UNIVERSITY OF GOTHENBURG

RADAR: An Approach for the Detection of Anti-patterns in UML Class Diagrams

Bachelor of Science Thesis in the Programme of Software Engineering

PETRA BÉCZI

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2015

PETRA BÉCZI

Academic Advisor:
MICHEL CHAUDRON

Examiner:
HÅKAN BURDEN

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2015

RADAR: An approach for the Detection of Anti-Patterns in UML Class Diagrams

Petra Béczi
Department of Computer Science and Engineering,
University of Gothenburg, Box 100, SE-405 30,
Gothenburg, Sweden.
gusbeczpe@student.gu.se

Michel Chaudron
Academic Advisor
chaudron@chalmers.se

***Abstract*—Anti-patterns in UML designs, alias bad design choices that are claimed as due to the incompetent participation of the designer to Object-Oriented (OO) system could lead to later issues regarding to its maintenance. That is why the early-capture of those is a common desirability and emphasized as it would be a highly important action for the prevention of issues. However, the discovery of anti-patterns is a difficult procedure in case of working with a large scale and complex OO system. There are existing metric based solutions in which the detection of anti-patterns is automatized, however, none of those operates on UML class diagrams. In this paper, we introduce our approach called RADAR, which is a solution to detect inter alia, Complex Class, Large Class, Lazy Class, and ManyFieldAttributesButNotComplex (MFABNC) in UML class diagrams and returns warnings of the results. Essentially, RADAR uses a combination of some existing software design (SD) metrics and rules of the anti-patterns, moreover provides a flexible algorithmic procedure. Since each class diagrams are structured uniquely, meaning that those consists of a different number of classes; where each classes are variedly sized; and sometimes the classes are purposely implements large number of attributes and operations, thereby causing hasty judgements in the detection procedure due to the characteristics of some anti-patterns, which can be detected specifically by the large number of attributes and operations of a class. Therefore, we are providing a supplementation for the detection algorithms to compare the size of each classes inside the diagrams with the calculations of quartiles and average. The purpose of this approach is not only the detection of anti-patterns, but the measurement of significance. The computations are used to analyze each classes in a class diagram, where the resulted values will be unique for every diagrams. Taking these uniquenesses into the detection rules, we enabled a more accurate decision-making for our algorithms in RADAR, such as the measurement of significance whether the suspected element is still considerable as an anti-pattern after the comparisons made against to the other classes. Hence, as the name of our approach indicates, RADAR "sees" the size of the suspects, but only "warns" in case of necessity. This research was carried out by the application of design research methodology with the induction of statistical analysis made on the test results compared to Ptidej tool suite v5.8.1. The test**

was performed on the same test materials and anti-pattern types, whereas the measurement- regarding to the test coverage from the aspect of what percentage can our RADAR approach locate from those classes per anti-pattern categories that Ptidej does- resulted to the average of 26% accuracy for RADAR in the detection of Complex Class, and 61% for the detection of Lazy Class symptoms. Unfortunately, this measurement could not be made for Large Class and MFABNC anti-patterns based on the reason that Ptidej tool could not detect those. We claimed this problem as due to code generation procedure by the software we used regarding to the arbitrarily generated lines of code (LOC), which is if is long, is actually one of the main symptom of the latter mentioned two anti-patterns. To compensate the deficiencies of the measurement regarding to the detection accuracy of the four anti-patterns, we requested a review for RADAR from various people in the person of PhD students. Based on their feedback, the accuracy of RADAR in the detection of Large Class is between the averages of 38%-75%, and 55%-80% for MFABNC. Regarding to the average accuracy of the other two anti-patterns, the detection of Complex Class is determined as between 68%-70%, and 80%-99% for Lazy Class.

***Keywords*—Anti-pattern; UML class diagram; Software design metric; Detection rule; Detection algorithm; Testing; Statistical analysis.**

1. Introduction

In the past years, Unified Model Language (UML) has been widely accepted in the field of software engineering, as much as it has become ubiquitous in many design contexts especially [1]. As the definition indicates, UML is a standard language to specify, construct, document and visualize artifacts of a software system. The core of UML is to visualize an architectural blueprint by drawing out individual components of the system including relationships among them. Since the blueprint represent a design of how the system is supposed to be at the end, therefore it should capture its characteristics in such way that is visually perceptive and interpretable to the viewers. Advantageously in reverse engineering, the construction of the architecture is usually precedes the code-writing and code debugging processes, thus this procedure enables the designer to preview the system s/he is about to build and make modifications before writing the actual code in cases of mistakes are found, for example within the composition of the models, or inaccurate relationships between components [2]. The early prevention of problems seems like an optimal act in theory, however the discovery of those is limited in practice, with a great emphasis on the designer's competence of Object-Oriented (OO) design. There are rules, patterns, and concepts that are necessary to acquire for designing good architectural models [3]. As we are interested in UML class diagrams, these patterns are then regarding to the declaration of classes with the rules of defining inter alia, the stereotypes for classes, class members (attributes and functions), abstractions, associations, types of the relationships, navigability and aggregations, stereotypes for dependency, and constraints by using the proper syntax and design. To efficiently utilize these design patterns, the developer is required to have an adequate understanding of them in order to properly create suitable instances for a software or a system. For example, a well-designed UML class diagram is also dependent of whether its designer has properly defined the interlinking of- just as the strength of the classes in the diagram [2][4]. These two inseparably cited concepts are called coupling and cohesion, where coupling is the measurement of relative interdepency between classes as one is associated with another class, while cohesion is regarding to the strength of attributes as how those are linked inside a class. If the coupling is high, and the cohesion is low, then it is considered as a non-optimal, highly complex OO

design. In parallel to design patterns, the existence of anti-patterns are notorious for being their undesirable contradictions meaning that, anti-patterns are the "collections" of consequential poor solutions to problems of frequent-occurrence [5][6]. For example, the lack of aforementioned competence of the designer regarding to the UML and OO design concepts could mistakenly provoke anti-patterns in the system. The outcome of the bad design choices could take formations as one or more types of known anti-patterns, which may occur at variety levels of the system [7][10], and knowing about their exact location is difficult due to the fact that the commitment of anti-patterns was unintentional. Moreover, working with an OO system that is large and complex raised the difficulty of the discovery of anti-patterns due to its size, meaning that it is an exceedingly time consuming procedure to look through manually [8-10], but should not be neglected to a later-maintenance, since anti-patterns could lead to issues that are related to software testing and database [11][12]. Although the current CASE tools just as StarUML, Visual Paradigm and Enterprise Architect that support many features, they do not have any anti-pattern detection feature. On the other hand, there are various anti-pattern detection solutions exists to moderate the work of the designer, which are mostly metric based and semi-automatic approaches [1,5,8,10,11][15-19] that unburdens the discovery procedure and realization of anti-patterns by overtaking most of the work via the automatic localization of those. Unfortunately, these solutions does not considering the detection of anti-patterns in UML class diagrams.

2. Background and Related Work

In this section we present the theory of anti-patterns. Moreover, we cite here the collection of those detection techniques, which served as the basis of this research project.

2.1. Theory of Anti-Patterns

According to Budgen [6], software design patterns suggest "good" solutions to the recurring design problems. To efficiently utilize the design pattern, a developer is required to have an adequate understanding of the design patterns in order to properly create suitable instances for the software. The lack of such a high-advanced proficiency usually result in anti-patterns in the software (code level or design level). The anti-pattern is indicated as a literary form that describes a commonly occurring solution to a problem, but generates decidedly negative results.

McCormick et al. [7] also describes the anti-pattern phenomenome as *"code smell"* at the source code level in software design, which he considers as a bad programming practice that impacts the readability and reusability of the source code. Likewise, the anti-pattern at the design level, which he indicates as the poor choices of the software architecture model, alias *"design smell"* that fails to represent the essence of the software structure. To some extent, the anti-pattern is a miscalculated or immature blueprint for the software.

Guéhéneuc [10] distinguishes between the variety of patterns and defects, such as: *idioms*, as programming languages or the implementation of class characteristics are low-level patterns, where the so-called intra-class patterns are describing the relationships and object containment of them. *Micro-patterns* are well-defined idioms related to the design of classes in object-oriented programming. *Design patterns* are iterative inter-class patterns, which are defined in terms of classes and relationships by using idioms, the task of the design patterns is to provide solutions to common design problems regarding to the disposition of classes. Eventually, he consider *design defects* as the "opposite" of *design patterns,* since they describe "bad solutions" to recurring design problems.

Back to the work of McCormick et al. [7], the book draws the attention to the menace of these multiform anti-patterns that are likely to be scattered at different levels of the software system. Anti-patterns usually occurs when developers of inexperienced in OO are attempting to implement applications using OO language in an inadequate way. For example, when developers create classes for each subroutines and by that ignoring the hierarchy of the class. Thereby, the code becomes similar to the solution of a structural language, however, it differs since its structuredness is provided from the structure of the class. A typical sign of this ignorance if the classes contains only one method per class for example. An anti-pattern can also be a class which contains an overwhelming number of attributes, functions, and/or even associated with many other data-object classes as well. Consequently, this type of anti-pattern causes the complexity of code or design, which then affects the degree of fulfillment of the functional requirements. The most common occurrence of this anti-pattern is when a class expropriates the process, while the task of the remaining classes is mainly the unification of the data. The problem with this "task-division" is that a single controller class (also called as a God Class) takes the most responsibilities, thereby growing its requirement regarding to the memory usage. Moreover, testing this class would be complicated and expensive due to the number of resources that might needed to accomplish a simple operation. Contrary to a large class, there can be classes that were implemented previously, but became unused, thus new functionality was never added. The consequences of having classes without furthermore purposes causes unnecessary costs in the system maintenance therefore should be eliminated according to Munro [14]. On the other hand, anti-patterns can also mean a program or a system which contains very little software structure. The main attribution of that if it has an exiguous amount of objects that contains methods with enormous implementations. These methods can trigger multistage processing. Consequently, such anti-pattern causes inflexibility in making changes to the code, which then limits the further development and maintenance of the system. Please note that these are only a few examples that we used for describing anti-patterns since these are the most commonly occurring ones usually, therefore, we have included a complete list of them in Appendix A.

