and practical approach* (2nd ed., p. 656). Boston, MA: PWS Publishing Co.

Fowler, M., & Beck, K. (1999). Refactoring: Improving the design of existing code. In *Proceedings of the second XP universe and first agile universe conference on extreme programming and agile methods*, (p. 256). Springer.

Genero, M., Piattini, M., & Calero, C. (2002). Empirical validation of class diagram metrics. In *Proceedings of the international symposium in empirical software engineering*. (2002), pp. 195–203.

Ghannem, A., Kessentini, M., & El-Boussaidi, G. (2011). Detecting model refactoring opportunities using heuristic search. In M. Litoiu, E. Stroulia, & S. MacKay (Eds.) *Proceedings of the 2011 conference of the center for advanced studies on collaborative research* (*CASCON '11*) pp. 175–187. IBM Corp.: Riverton, NJ, USA.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning* (p. 372). Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Harman, M., & Tratt, L. (2007). Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on genetic and evolutionary computation,* (London, England), pp. 1106–1113. 1277176: ACM.

Jensen, A. C., & Cheng, B. H. C. (2010). On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the 12th annual conference on genetic and evolutionary computation*. (Portland, Oregon, USA), pp. 1341–1348. 1830731: ACM.

Kessentini, W., Kessentini, M., Sahraoui, H., Bechikh, S., & Ouni, A. (2014). A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Transactions on Software Engineering, 40*(9), 1.

Kessentini, M., Sahraoui, H., & Boukadoum, M. (2008). Model transformation as an optimization problem. In *Proceedings of the 11th international conference on model driven engineering languages and systems*. (Toulouse, France), pp. 159–173. Springer.

Kessentini, M., Sahraoui, H., Boukadoum, M., & Wimmer, M. (2011). Search-based design defects detection by example. In *Proceedings of the 14th international conference on fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software* (Saarbrücken, Germany), pp. 401–415. Springer.

Kessentini, M., Vaucher, S., & Sahraoui, H. (2010). Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM international conference on automated software engineering*. (Antwerp, Belgium), pp. 113–122. ACM.

Khomh, F., Vaucher, S., Gueheneuc, Y. G., & Sahraoui, H. (2009). A bayesian approach for the detection of code and design smells. In *Proceedings of the 9th international conference on quality software (QSIC)* pp. 305–314.

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection* (p. 680). Cambridge, MA, USA: MIT Press.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE international conference on software maintenance (ICSM)*. pp. 350–359.

Mens, T., Taentzer G., & Dirk. (2007). Challenges in model refactoring. In *Proceedings of the 1st workshop on refactoring tools* University of Berlin.

Moha, N., Gueheneuc, Y. G., Duchien, L., & Le Meur, A. F. (2008). DECOR: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions, 36*(1), 20–36.

O'Keeffe, M. (2008). Search-based refactoring: An empirical study. *Journal of Software: Maintenance and Evolution (JSME), 20*, 345–364.

Ouni, A., Kessentini, M., Sahraoui, H., & Boukadoum, M. (2013). Maintainability defects detection and correction: A multi-objective approach. *Automated Software Engineering (ASE), 20*, 47–79. doi:10.1007/s10515-011-0098-8.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2014). Detecting bad smells in source code using change history information. ASE 2013 pp. 268–278.

Seng, O., Stammel, J., & Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th annual conference on genetic and evolutionary computation*. (Seattle, Washington, USA), pp. 1909–1916. 1144315: ACM.

Van Der Straeten, R., Jonckers, V., & Mens, T. (2007). A formal approach to model refactoring and model refinement. *Software and Systems Modeling (SoSyM), 6*, 139–162. doi:10.1007/s10270-006-0025-9.

Van Kempen M., Chaudron M., Kourie, D., & Andrew, B. (2005). Towards proving preservation of behaviour of refactoring of UML models. In *Proceedings of the 2005 annual research conference of* *the south african institute of computer scientists and information technologists on IT research in developing countries (SAICSIT) (South African Institute for Computer Scientists and Information Technologists, Republic of South Africa)* (pp. 252–259). 1145703.

Zhang, J., Lin, Y., & Gray, J. (2005). Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven Software Development—Research and Practice in Software Engineering* (pp. 199–217). doi:10.1007/3-540-28554-7_9.

Adnane Ghannem is a PhD student at ETS-Canada. He is working on the application of artificial intelligence techniques in software engineering. His research interests include model-driven engineering, model refactoring, and software quality.

Ghizlane El Boussaidi is an associate professor in software engineering at the department of software and IT engineering of ETS-Canada. Her research interests include software architecture and design, software re-engineering and modernization, model-driven development, and model refactoring and transformation.

Marouane Kessentini is a tenure-track assistant professor at University of Michigan. He is the founder of the research group: Search-based Software Engineering@Michigan. He holds a PhD in Computer Science, University of Montreal (Canada), 2011. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering), software testing, model-driven engineering, software quality, and re-engineering. He has published around 50 papers in conferences, workshops, books, and journals including three best paper awards. He has served as program committee member in several major conferences (GECCO, ICMT, CLOSER, CAL, LMO, MODELS, etc.) and as organization member of many conferences and workshops (LMO, MDEBE, etc.). He is also the co-chair of SBSE track at the prestigious conference GECCO2014 (22nd International Conference on Genetic Algorithms and the 18th Annual Genetic Programming Conference).