2. 2. Existing Solutions to Detect Anti-Patterns

In 1997, Grotehen et al. [18] proposed their approach called METHOOD, which was the first one with the idea behind to measure size, hiding, coupling and cohesion to find anti-patterns in UML designs. However their approach have been implemented in a tool named MEX, yet it have not been experimentally tested or evaluated, and the procedure of anti-pattern detection mostly remained as the golden age of investigation approaches at the source code level till the next decade.

Manual detection and refactoring solutions, for example *Refactoring of Responsibilities, Object−Oriented Reengineering,* and *Ghostbusting* were proposed by McCormick et al. [7], as recommendations to deal with anti-patterns at code level.

Khomh et al. [10] and Gueheneuc [11] claimed that the manual detection of anti-patterns is not only sounds as an awfully exhaustive procedure, but indeed complicated in the case of looking through a large scale OO system, which is time consuming to the extent that it takes 75 percent of the maintainers work.

In 2004, Marinescu and Trifu et al. suggested their semi-automatic approaches to detect design flaws with the tools iPlasma [15] and jGoose [16] design database creator. Essentially in both techniques, the definitions of anti-patterns are implemented as rules to analyze the code, then basic metrics are used to filter the code for symptoms just as, high coupling, the complexity of methods, lines of code, and the entire control flow. However in Trifu's approach, the design database that was created by the tool jGoose is then stored as an XMI model (using XQuery to access into design flaws).

In 2006, AliKacem et al. [17] recommends an approach, which uses a meta-model to represent the source code and detect violations against quality rules in OO programs. They have classified the quality rules into three subcategories: (1) metric-based rules, (2) structural information-based rules and (3) rules expressing abstract notions. These rules are then expressed in a language called Backus Normal Form (BNF) that is independent of the programming language.

Moha et al. [8][1] introduced the first approach, which uses the specification of design defects expressed by rule cards. Each rule card is a representation of a "*code smell"*, just as a defect that can be tracked down by measurable, structural, and lexical properties. Similarly to AliKacem et al. [17], Moha's rule cards are expressed by the BNF grammar that can determine the exact syntax for a language, where an auto-generated algorithm is responsible for detecting and correcting the design defects. This approach is beneficial from the aspect of reducing the time spent for discovering anti-patterns, however, the specification of a rule cards is still depending on the competence of the designer in the correct declaration of those to detect specific design defects. In the year of 2007, Montréal and Guéhéneuc [10] have offered pattern identification algorithms with their tool, Ptidej, which implements the improvements they made based on their previous work [8][1]. Essentially, this solution includes PADL (Pattern and Abstract-level Description Language) meta-model that represents OO systems and patterns with a unified language. PADL is then used to analyze the system on three different levels: (1) analysis directly on models of the systems and patterns, (2) UI-related analyses on models of the systems and patterns to change their graphical representation, and (3) analysis of UI extensions, allowing a richer interaction between the analyses and the user. The purpose of this tool is to warn system maintainers for bad design choices. The detection procedure can be started by the user through a checklist that is provided by the interface. The elements of the checklist are anti-patterns, and can be selected one by one or all-together. The back-end then runs the appropriate algorithms that are based on rules that are dedicated to recognize and capture anti-patterns. At the end of the detection procedure, the tool uses a red signal that indicates the location of the anti-pattern(s) detected, and generates textural reports as well.

A year later, Ballis et al. [19] offers a solution to detect anti-patterns at design level instead of code level by inspecting those in diagrams via rules, which are defined texturally or in a graphical language that extends UML with a few graphical primitives. The detection procedure involves graphical notation as a warning service similarly to Ptidej tool [11]. Unfortunately, the success rate of this approach is dependent of how well is the diagram defined and structured that is being tested.

Fourati et al. [5] introduced a metric-based approach that can successfully detect five anti-patterns in sequence diagrams by the measurements of (1) coupling, (2) cohesion, (3) complexity and (4) inheritance. This solution including the examination of both structural and behavioral information of the testable diagram, which are necessary steps from the aspect of detecting anti-pattern symptoms at design level. To begin the examination procedure, they have applied those of the OO software metrics, which can be used to measure quantifiable properties of sequence diagrams, and are grabbing the relevant types of information regarding to the characteristics of those anti-patterns, which they have selected. See Table 1.

Coupling	***CBO***	Coupling Between Objects
	RFC	Response For Call
Cohesion	***LCOM***	Lack Of Cohesion in Methods
	TCC	Tight Class Cohesion
	LCC	Loose Class Cohesion
	Coh	For class with N methods
	WMC	Weighted Method per Class
	NAtt	The Number of Attributes of a class

Complexity	***NPrAtt***	The Number of Private Attributes.
	NOM	The Number of Methods of a class including the constructor.
	NII	The Number of Imported Interfaces.
Inheritance	***DIT***	Depth of Inheritance of a class.
	NOC	Number Of Children.
	NAcc	The Number of Accessors in a class.
	NAss	The Number of Associations.
	NInvoc	The Number of Invoked methods (Call Action in the sequence diagram) of a class.
	NReceive	The Number of Received messages that invoke methods of this class.

Table 1. Useful OO software metrics (Resource reference: Fourati et al. [5]).

For example, to detect Blob symptoms (large controller class, surrounded by many data classes) of a sequence diagram, the implementation of the following metrics are suggested by Fourati et al. [5]. ***NAtt*** high and ***NOM*** high and ***Coh*** low and ***Coh1*** is true and ***Coh2*** is true and ***DIT*** low and ***NOC*** low and ***RFC*** high and ***CBO*** high and ***IsController*** is true. ***NAcc*** high and ***NOM*** low and ***DIT*** low and ***NOC*** low and ***IsAccessor*** is true. Similarly to this implementation of the metrics, their approach can successfully detect Lava Flow, Functional Decomposition, Poltergeists and Swiss Army Knife anti-patterns besides Blob. However, different metrics are used or combined together for the detection of others, since all of the anti-patterns have their own characteristics (symptoms) that could be either unique or partially true for others.

These metric based and semi-automated techniques we mentioned above mostly performs at code level, and out of the few that operates at design level does not consider the detection of anti-patterns in UML class diagrams yet.

3. Project Aim

3. 1. Objectives

We have set the detection of anti-patterns in UML class diagrams as our project aim, since there were no approach that could operate on those before, however, is claimed as it could be especially needful for complex OO system designs from the aspect of maintenance, regarding to difficulties in the management of changes and time [7,10,11]. As our motivation was to help software designers to locate design defects in his/her UML class diagram, our primary sub goal was to (1) write such algorithms that can detect the anti-patterns, inter alias, the *Complex Class*, *Large Class, Lazy Class*, and *ManyFieldAttributesButNotComplex*. The purpose of choice regarding to the quantitative selection of anti-pattern types to be detected was not considered as one of the goals for our solution. We selected these few out of the multiform types rather to serve as examples for the introduction of algorithmic procedures required to scan for symptoms in class diagram designs, with the intention to provide initial guidance to the readers. On the other hand, we did not consider to detect the more widely known anti-patterns just as Blob or Spaghetti Code due to those were exhaustively used as examples by various authors of other approaches, however the symptoms of Large and Complex Class shares similar attributes with the latter mentioned ones. To define the detection rules, first we gathered the appropriate software design (SD) metrics e.g.: *NumAttr*, and *NumOps*, that could be used to measure properties of UML class diagrams, which are as well relevant to examine the symptoms of these four anti-patterns. The secondary sub goal was to (2) improve the detection procedure with the consideration of size in order to measure the significance of the found anti-pattern(s). Since each class diagrams are structured uniquely, meaning that those consists of a different number of classes; where each classes are variedly sized; and sometimes the classes are purposely implements large number of attributes and operations, thereby causing hasty judgements in the detection procedure due to the

characteristics of some anti-patterns, which can be detected specifically by the large number of attributes and operations of a class. Finally, the third sub goal was then to (3) provide warnings of the results with the textural output of the detected anti-pattern name.
The expected outcome of this research was to work out an algorithmic solution, which enables the detection of the selected anti-patterns in UML class diagrams.

3. 2. Research Questions

Main RQ *How to detect anti-patterns in UML class diagrams?* There are several approaches already proposed to deal with anti-patterns in both code and design level, however, none of the propositions considers the detection of anti-patterns in UML class diagrams yet, and the fact that the solution was unknown made us the pioneers from this aspect. To provide an appropriate answer for the main research question, first we have conducted a deeper investigation, which was an imperative step in order to put together the pieces of the puzzle. For that, we have applied design research methodology [21][22] that we could use to accommodate different techniques to find answers for the following questions.

RQ1 *Which of the existing software designs metrics are useful to examine properties of UML class diagrams to measure anti-pattern symptoms of the selected ones?* We study the definition and characteristics of the selected anti-patterns in order to know which SD metrics are relevant. By following that, we write conceptual rules for each anti-pattern by using the appropriate SD metrics. After that, we implement the conceptually designed rules into basic functions, and run tests.

RQ2 *What procedure could help to measure the significance of the found anti-pattern(s) uniquely for all the classes of a diagram, thereby raising the accuracy of the detection results?* For this question, we investigate which mathematical solution could we use to compare each classes of a UML class diagram, and by that to decide upon whether is indeed an anti-pattern suspicious element.

RQ3 *Does the detection with our RADAR approach showing significantly different results than a test made with Ptidej tool in a comparison?* This question involves the conduction of a test against a trusted anti-pattern detection tool, Ptidej tool suite v5.8.1, which action can be considered as a small experiment to compare the accuracy of the RADAR approach. For that, we have selected 63 test materials of anti-pattern suspicious UML class diagrams regarding to either Complex Class, Large Class, Lazy Class or ManyFieldAttributesButNotComplex symptoms. These test materials were then segregated into two data sets, models41 and models22 due to different file formats, which selection we explain in the next section. Our general hypothesis was that the RADAR approach could also detect those anti-patterns that Ptidej tool does. The alternative hypotheses for the test [20] are stated below:

- H0 Set1 _ There is no difference in the result (i.e. found number of anti-patterns/category) between the tests performed with RADAR and Ptidej tool on data set models41.
- H1 Set1 _ There is a difference in the result (i.e. found number of anti-patterns/category) between the tests performed with RADAR and Ptidej tool on data set models41.
- H0 Set2 _ There is no difference in the result (i.e. found number of anti-patterns/category) between the tests performed with RADAR and Ptidej tool on data set models22.

- H1 Set2 _ There is a difference in the result (i.e. found number of anti-patterns/category) between the tests performed with RADAR and Ptidej tool on data set models22.

RQ4 *What is the average accuracy of RADAR in the detection of the four anti-pattern types?* The last question involves the evaluation of RADAR by experts.
In the next section, we describe our research strategy in more detail that we applied to carry out this project. After that, we present the solution that we provide with our RADAR approach and how it performs in a test compared to Ptidej tool, and present the accuracy of RADAR in section 5. Ultimately we express our conclusion in section 6.

4. Research Methodology

In this section we describe the research strategy we have chosen to conduct this research, just as how did we plan each of the steps to study the topic, develop and evaluate the RADAR solution.

4. 1. Research Strategy

We conducted this project under academic setting and with the application of design research methodology [21][22]. The decision of using design research was mainly due to the reason that the accomplishment of RADAR required flexibility, which concerned three major phases, (1) data gathering stage, (2) design and implementation stage, finally the (3) validation stage. Design research was then engaging from the perspective that it has no formal rules, thereby allowing the flexibility for us to apply different techniques. Thus we accommodated both quantitative and qualitative strategies this study. Due to the academic setting and the aim of the project, the design research approach is selected as the guideline for our project development. Although some quantitative research approaches were considered in the initial stage as we planned to send out surveys and questionnaires in the campus to gather students' opinion towards our solution to evaluate the accuracy of RADAR. Then we realized the students may not be equipped with enough profound knowledge to give us professional results. Hence we shifted our focus on other quantitative methods such as statistical analysis made on the test results between RADAR compared to Ptidej tool, and a review of the feedback received from PhD students upon the accuracy of RADAR. The design research methodology is a perfect fit for the one kind of development, which is meant to provide clear and verifiable contributions [24]. And this trait exactly agrees with the idea of our research: to provide a verifiable anti-pattern detection algorithm for software developers.

In the first stage, we have conducted a deeper investigation to study and collect materials that we found useful to design and test our RADAR approach, which can be understood as an imperative step in order to put together the pieces of the puzzle.

In the design and implementation stage, we used the knowledge we gained from the first phase when we implemented the basic detection algorithms, which we later improved in order to not only to analyze the test material for anti-pattern symptoms, but as well compare each classes of the UML class diagram, and by that to decide upon whether the detected element is indeed an anti-pattern suspicious element.

In the validation phase, we have conducted a test with a trusted anti-pattern detection tool, Ptidej tool suite v5.8.1, which action can be considered as a small experiment to compare the accuracy of our approach. To measure that, we feed the tool with the same materials we used for RADAR. By following that, we compared the results under the application of statistical analysis. The entire research have been conducted under the assistance of an

academic supervisor, who were continuously asked to accompany us with his approval. On the other hand, we have requested experts of the field, such as PhD students to give us their fruitful feedback in order to assist us in the determination of the accuracy level of RADAR solution.

4. 2. Data Collection Plan

In the beginning of this research project, we have started reviewing papers to study the multiform anti-patterns including the root causes and possible consequences of its appearance, but most importantly, to explore the existing metric based approaches regarding to the detection of anti-pattern symptoms in general. We have gathered qualitative data mainly with the facilitation of digital libraries such as IEEE Xplore, where we entered keywords to see the selective output of such reliable papers that are in pursuance of our research topic and to our research questions. However, we did not set hard inclusion-exclusion criteria and quality assessment of the paper selections as we would strongly consider in the application of systematic literature review (SLR) method [23]. The reason why we "committed this violation" against the rules of SLR was due to that our contribution was an intention to provide an innovation, a reform of the former anti-pattern detection approaches, which as well the first approach to detect anti-patterns in UML class diagrams as those have not been considered before. Based on this reason and to gain an adequate amount of information that we can use, we were venturous to review websites and forums as well. The data we gathered in this phase was then stored in an excel spreadsheets for organization purposes, and placed under the following categories: (a) list of anti-patterns, (b) useful existing detection techniques, and (c) relevant SD metrics to design level detection, along with the resource references. After we have gained enough competence to the recognition of different anti-patterns by looking at pictures, we were ready to browse for test materials of anti-pattern suspicious UML class diagrams. Only those diagrams were selected, which were at least suspicious for one of the four types of anti-patterns (*Complex Class, Large Class, Lazy Class, and ManyFieldAttributeButNotComplex*) that we have selected as the goal to detect. The result was the collection of 41 UML class diagrams in both JPG and XMI formats, which we downloaded from the online repository www.models-db.com. Moreover, we received files from our academic supervisor including C++ source code of another 22 class diagrams. Thereby, we gathered 63 UML class diagrams in total, which we saved into a file collection. As next, we recorded down the collected diagrams and manually stored them into Mircosoft Excel 2013 spreadsheet where we created two main tables, models_41 and models_22. Initially, these two tables were then used to isolate all diagrams of the two sets, where we organize them under *Model No.* as integers, and *Class Name* as strings. Since each model contained different number of classes, therefore the same model number was entered to those. Eventually, we have extended the categories of the tables with the previously collected SD metrics, where the data entries were integers. See Table 2.

Model No.	*Class Name*	*NumAttr*	*NumOps*	*Metric N*
1	Configurator	3	2	..
1	DBManager	3	1	..
1	EventTimer	4	8	..
2	Publication	21	2	..
ModelIntN	NameStringN	MetricIntN	MetricIntN	MetricIntN

Table 2. Categorized UML class diagrams with SD metrics.

For testing our RADAR approach, we have implemented the categories for the anti-patterns we intended to detect. The detection rules we created were entered as functions, which selects entries of Model No. column along with entries of relevant SD metrics columns to measure

symptoms of those anti-patterns. If the functions returned true, the entry for that anti-pattern category - on the appropriate row for the class - automatically changed to "Anti-pattern name", meaning that the measurement of the symptoms indicated the appearance of that anti-pattern for that class, otherwise changes to "Not+anti-pattern name". Designing and testing of the functions was a repetitive step till we reached the desired results, including more than one function to catch the same types of anti-patterns. The excel sheet then contained the test results made with RADAR approach of the 63 UML class diagrams, with a clear indication of which classes are the problematic ones.

On the other hand, we performed tests for detecting the same anti-patterns with Ptidej tool suite (v5.8.1) on the same class diagrams. However, this action required the preliminary steps of code generation regarding to the inappropriate file format of the 41 diagrams. The reason for that was due to Ptidej tool cannot import XMI files, and can only detect anti-patterns in the source code, which caused us the necessity to generate source code from the 41 class diagrams since we only had them in XMI format. To overcome this barrier, we used StartUML tool to generate both Java and C++ code out of the XMI files, which we saved into the file collection that we created before. In the following step, we tested the 41 diagrams by feeding Ptidej tool with the (1) generated Java and (2) generated C++ code separately. Testing the other 22 diagrams did not require code generation procedure since we had the appropriate file format previously, therefore we could import and test the (3) C++ code effortlessly. At the end of the test, we took the detection result that Ptidej tool generated as text files, which were automatically categorized by anti-pattern types, and saved them into the file collection we created before. Eventually, we extended our excel spreadsheet (models_41 and models_22) tables with the test generated results of Ptidej by taking those and manually store each of them under the extension columns for anti-pattern categories, where the entries of positive and negative result were associated with the appropriate id and class name of the diagrams. Ultimately, we emailed our excel spreadsheet to our academic supervisor for approval, which contained inter alia, all the 63 diagrams, metrics and algorithms used by RADAR, and the test results including Ptidej ones. Moreover, we as well attached our file collection to that email in order to show the evidences.

For the evaluation of our solution, we have contacted with PhD students of those who are equipped with profound knowledge of anti-patterns, but were not involved in any other phases of our research project. The first contact with them was made verbally then via email to formalize the agreement between us. The email then contained the attachment of our (1) Excel file of RADAR solution, and the (2) file collection containing all the test materials. Moreover, they have been asked to examine all the class diagrams and by following that, to extend our Excel file with their judgments by giving marks "agree" and " disagree" regarding to positive, negative, false positive and false negative results of ours. Their ultimate task was then to send their extended Excel file back to us via email.

Item	Purpose	Reference / Resource
Anti-pattern definitions and examples	To collect textural description and visual examples of *Complex Class, Large Class, Lazy Class, and ManyFieldAttributesButNotComplex anti-patterns.*	[1], [5], [7],[9], also accessible at url: http://wiki.ptidej.net/doku.php?id=sad
Rules for anti-pattern detection	To collect rules for we could use to express our rules for the four anti-patterns.	[8], [13],also accessible at url: http://www.ptidej.net/search?SearchableText=rule+card
SD metrics	To collect software design metrics for UML class diagrams.	[5], also accessible at url: http://www.sdmetrics.com/LoM.html
UML repository of class diagrams	To collect anti-pattern suspicious test materials in XMI and JPG file formats.	Accessible at url: http://modelsdb.com/

Anti-pattern detection test results with Ptidej tool	To perform tests on the selected class diagrams, and to collect generated test reports out of them.	See in the results section of this paper. The tool can be downloaded at url: http://www.ptidej.net/tools/designpatterns/.
Anti-pattern detection test results with RADAR approach.	To perform tests on the same class diagrams, and to collect the textural output e.g.: the results.	See in the results section of this paper.

Table 3. Summary of the collected data.

4.3. Data Analysis Plan

In the previous paragraph, we described how we collected the data that preceded the conduction of tests with RADAR and Ptidej, moreover how we stored the test results. However, the conduction of tests and the approval from our mentor did not give us enough confidence to provide answer to the main research question, neither to conclude a high accuracy of our anti-pattern detection algorithms. Therefore, we performed statistical analysis [25] on the test results in a manner of percentage of the test coverage by RADAR compared to Ptidej.

Before we begun the statistical data analysis procedure, first we cloned our spreadsheet tables and modified the entries of the anti-pattern categories from String data type to Integers by giving number 1 for positive, and number 0 for negative test results. By following that, we computed the number of found anti-patterns individually for all categories per RADAR and per Ptidej, and as well for both data sets (Models41, Models22). We then created a table of summary. However, as we originally provided more than just one strategy per category for RADAR to detect the same anti-pattern types differently, those mostly returned same results nonetheless. Correspondingly, this was true to the results of Ptidej regarding to the model set 41, which we tested twice by feeding the tool with the source code of two different programming languages, which were in fact generated from the same XMI files. Therefore, we have considered the fact that the synthesization of the data per anti-pattern category would contain duplicates in our newly created table of summary. However, we were not interested to compare between the particular detection strategies of anti-pattern categories in RADAR, neither in Ptidej, we considered this as a necessary preceding step to partially answer research question 3. Meaning that the outcome of this step, i.e. the table of summary, served as the evidence for the later presentation of the statistical measurement of the test results between the two approaches. More explicitly, to the second step, which was a measurement regarding to the test coverage from the aspect of what percentage could our RADAR detector approach locate from those anti-patterns that Ptidej did. In order to find that out, first it was necessary to eliminate the duplicates due to the reason that those would influence the measurement of the coverage to the extent, that either positively or negatively, but would cause distortion to the facts. Therefore as the next procedure, we planned to eliminate those by uniting of various detection strategies per same anti-pattern category into one. For that, we filtered the different anti-pattern detection strategies for entries where equals to 1 (true), and if the same *Model No*. and *Class Name* was associated to more than one entry from these strategies per same category, then we counted it once. On the other hand, if at least one of the detection strategies of that anti-pattern type resulted 1, then it is also considered as 1. See Table 4 as an example.

From this table,

Model No.	***Class Name***	***Strategy1 AntipA***	***Strategy2 AntipA***	***Strategy3 AntipA***	***Strategy1 AntipB***	***Strategy2 AntipB***
1	Configurator	1	1	0	0	0
2	DBManager	1	0	1	0	1

we created the following table

Model No.	*Class Name*	*AntipatternCategoryA*	*AntipatternCategoryB*
1	Configurator	1	0
2	DBManager	1	1

Table 4. The conjunction of different detection techniques per anti-pattern categories.

We have merged the different strategies per anti-pattern types in both model set 41 and 22. However in modelset22, we also merged Java and C++ results the same way to eliminate duplicates. This conjunction was then the recapitulation of the data, which we prepared for to begin the statistical analysis. By following that, we planned to analyze the above mentioned table (similarly to the previous filtering method) by taking all the *Model No.* and *Class Name* from Ptidej per anti-pattern categories, distinguished between data set 41 and 22, where the entry was number 1(true). This step was relevant in order to see which classes were judged as the ones containing anti-pattern(s) by Ptidej tool. Taken the same information, we filtered the results from RADAR as well. Finally, we created the last table to present the identification of those classes of the UML class diagrams that RADAR could find from the ones that Ptidej did, thereby providing answer to research question 3. On the other hand, we computed the sum of the number of found anti-patterns per categories (types) for Models41, Models22, and for both RADAR and Ptidej. This data represented the "observed" data from which we used to calculate the differences (the deviation) regarding to the detection of the number of anti-patterns per UML class diagrams. To measure the data distribution regarding to the hypotheses, we calculated the p-values for both data sets via the formula of the Chi Square test [26]. See Figure 1.

$$x^2 = \sum \frac{(O - E)^2}{E}$$

Fig. 1. Chi Square formula.

Where O = Observed frequency and E = Expected frequency.

However, we believed that another test should be performed in order to confidently disclosure whether the p-value is significant. Therefore in this last step, we have performed the Mann Whitney U test [27] to see the difference between the results (i.e. found number of anti-patterns/category). Our main reason of choice to conduct this test is that this test could be used to compare differences between two independent groups when the dependent variable is either ordinal or continuous. For the conduction of this test, we have the formula on Figure 2.

$$U_1 + U_2 = n_1 n_2 + \frac{n_1(n_1 + 1)}{2} = R_1 + n_1 n_2 + \frac{n_2(n_2 + 1)}{2} - R_2$$

Fig. 2. Mann-Whitney U formula.

Where n1= Sample size for sample 1, R1= Sum of ranks in sample 1, and n2= Sample size for sample 2, R2= Sum of ranks in sample 2.

After we have interpreted the results of the Chi Square test to get the p-values, and we performed the Mann-Whitney U test to get the p-values and u-values, then these u-values were used to verify the significance level. The general rule of the p-value is that if the result is less than the critical 0.05, then there is no significant difference in the distribution of the data sets. Hence, the value we got from the calculations compared to this critical 0.05 played the dominant role when we were deciding whether we should reject or not the hypotheses. Eventually, we planned to present statistical results in forms of tables and charts.

The ultimate step of the data analysis was then the interpretation of the feedback we received from the experts. In order to answer research question 4, first we opened the

extended Excel file to read the occurrence of each “agree” and “disagree”. Then similarly to the previous procedures, we created a table containing the numeric representation of their feedback that we categorized under “agreed”, “disagreed”, and “commented (neither agreed nor disagreed)” in a matrix of detection strategies per anti-pattern categories. By having the summary of their feedback in numeric format, we then compared the numbers of “strongly agreed” to the sum of the anti-patterns that our test resulted previously, then we calculated the percentage of the coverage for each detection strategies. By having the percentage of coverage for each, we then computed the average number in order to declare the accuracy of RADAR in the four anti-pattern categories where the different strategies per detection categories were merged.
For answering research question 1, we present the SD metrics we used to measure properties of UML class diagrams, while for research question 2, we describe our solution for the detection procedure we designed for the four anti-pattern categories by providing those in pseudo code written algorithms.

4.4. Validity Threats

Several validity threats were identified during our development. The first issue is the selection of our candidate class diagrams, which could be with anti-pattern behavior. Under the consideration of time, we prone to choose the problematic class diagrams instead of randomly selecting class diagrams. One thought behind that is we have tested with a number of diagrams in the beginning, and frustratingly we discovered that most of them are without any anti-patterns. Then we realized that spending too much time on randomly selecting class diagrams would purposelessly waste our energy and may not provide direct and effective contribution to our research goal. Therefore, we decided to lay our emphasis mainly on the problematic diagrams when we were searching for those to test our algorithms. The second issue is the hardship in keeping the alignment between our test results and Ptidej tool. As we mentioned before we know Ptidej tool performs the detection at code level, and although the tool uses the same definition as ours to detect the anti-patterns, however the metrics between these two tools vary at some level. The reason could be the different characteristics within the source code and UML model. On the other hand, the quantum of measurement is what we identified as the most critical aspect of the validity regarding to the accuracy of our algorithms that we heralded next to the fabrication of a solution. Hence, the “solution” to be loyal to its meaning requires multitude measurements before we could declare it as indeed one, withal the involvement of people evidently to decrease the impression of a bias. Subsequently, another threat to the validity can be address here that is the determination of which group of people is equipped with the profound knowledge. We have considered the persons of professors and/or PhD students of software engineering, who has some degrees of competence in the field of reverse engineering, and familiar with the phenomenon of anti-patterns as appropriate candidates for participating in the evaluation of our solution. However, as we raised our expectations regarding to the skills of people, as decreased the number of selected participants in parallel. Knowing that the lack of measurement can strongly question the validity, therefore, we decided to carefully phrase our sentences to prevent any misleading conclusion that is associated with the accuracy of our solution in such case.

5. The RADAR Solution

In this section we present the results of the rules we applied to detect the different types of anti-patterns on the selected UML class diagrams as test materials for our project. Moreover, here we compare our test results with the results we received from Ptidej tool and ultimately, we present the evaluation of our approach.

As the outcome of the data collection regarding to which metrics could be useful to detect for anti-pattern symptoms in UML class diagrams, we have gathered the following SD metrics:

- **NumAttr:** Number of attributes in a class.
- **NumOps:** Number of operations in a class.
- **NAss:** Number of associations (coupling) with that class.
- **NOC:** Number of children of that class.

However, we realized that using only these SD metrics could not consider the uniqueness of each class diagrams from the aspect of size differences when declaring the detection rules for anti-pattern symptoms. For example: given the rule of the gauge "high" is >=8, while analyzing a class in a diagram. The class have the characteristics of NumOps = 11 and NAss = 16. These metrics (excluding these specific integers) are used to measure symptoms of the Large Class anti-pattern, where the former detection rule would be: if NumOps high is true and NAss high is true, then the class is a Large Class in the inspected diagram. But this raised the question in us regarding to what happens in such case, when other classes also having similar characteristics in the inspected diagram. The former detection rule by using these basic SD metrics would declare them all as infected with Large Class anti-pattern, meanwhile as well not considering the possibility such as the diagram was designed to be large on purpose.

5.1. RADAR Detection Algorithms for Anti-patterns in UML class diagrams

To precisely catch the anti-pattern in UML class diagrams, numerous testing and comparing works are involved in this process. In the beginning phase, the hardest task is to find a suitable threshold to evaluate our approach. For example, High No. Attributes and High No. Operations together determine whether or not the target class of the diagram is a Complex Class. But to what extent it can give us evidence about that this class has a higher number of attributes and operations than other classes is not a one-day-to-answer question. The very first idea that occurred to us is to use "one-third" as the detection threshold. Take how we check the high number of attributes as an example, we first get the sum of all attributes in one entire diagram and compute the average value of it. Then we compare the each number of attributes in one single diagram with the previously gained one-third value. If it is higher than the one-third value then we consider it as the high number of attributes. To implement these calculations, we have created the following SD metrics to RADAR:

- **NumAttr Quartile 25%** = Calculate the 1st quartile from the number of attributes taken from the classes of the diagram.
- **NumAttr Quartile 75%** = Calculate the 3rd quartile from the number of attributes taken from the classes of the diagram.
- **Low No. Attributes (Quartile)** = Compare if the number of attributes in the selected class is less or equals to the value of NumAtrr Quartile 25%, then return true else false.
- **High No. of Attributes (Quartile)** = Compare if the number of attributes in the selected class is larger than the value of the NumAtrr Quartile 75%, then return true else false.
- **No. Attr/ No. Classes** = Calculate the average number for the diagram from the number of attributes taken from the classes.

- **No. Attr/ No. Classes that contains attributes** = Calculate the average number of attributes for classes with at least one attribute in the class.
- **NumOps Quartile 25%** = Calculate the 1st quartile from the number of operations taken from the classes of the diagram.
- **NumOps Quartile 75%** = Calculate the 3rd quartile from the number of operations taken from the classes of the diagram.
- **Low No. of Operations (Quartile)** = Compare if the number of operations in the selected class is less or equals to the value of NumOps Quartile 25%, then return true else false.
- **High No. Operations (Quartile)** = Compare if the number of operations in the selected class is larger than the value of the NumOps Quartile 75%, then return true else false.
- **New High No. Operations (Quartile)** = Number of operations is not zero and also larger or equal to the rounded down integer sum of the NumOps Quartile 75% and Average No. Operations.
- **No. Opr/ No. Classes** = Calculate the average number of operations for all the classes in the diagram.
- **No. Opr/No. classes with at least one operation** = Calculate the average number of operations for classes with at least one operation in the class.
- **Coupling Quartile 25%** = Calculate the 1st quartile from the number of associations in the diagram.
- **Coupling Quartile 75%** = Calculate the 3rd quartile from the number of associations in the diagram.
- **Average No. Coupling** = Calculate the average number for the diagram from the number of associations with the classes.
- **Low No. of Coupling (Quartile)** = Compare if the number of associations with the selected class is less or equals to the value of Coupling Quartile 25%, then return true else false.
- **High No. Coupling (Quartile)** = Compare if the number of associations with the selected class is larger than the value of the Coupling Quartile 75%, then return true else false.
- **New High No. Coupling (Quartile)** = Number of associations is not zero and also larger or equal to the rounded down integer sum of the Coupling Quartile 75% and Average No. Coupling.

See the computations in pseudo code written algorithms in Appendix B.

5. 1. 1. The Detection of Complex Class

The characteristics of Complex class is that it consist of a large number of operations while as well the number of coupling (associations) is high. We have used four different detection strategies to measure for these symptoms. Also see Appendix B. 1.

- ***C1:*** According to this rule, if **New High No. Operations (Quartile)** is true and **New High No. Coupling (Quartile)** is true and **NumOps** is larger than **No. Opr/ No. Classes**, then the class is infected with the Complex Class anti-pattern.
- ***C2:*** According to this rule, if **New High No. Operations (Quartile)** is true and **New High No. Coupling (Quartile)** is true and **NumOps** is larger than **No. Opr/No.**

classes with at least one operation, then the class is infected with the Complex Class anti-pattern.

- ***C3:*** According to this rule, if **New High No. of Operations (Quartile)** is true and **New High No. Coupling(Quartile)** is true, then the class is infected with the Complex Class anti-pattern.
- ***C4:*** According to this rule, if **High No. of Operations (Quartile)** is true and **High No. of Coupling (Quartile)** is true, then the class is infected with the Complex Class anti-pattern.

5. 1. 2. The Detection of Large Class

Large Class, just as its name describes, is a class that contains an overwhelming number of methods stuffed with hundreds of lines of code. We have used four different detection strategies to measure for these symptoms. Also see Appendix B. 2.

- ***LAR1:*** According to this rule, if **New High No. of Operations (Quartile)** is true and **New High No. of Coupling (Quartile)** is false and **NumOps** is larger than **No. Opr/ No. Classes**, then the class is infected with the Large Class anti-pattern.
- ***LAR2:*** According to this rule, if **New High No. of Operations (Quartile)** is true and **New High No. of Coupling (Quartile)** is false and **NumOps** is larger than **No. Opr/No. classes with at least one operation,** then the class is infected with the Large Class anti-pattern.
- ***LAR3:*** According to this rule, if **High No. of Operations (Quartile)** is true and **New High No. Coupling (Quartile)** is false, then the class is infected with the Large Class anti-pattern.
- ***LAR4:*** According to this rule, if **High No. of Operations (Quartile)** is true and **High No. of Coupling (Quartile)** is false, then the class is infected with the Large Class anti-pattern.

5. 1. 3. The Detection of Lazy Class

Lazy Class anti-pattern is a class that is in lack of children and fields just as attributes and operations. We have used the following detection strategy to measure for these symptoms. Also see Appendix B. 3.

- ***LAZ:*** According to this rule, if **Low No. of Attributes (Quartile)** is true and **Low No. of Operations (Quartile)** is true and **Low No. of Coupling (Quartile)** is true and **NOC** is 0, then the class is infected with the Lazy Class anti-pattern.

5. 1. 4. The Detection of ManyFieldAttributesButNotComplex (MFABNC)

MFABNC is a class with high number of attributes but low number of operations. We have used three different detection strategies to measure for these symptoms. Also see Appendix B. 4.

- **M1:** According to this rule, if **High No. of Attributes (Quartile)** is true and **Low No. of Operations (Quartile)** is true and **Low No. of Coupling (Quartile)** is true, then the class is infected with the MFABNC anti-pattern.

- **M2:** According to this rule, if **High No. of Attributes (Quartile)** is true and **Low No. of Operations (Quartile)** is true and **Low No. of Coupling (Quartile)** is true and **NAtt** is larger than **No.Atrr/ No. Classes,** then the class is infected with the MFABNC anti-pattern.
- **M3:** According to this rule, if **High No. of Attributes (Quartile)** is true and **Low No. of Operations (Quartile)** is true and **Low No. of Coupling (Quartile)** is true and **NAtt** is larger than **No. Attr/ No. Classes that contains attributes,** then the class is infected with the MFABNC anti-pattern.

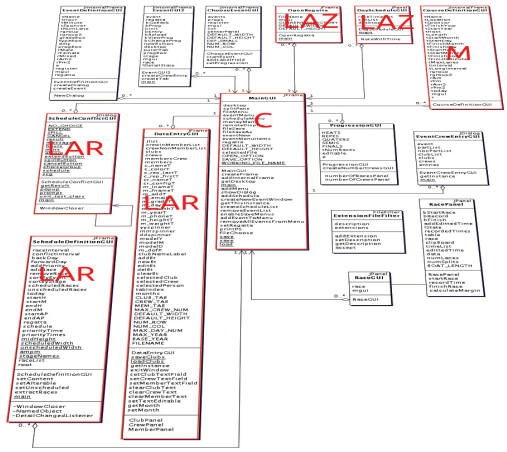

Fig. 3. Anti-pattern detection example.

Figure 3 showing a real example of a UML class diagram that is infected by the detection-desired anti-patterns of ours. According to our detection algorithms, the conclusion was based on the follwoing calculations: since in this diagram, the 1st quartile of the (attributes/operations/coupling) is (5/2/1), the 3rd quartile is (20/6/2), and the average number is (15/5/2), therefore when the detection rules of the anti-patterns compared these metric values to the metrics of each classes-those that have been found over or under the range- were concluded true.

Anti-pattern name	***Complex Class***	***Large Class***	***Lazy Class***	***MFABNC***
Detection rule applied	C1, C2, C3, C4	LAR1, LAR2, LAR3, LAR4	LAZ	M1, M2, M3

Table 5. Summary of RADAR detection rules applied for anti-patterns.

5.2. The Comparison of RADAR Detection to Ptidej Detection

Table 6 below represent the total number of found anti-patterns in both data sets, where the results including duplicates regarding to those classes of the diagrams that were caught more than once by different strategies per anti-pattern category. See Table 6.

Data set Models41					**Data set Models22**				
RADAR test				**Ptidej test**	**RADAR test**				**Ptidej test**
C1	***C2***	***C3***	***C4***	***Complex Class***	***C1***	***C2***	***C3***	***C4***	***Complex Class***
8	8	8	41	18	4	4	4	11	11
LAR1	***LAR 2***	***LAR 3***	***LAR 4***	***Large Class***	***LAR1***	***LAR2***	***LAR 3***	***LAR 4***	***Large Class***
20	17	55	33	0	19	16	61	53	0
LAZ				***Lazy Class***	***LAZ***				***Lazy Class***

139			18	146			0
M 1	***M 2***	***M 3***	***MFABNC***	***M 1***	***M 2***	***M 3***	***MFABNC***
14	13	10	0				0

Table 6. Total number of observed anti-patterns.

The anti-pattern categories are represented by the type of strategies under RADAR. Therefore C (1-4) stands for Complex Class, LAR (1-4) for Large Class, LAZ for Lazy Class and M (1-3) is for ManyFieldAttributesButNotComplex. While under the results of Ptidej, the last anti-pattern type MFABNC is an abbreviation of the latter mentioned.

After the elimination of the duplicates, we have interpreted the following results. RADAR could detect a total of 249 anti-patterns out from the 575 classes from Models41 data set, where the division regarding to the four types of anti-patterns is 41 Complex Class, 55 Large Class, 139 Lazy Class, and 14 ManyFieldAttributesButNotComplex (BFABNC). While in Models22 data set, this total number is 220 with the division of 11 Complex Class, 59 Large Class, 146 Lazy Class, and 4 MFABNC anti-patterns. On the other hand, we have observed the following results from Ptidej test. 18 Complex Class and 18 Lazy Class for Models41 data set, which sums up a total of 36 anti-patterns only. Regarding to Models22 data set, the total number is 11 by the detection of Complex Class. The tests we performed with Ptidej tool regarding to the detection of Large Class and MFABNC were completely unsuccessful. However, we have unintentionally discovered the fact that this undesired outcome is due to code generation. As we generated source code from the XMI files of UML class diagrams, some information such as the lines of code (LOC SD metric) was arbitrarily generated (e.g. to one single line) by the software we used. The lack of the LOC then caused trouble for Ptidej tool when its detection technique attempted to measure that, which if is long, is actually one of the main symptom of Large Class and MFABNC anti-patterns. Therefore, we could only compare the test results between Complex Class and Lazy Class anti-patterns. Regarding to the percentage of coverage from the aspect of detecting the same classes per anti-pattern type, we interpreted the following results. Out from the 18 classes that Ptidej judged as containing the Complex Class anti-pattern RADAR found its 16.67%, while from the 18 Lazy Class symptomed classes by Ptidej, RADAR located 61.11% of the same classes in Models41. On the other hand, Ptidej could only find 18 Complex Class symptomed classes in Models22, therefore we could only compare the same results of Complex Class in the second data set, where RADAR covered 36.36% of those. See Appendix C. 1. Regarding to the hypotheses, the tests resulted the following:

- **H0 Set1**: Since the P-value from Chi Square test is less than 0.05, we can reject the H0 Set1 and there are differences between two result sets. However, from Mann-Whitney U test result we could only disapprove H0 Set1 for Lazy Class.

- **H1 Set1:** Since the P-value from Chi Square test is less than 0.05, we can approve H1 and there are differences between two result sets. However, from Mann-Whitney U test result we could only approve H1 for Lazy class.

- **H0 Set2:** Since the P-value from Chi Square test is less than 0.05, we can reject H0 Set2 and there are differences between two result sets. However, from Mann-Whitney U test result we failed to reject H0 Set2.

- **H1 Set2:** Since the P-value from Chi Square test is less than 0.05, we can approve H1 Set2 and there are differences between two result sets. However, from Mann-Whitney U test result we failed to approve H1 Set2. See Appendix C. 2. And C. 3.

5.3. The Evaluation of RADAR Approach

Regarding to the feedback we received from student A, he judged the average accuracy of RADAR regarding to the detection of Complex Class as 68%, 75 % for Large Class, 99% for Lazy Class, and the accuracy of 55% for MFABNC. Meanwhile he gave comments in four cases instead of a decision whether he agrees or disagrees with our results. According to the feedback from student B, he assessed the average detection accuracy regarding to Complex Class as 70%, 38% for Large Class, 80% for Lazy Class, and the accuracy of 80% for detecting the MFABNC anti-pattern. On the other hand, he did not provide any answer in 17 cases and unfortunately we could not request another assessment due to the reason that we received his answer on the same day as the end of the deadline that was determined to this thesis project. According to the occurrence of agreement from the reviewers, the average accuracy of RADAR in the detection of Complex Class is between 68%-70%, 38%-75% of Large Class, 80%-99% of Lazy Class, and the accuracy between 55%-80% for the detection of MFABNC anti-pattern. See Figure 4 and Appendix D.

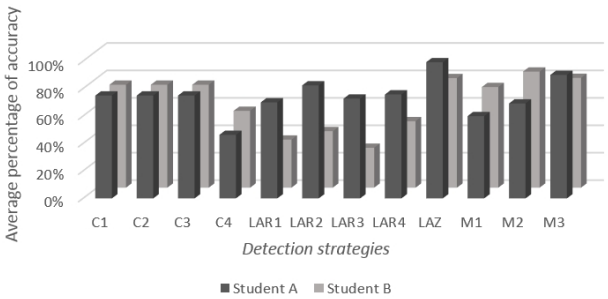

Fig. 4. Summary of the accuracy denoted to the detection strategies.

A UML model could be meant to be the blueprint of the system, therefore the design should be not only error-free, but as well capably and thoughtfully designed in order to reduce the possibility of issues that anti-pattern may cause with their presence [2,3,4,11,12]. Authors claimed that it is difficult to realize the creation of anti-patterns, especially when that was unintentionally caused by designers of inexperienced in OO language. On the other hand, writing the source code of UML designs for large scale and complex OO systems is generally a challenging task, therefore is why the realization and capture of anti-patterns is laying upon the involvement of system maintainers [6,7,10,11]. The issue of that is not only that their work takes place in a later time of the software development process, but as well claimed as very time consuming procedure that requires high level of management regarding to resources and budget. There are various anti-pattern detection solutions exists to moderate the work of the designer, which are mostly metric based and semi-automatic approaches [1,5,8,10,11][15-19] that unburdens the discovery procedure and realization of anti-patterns by overtaking most of the work via the automatic localization of those. However, these solutions do not consider the detection of anti-patterns in UML class diagrams. These are the reasons that inspired us to come up with our solution that could be used to bring forward the actions to

avert anti-patterns. Hence the benefits of running detection on UML class diagrams at the design level could enable the designer to avoid anti-patterns during the design time. With the appropriate warnings such as which class is infected, s/he could facilely interpret and correct those due to visual appearance provided by the model view of the editor software in use. The solution of RADAR could be implemented as a plugin for visual modeling editor tools such as Enterprise Architect, StarUML or Visual Paradigm, where the designer could run a quick detection even on small increments of the class diagrams since our algorithms automatically handles the flexibility regarding to the sizes. On the other hand, the drawback of the design level detection is the lack of measurable properties, meaning that we will not be able to see the amount of code implemented in the operations, which could be one of the symptoms of some anti-pattern types. This issue was found when we were generating source code to run detection tests with Ptidej tool. The tool uses the measurement of the SD metric called LOC, which stands for the lines of code, and concludes some anti-patterns depending on the length of these lines. The problem with this is that software tools may arbitrarily generate the lines of code, which can be generated on one single line, thereby causing false measurement.

6. Conclusion

This paper is oriented to contribute with a solution to the detection of anti-patterns in UML class diagrams. Therefore, we introduced our approach called RADAR that is designed for detecting the anti-patterns inter alia, *Complex Class, Large Class, Lazy Class, and ManyFieldAttributesButNotComplex (MFABNC)*. The purpose of choice regarding to the quantitative selection of anti-pattern types to be detected was not considered as one of the goals for our solution. We selected these few out of the multiform types rather to serve as examples for the introduction of algorithmic procedures required to scan for symptoms in class diagram designs, with the intention to provide initial guidance to the readers. With the aim of our research, we investigated the pieces of the puzzle in order to present the combination of those SD metrics and mathematical solutions that we used to measure properties of UML class diagrams in the capture of anti-patterns based on their symptoms, while as well considering the size of classes relative to one another in each diagrams. The solution we provided with RADAR is then the use of SD metrics regarding to the *number of (attributes, operation, associations, children)* as numeric representation of UML class diagram properties into the detection algorithms, while the algorithms as well perform the calculations of quartile and average on those unique values to compare the sizes differences in each class diagrams. Eventually, we have imposed RADAR for a statistical analysis in order to measure its accuracy compared to a trusted anti-pattern detection tool, Ptidej v5.8.1. The statistical analysis, regarding to the test coverage from the aspect of what percentage can our RADAR approach locate from those classes per anti-pattern categories that Ptidej does, resulted to the average of 26% accuracy for RADAR in the detection of Complex Class, and 61% for the detection of Lazy Class symptoms. Unfortunately, this measurement could not be made for Large Class and MFABNC anti-patterns based on the reason that Ptidej tool could not detect those. We claimed this problem as due to code generation procedure by the software we used regarding to the arbitrarily generated lines of code (LOC), which is if is long, is actually one of the main symptom of the latter mentioned two anti-patterns. Therefore, we cannot confirm the correctness of RADAR based on this comparison. To compensate the deficiencies of the measurement, we requested a review for RADAR from PhD students. Regarding to the feedback we received from them, the average accuracy of RADAR in the detection of the four anti-patterns is somewhat high. However, due to the low

number of participants we cannot strongly confirm the correctness of our RADAR solution, and our conclusion is that furthermore measurements will be required to take based on this reason. On the other hand, the future project could be the implementation of RADAR algorithms as plugin for Enterprise Architect visual modeling and design tool, and the conduction of a software experiment involving large group of people.

7. Acknowledgements

We thank Michel Chaudron and Bilal Karasneh for assisted us with their guidance toward this project, moreover to Rodi Jolak and Truong Ho Quang for their feedback.

8. References

[1] Moha, N., Gueheneuc, Y. G., Duchien, L., & Le Meur, A. (2010). DECOR: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, *36*(1), 20-36.

[2] V. Rompaey, B. D. Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," IEEE Transactions on Software Engineering, vol. 33, no. 12, pp. 800–817, 2007 [3] David, V. (2006). UML class diagrams.

[4] Saxena, V., & Kumar, S. (2012). Impact of Coupling and Cohesion in Object-Oriented Technology. *Journal of Software Engineering and Applications*, *5*(09), 671.

[5] Fourati, R., Bouassida, N., & Abdallah, H. B. (2011). A metric-based approach for antipattern detection in UML designs. In *Computer and Information Science 2011* (pp. 17-33). Springer Berlin Heidelberg.

[6] Budgen, D. (2003). *Software design*. Pearson Education.

[7] McCormick, H. W., Mowbray, T. J., & Malveau, R. C. (1998). AntiPatterns: refactoring software, architectures, and projects in crisis.

[8] Moha, N., Gueheneuc, Y. G., & Leduc, P. (2006, September). Automatic generation of detection algorithms for design defects. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on* (pp. 297-300). IEEE.

[9] Khomh, F., Vaucher, S., Guéhéneuc, Y. G., & Sahraoui, H. (2011). BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, *84*(4), 559-572.

[10] Guéhéneuc, Y. (2007, October). Ptidej: A flexible reverse engineering tool suite. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*(pp. 529-530). IEEE.

[11] G. Bruno, P. Garza, E. Quintarelli, and R. Rossato, "Anomaly detection in xml databases by means of association rules," in DEXA '07: Proceedings of the 18th International

Conference on Database and Expert Systems Applications. Washington, DC, USA: IEEE Computer Society, 2007, pp. 387–391

[12] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan, "Automating the detection of snapshot isolation anomalies," in VLDB '07: Proceedings of the 33rd international conference on Very large data bases. VLDB Endowment, 2007, pp. 1263–1274.

[13] Moha, N., Guéhéneuc, Y. G., Le Meur, A. F., Duchien, L., & Tiberghien, A. (2010). From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, *22*(3-4), 345-361.

[14] Munro, M. J. (2005, September). Product metrics for automatic identification of" bad smell" design problems in java source-code. In *Software Metrics, 2005. 11th IEEE International Symposium* (pp. 15-15). IEEE.

[15] Marinescu, R. (2004, September). Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on* (pp. 350-359). IEEE.

[16] Trifu, A., Seng, O., & Genssler, T. (2004, March). Automated design flaw correction in object-oriented systems. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on* (pp. 174-183). IEEE.

[17] AliKacem, H., & Sahraoui, H. (2006). Détection d'anomalies utilisant un langage de description de règle de qualité, in actes du 12e colloque LMO. *LMO, Ed*.

[18] Grotehen, T., & Dittrich, K. R. (1997). The methood approach: Measures, transformation rules and heuristics for object-oriented design.

[19] Ballis, D., Baruzzo, A., & Comini, M. (2008). A rule-based method to match Software Patterns against UML Models. *Electronic Notes in Theoretical Computer Science*, *219*, 51-66.

[20] Montgomery, D., Design and Analysis of Experiments, John Wiley & Sons, USA, 2000.

[21] AvAlan Hevner,Samir Chatterjee. (2010). Design Research in Information Systems: Theory and Practice. (pp. 270-271).

[22] Blessing, L. T., & Chakrabarti, A. (2009). DRM, a design research methodology. Springer Science & Business Media.

[23] Keele, S. (2007). *Guidelines for performing systematic literature reviews in software engineering* (pp. 1-57). Technical report, EBSE Technical Report EBSE-2007-01.

[24] Jones, J. C. (1963). A method of systematic design. In *Conference on design methods* (pp. 53-73). Pergamon Press, Oxford/New York.

[25] Box, G. E., Hunter, W. G., & Hunter, J. S. (1978). Statistics for experimenters.

[26] Satorra, A., & Bentler, P. M. (2001). A scaled difference chi-square test statistic for moment structure analysis. *Psychometrika*, *66*(4), 507-514.

[27] McKnight, P. E., & Najab, J. (2010). Mann-Whitney U Test. *Corsini Encyclopedia of Psychology*.

Appendix A. Existing Anti-patterns

Anti-pattern name	Most applicable scale	Root causes	Unbalanced forces
Development anti-patterns			
Blob	Application	Sloth, Haste	Management of Functionality, Performance, Complexity
Lava Flow	Application	Avarice, Greed, Sloth	Management of Functionality, Performance, Complexity
Functional Decomposition	Application	Avarice, Greed, Sloth	Management of Complexity, Change
Poltergeists	Application	Sloth, Ignorance	Management of Functionality, Complexity
Golden Hammer	Application	Ignorance, Pride, Narrow−Mindedness	Management of Technology Transfer
Spaghetti Code	Application	Ignorance, Sloth	Management of Complexity, Change
Cut−and−Paste Programming	Application	Sloth	Management of Resources, Technology Transfer
Software architecture anti-patterns			
Stovepipe Enterprise	Enterprise	Haste, Apathy, Narrow−Mindedness	Management of Change, Resources, Technology Transfer
Stovepipe System	System	Haste, Avarice, Ignorance, Sloth	Management of Complexity, Change
Vendor Lock−In	System	Sloth, Apathy, Pride/Ignorance (Gullibility)	Management of Technology Transfer, Management of Change
Architecture by Implication	System	Pride, Sloth	Management of Complexity, Change, and Risk
Design by Committee	Global	Pride, Avarice	Management of Functionality, Complexity, and Resources
Reinvent the Wheel	System	Pride, Ignorance	Management of Change, Technology Transfer
Software project management anti-patterns			
Analysis Paralysis	System	Pride, Narrow−Mindedness	Management of Complexity
Death by Planning	Enterprise	Avarice, Ignorance, Haste	Management of Complexity
Corncob	Enterprise	Avarice, Pride, Narrow−Mindedness	Management of Resources, Technology Transfer
Irrational Management	Enterprise	Responsibility (the universal cause)	Management of Resources
Project Mismanagement	Enterprise	Responsibility (the universal cause)	Management of Risk (the universal force)
Mini anti-Patterns			
Ambiguous Viewpoint, Autogenerated Stovepipe, Blowhard Jamboree, Boat Anchor, Continuous Obsolescence, Cover Your Assets, Dead End, E−mail Is Dangerous, Fear of Success, The Feud, Fire Drill, The Grand Old Duke of York, Input Kludge, Intellectual Violence, Jumble, Mushroom Management, Smoke and Mirrors, Swiss Army Knife, Throw It over the Wall, Viewgraph Engineering, Walking through a Mine Field, Warm Bodies, Wolf Ticket.			

Table 7. Collection of existing anti-patterns. (Resource reference: McCormick et al. [7]).

Appendix B. Algorithms of RADAR

RADAR detection solution:

- **Given that we have defined a function to calculate quartile**:
  function quartileCalc(quartile, array):
  m_index = (quartile/4)*(array.length-1)+1
  remainder = m_index % 1
  m_index = m_index - remainder
  result = array[m_index]+(remainder*(array[m_index+1]-array[m_index]))return result.

- **Calculations from the number of attributes**:
  - NumAttr Quartile 25% = quartileCalc(1, numberOfAttributeArray)
  - NumAttr Quartile 75% = quartileCalc(3, numberOfAttributeArray)
  - Low No. Attributes (Quartile) = (numberOfAttributes<=quartileCalc(1,numberOfAttributeArray)
  - High No. of Attributes (Quartile) = (numberOfAttributes>quartileCalc(3,numberOfAttributeArray) &&numberOfAttributes>quartileCalc(1, numberOfAttributeArray))
  - No. Attr/ No. Classes = totalNumberOfAttributesInModel/totalNumberOfClassesInModel
  - No. Attr/ No. Classes that contains attributes = totalNumberOfAttributesInModel/totalNumberOfClassesContainsAttributesInModel

- **Calculations from the number of operations**:
  - NumOps Quartile 25% = quartileCalc(1, numberOfOperationsArray)
  - NumOps Quartile 75% = quartileCalc(3, numberOfOperationsArray)
  - Low No. of Operations (Quartile) = (numberOfOpetaions<=quartileCalc(1, numberOfOperationsArray))
  - High No. Operations (Quartile) = (numberOfOpetaions>=quartileCalc(3, numberOfOperationsArray)
  - && numberOfOpetaions>quartileCalc(1, numberOfOperationsArray))
  - Average No. Operation = totalNumberOfOperations/totalNumberOfClassesContainsOperationsInModel
  - New High No. Operations (Quartile) = (numberOfOpetaions>=round(High No. Operations(Quartile)+Average No. Operation))

- **Calculations from the number of coupling**:
  - Coupling Quartile 25% = quartileCalc(1, numberOfCouplingsArray)
  - Coupling Quartile 75% = quartileCalc(3, numberOfCouplingsArray)
  - Low No. of Coupling (Quartile) = (numberOfCouplings<=quartileCalc(1, numberOfCouplingsArray))
  - High No. Coupling (Quartile) = (numberOfCouplings>=quartileCalc(3, numberOfCouplingsArray) && numberOfCouplings>quartileCalc(1, numberOfCouplingsArray))
  - Average No. Coupling = totalNumberOfCouplings/totalNumberOfClassesContainsCouplingsInModel
  - New High No. Coupling (Quartile)= (numberOfCouplings>=round(High No. Couplings(Quartile)+Average No. Couplings)**)**

B. 1. Complex Class

C1:

```
if (numberOfOpetaions>=round(High No. Operations(Quartile)+Average No. Operation) &&
  numberOfCouplings>=round(HighNo.Couplings(Quartile)+AverageNo.Couplings)&&
  numberOfOpetaions>totalNumberOfOperations/totalNumberOfClassesInModel)
{
  print("is complex 1");
} else {
  print("is not complex 1");
}
```

C2:

```
if (numberOfOpetaions>=round(High No. Operations(Quartile)+Average No. Operation) &&
  numberOfCouplings>=round(HighNo.Couplings(Quartile)+AverageNo.Couplings)&&
  numberOfOpetaions>totalNumberOfOperations/totalNumberOfClassesContainsOperationsInModel)
{
  print("is complex 2");
} else {
  print("is not complex 2");
}
```

C3:

```
if (numberOfOpetaions>=round(High No. Operations(Quartile)+
Average No.Operation) &&
```

```
  numberOfCouplings>=round(HighNo.Couplings(Quartile)+AverageNo.Couplings))
{
  print("is complex 3");
} else {
  print("is not complex 3");
}
```

C4:

```
if ((numberOfOpetaions>=quartileCalc(3, numberOfOperationsArray)&& numberOfOpetaions>quartileCalc(1,
numberOfOperationsArray)) &&
  (numberOfCouplings>=quartileCalc(3, numberOfCouplingsArray) && numberOfCouplings>quartileCalc(1,
numberOfCouplingsArray)))
{
  print("is complex 4");
} else {
  print("is not complex 4");
}
```

B. 2. Large Class

LAR1:

```
if (numberOfOpetaions>=round(High No. Operations(Quartile)+Average No. Operation)&&
  numberOfCouplings<round(High No. Couplings(Quartile)+Average No. Couplings)&&
  numberOfOperations>totalNumberOfOperations/totalNumberOfClassesInModel)
{
  print("is large class 1");
} else {
  print("is not large class 1");
}
```

LAR2:

```
if (numberOfOpetaions>=round(High No. Operations(Quartile)+Average No. Operation)&&
  numberOfCouplings<round(High No. Couplings(Quartile)+Average No. Couplings)&&
  numberOfOperations>totalNumberOfClassesContainsOperationsInModel)
{
  print("is large class 2");
} else {
  print("is not large class 2");
}
```

LAR3:

```
if (numberOfOpetaions>=round(High No. Operations(Quartile)+Average No. Operation) &&
  numberOfCouplings<round(High No. Couplings(Quartile)+Average No. Couplings))
{
  print("is large class 3");
} else {
  print("is not large class 3");
}
```

LAR4:

```
if (numberOfOpetaions>=quartileCalc(3,numberOfOperationsArray)&&numberOfOpetaions>quartileCalc(1,
numberOfOperationsArray) &&
  !(numberOfCouplings>=quartileCalc(3,numberOfCouplingsArray)&&numberOfCouplings>quartileCalc(1,
numberOfCouplingsArray)))
{
  print("is large class 4");
} else {
```

```
  print("is not large class 4");
}
```

B. 3. Lazy Class

LAZ:

```
if ((numberOfAttributes<=quartileCalc(1, numberOfAttributeArray)&&numberOfOpetaions<=quartileCalc(1,
numberOfOperationsArray)&&numberOfCouplings<=quartileCalc(1, numberOfCouplingsArray)&&
numberOfChildren==0)
{
  print("is lazy class 1");
} else {    print("is not lazy
class 1"); }
```

B. 4. ManyFieldAttributesButNotComplex(MFABNC)

M1:

```
if((numberOfAttributes>quartileCalc(3,numberOfAttributeArray)&&numberOfAttributes>quartileCalc(1,
numberOfAttributeArray))&&numberOfOpetaions<=quartileCalc(1, numberOfOperationsArray)&&
numberOfCouplings<=quartileCalc(1, numberOfCouplingsArray))
{
  print("is MFABNC 1");
} else {
  print("is not MFABNC 1");
}
```

M2:

```
if(numberOfAttributes>quartileCalc(3, numberOfAttributeArray)&&numberOfAttributes>quartileCalc(1,
numberOfAttributeArray)&&numberOfOpetaions<=quartileCalc(1, numberOfOperationsArray)&&
numberOfCouplings<=quartileCalc(1, numberOfCouplingsArray)&&
numberOfAttributes>totalNumberOfAttributesInModel/totalNumberOfClassesInModel)
{
  print("is MFABNC 2");
} else {
  print("is not MFABNC 2");
}
```

M3:

```
if(numberOfAttributes>quartileCalc(3, numberOfAttributeArray)&&numberOfAttributes>quartileCalc(1,
numberOfAttributeArray)&&numberOfOpetaions<=quartileCalc(1, numberOfOperationsArray)
&&numberOfCouplings<=quartileCalc(1, numberOfCouplingsArray)&&
numberOfAttributes>totalNumberOfAttributesInModel/totalNumberOfClassesContainsAttributesInModel)
{
  print("is MFABNC 3");
} else {
  print("is not MFABNC 3");
    }
```

Appendix C. Test Results

C. 1. Matches between RADAR and Ptidej

Lazy Classes in models41 set (found by both)		**Complex Classes in models41 set (found by both)**	
XMI_ID	***Class Name***	***XMI_ID***	***Class Name***

1	Timer	1	DBManager
1	AutoResetEvent	4	Controller
1	TimerCallback	17	OrderHolon
1	SmtpClient	**Complex Classes in models22 set** (found by both)	
1	SqlConnection	***XMI_ID***	***Class Name***
1	Object	8	CImage
1	XmlDocument	8	CPrimitive
3	NativeMethods	8	CObject3D
18	Block	10	Annoyme
27	ParamError		
27	CircuitError		

Table 8. Matches found between RADAR and Ptidej.

C. 2. Chi Square test results between RADAR and Ptidej

	In data set Models41				**In data set Models22**			
Test type	***Observed (O) Anti-Pattern Types***				***Observed (O) Anti-Pattern Types***			
RADAR	Complex Class	Large Class	Lazy Class	MFABNC	Complex Class	Large Class	Lazy Class	MFABNC
	41	55	139	14	11	59	146	4
Ptidej	18	0	18	0	11	0	0	0
	Expected (E) Anti-Pattern Types				***Expected (E) Anti-Pattern Types***			
RADAR	Complex Class	Large Class	Lazy Class	MFABNC	Complex Class	Large Class	Lazy Class	MFABNC
	51.54736842	48.052632	137.1684211	12.2315789	20.952381	56.190476	139.04762	3.8095238
Ptidej	7.452631579	6.9473684	19.83157895	1.76842105	1.047619	2.8095238	6.952381	0.1904762
	(O-E) Anti-Pattern Types				***(O-E) Anti-Pattern Types***			
RADAR	Complex Class	Large Class	Lazy Class	MFABNC	Complex Class	Large Class	Lazy Class	MFABNC
	-10.54736842	6.9473684	1.831578947	1.76842105	-9.952381	2.8095238	6.952381	0.1904762
Ptidej	10.54736842	-6.9473684	-1.83157895	1.76842105	9.952381	-2.8095238	-6.952381	-0.1904762
	((O - E)²) Anti-Pattern Types				***((O - E)²) Anti-Pattern Types***			
RADAR	Complex Class	Large Class	Lazy Class	MFABNC	Complex Class	Large Class	Lazy Class	MFABNC
	111.2469806	48.265928	3.35468144	3.12731302	99.049887	7.893424	48.335601	0.0362812
Ptidej	111.2469806	48.265928	3.35468144	3.12731302	99.049887	7.893424	48.335601	0.0362812
	((O - E)² / E) Anti-pattern Types				***((O - E)² / E) Anti-Pattern Types***			
RADAR	Complex Class	Large Class	Lazy Class	MFABNC	Complex Class	Large Class	Lazy Class	MFABNC
	2.158150533	1.0044388	0.02445666	0.25567533	4.727381	0.1404762	0.3476191	0.0095238
Ptidej	14.92720785	6.9473684	0.169158565	1.76842105	94.547619	2.8095238	6.952381	0.1904762
	x^2 = 27.25487722 ***P*** = 0.00000520569				x^2 = **109.725** ***P*** = 0.00000000000000000000000125753			

Table 9. Chi Square test result of the data sets regarding to the detection of anti-pattern types between RADAR and Ptidej.

C. 3. Mann Whitney-U test results between RADAR and Ptidej

Hypothesis No.	*Hypothesis*	*P-value by Chi Square test*	*P and U-value by Mann-Whitney U test*	*Result*
H0 Set1	There is no difference in the result (i.e found number of anti-patterns/category) between the tests performed with RADAR and Ptidej tool on data set models41.	P = 0.00000520569	For Complex Class: The U-value is 634, which is less than expected U-value 840.5 and the data sets are normal distributed. However, the Z-Score is 1.9104 with a p-value 0.05614, so the result is not significant at p≤ 0.05. We failed to reject that there is no differences between complex classes found in two test results.	Since the P-value from Chi Square test is less than 0.05, we can reject the H0 and there is differences between two result sets. However, from Mann Whitney U test result we could only disapprove H0 for Lazy Class.
			For Lazy Class: The U-value is 226.5, which is less than expected U-value 1578 and the data sets are normal distributed. Furthermore, the Z-Score is 5.6896with a p-value 0, so the result is significant at p≤ 0.05. Therefore we successfully rejected H0. There is differences between lazy classes found in two test results.	
H1 Set1	There is a difference in the result (i.e. found number of anti-patterns/category) between the tests performed with RADAR and Ptidej tool on data set models41.	P=0.00000520569	For Complex Class: The U-value is 634, which is less than expected U-value 840.5 and the data sets are normal distributed. However, the Z-Score is 1.9104 with a p-value 0.05614, so the result is not significant at p≤ 0.05. We failed to approve that there is differences between complex classes found in two test results. For Lazy Class: The U-value is 226.5, which is less than expected U-value 1578 and the data sets are normal distributed. Furthermore, the Z-Score is 5.6896with a p-value 0, so the result is significant at p≤ 0.05. Therefore we successfully accepted H1. There is differences between lazy classes found in two test results.	Since the P-value from Chi Square test is less than 0.05, we can approve H1 and there is differences between two result sets. However, from Mann Whitney U test result we could only approve H1 for Lazy Class.
H0 Set2	There is no difference in the result (i.e. found number of anti-patterns/category) between the tests performed with RADAR and Ptidej tool on data set models22.	P = 0.00000000000000000001257 53	The U-value is 218.5, which is less than expected U-value 242 and the data sets are normal distributed. However, the Z-Score is 0.5399 with a p-value 0.05892, so the result is not significant at p≤ 0.05. We failed to reject that there is no differences between two test results.	Since the P-value from Chi Square test is less than 0.05, we can reject H0 and there is differences between two result sets. However, from Mann-Whitney U test result we failed to reject H0.
H1 Set2	There is a difference in the result (i.e. found number of anti-patterns/category) between the tests performed with RADAR and Ptidej tool on data set models22.	P = 0.00000000000000000001257 53	The U-value is 218.5, which is less than expected U-value 242 and the data sets are normal distributed. However, the Z-Score is 0.5399 with a p-value 0.05892, so the result is not significant at p≤ 0.05. We failed to approve that there is differences between two test results.	Since the P-value from Chi Square test is less than 0.05, we can approve H1 and there is differences between two result sets. However, from Mann-Whitney U test result we failed to approve H1.

Table 10. Testing the hypotheses.

D. The evaluation of RADAR by Reviewers

	From	*Agreed*	*Disagreed*	*Commented (Neither agreed or disagreed)*	*Percentage of Accuracy*	*Average of (%) Accuracy*
C1	8	6	-	"No. of operations is not so high and the coupling is somehow low in respect to the number of the classes within the model", "no. of operations is low, but however the coupling is high regarding the no. of the classes of the model".	75%	Complex Class 68%
C2	8	6	-		75%	
C3	8	6	-		75%	
C4	41	19	15		46.34%	
LAR1	20	14	6		70%	Large Class 75%
LAR2	17	14	3		82.35%	
LAR3	55	40	15		72.73%	
LAR4	33	25	8		75.76%	
LAZ	139	138	1		99.28%	Lazy Class 99 %
M1	15	9	5	"Not so many attributes".	60%	MFABNC 55 %
M2	13	9	3		69.23%	
M3	10	9	1		90%	

Table 11. Summary of the percentage of accuracy in Models41 by reviewer A.

	RADAR Results	*Agreed*	*Disagreed*	*Not Answered*	*Percentage of Accuracy*	*Average of (%) Accuracy*
C1	8	6	2	-	75%	Complex class 70 %
C2	8	6	2	-	75%	
C3	8	6	2	-	75%	
C4	41	23	18	-	56.1%	
LAR1	20	7	13	-	35%	Large Class 38 %
LAR2	17	7	10	-	41.18%	
LAR3	55	16	39	-	29.09%	
LAR4	33	16	7	7	48.48%	
LAZ	139	111	20	8	79.86%	Lazy Class 80 %
M1	15	11	4	-	73.33%	MFABNC 80%
M2	13	11	1	1	84.62%	
M3	10	8	1	1	80%	

Table 12. Summary of the percentage of accuracy in Models41 by reviewer B.