

# Are the clients of flawed classes (also) defect prone?

Radu Marinescu and Cristina Marinescu  
 LOOSE Research Group  
 “Politehnica” University of Timișoara, Romania  
 Email: {radu.marinescu, cristina.marinescu}@cs.upt.ro

**Abstract**—Design flaws are those characteristics of design entities (*e.g.*, methods, classes) which make them harder to maintain. Existing studies show that classes revealing particular design flaws are more change and defect prone than the other classes. Since various collaborations are found among the instances of classes, classes are not isolated within the source code of object-oriented systems. In this paper we investigate if classes using classes revealing design flaws are more defect prone than classes which do not use classes revealing design flaws. We detect four design flaws in three releases of Eclipse and investigate the relation between classes that use/do not use flawed classes and defects. The results show that classes that use flawed classes are defect prone and this does not depend on the number of the used flawed classes. This findings show a new type of correlation between design flaws and defects, bringing evidence related to an increased likelihood of exhibiting defects for classes that use classes revealing design flaws. Based on the provided evidence, practitioners are advised once again about the negative impact design flaws have at a source code level.

**Keywords**—source code, defects, design flaws, detection strategies, software repositories, empirical software engineering.

## I. INTRODUCTION

When developing object-oriented systems it is wise to code them according to different heuristics, patterns and best practices like the ones presented in [1], [2], [3], [4], [5]. When the source code is not written following such techniques, most of the times it reveals different design flaws (*i.e.*, code smells [6]) that hampers its maintenance.

A frequent category of design flaws we can find within the source code is called *Identity Disharmonies* [7]. This category includes classes which do not have an harmonious size, exhibit more than one responsibility and the contained data and operations do not collaborate harmoniously. Concretely, within this category we find entities revealing the following design flaws: Data Class [2], [6] (a data container without complex functionality), God Class [2] (a class that tends to centralize the intelligence of the system – also known as Large Class in [6]), Brain Class [7] (a complex class which has an excessive amount of intelligence) and Feature Envy [6] (a method which use more data of other classes instead their own classes).

Currently there are many empirical study which investigate if flawed entities (*i.e.*, entities revealing design flaws) make maintenance more difficult [8], change more often [9] and exhibit more defects <sup>1</sup> [10], [11] than entities which do

<sup>1</sup>bugs in the source code.

not reveal design flaws. All the mentioned studies analyze *in isolation* the entities revealing design flaws. According to the definition given by Booch in [12] object-oriented programming “is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships” classes do not stand in isolation within the source code of a software system. Consequently, we consider it worthwhile to take a look also at the classes that use (*i.e.*, the clients of) classes revealing design flaws in order to see if the clients of classes revealing design flaws are more difficult to maintain, change more often and exhibit more defects than the clients of classes which do not reveal design flaws. This way, we investigate if the positive correlations between a hampered maintenance, change/defect proneness and classes that reveal design flaws are spread also into their clients.

In this paper we perform an empirical study whose goal is to explore the relations between the clients of entities revealing identity disharmony design flaws and defects. Concretely, we investigate if

- clients of classes revealing identity disharmonies have an increased likelihood to exhibit defects than the clients of classes that do not reveal identity disharmonies.
- the greater the number of flawed classes used, the greater the likelihood for a client to exhibit defects.

The paper is structured as follows: in Section 2 we relate our empirical study to existing works. In Section 3 we explain how we extracted the data involved in this study. In the first part of the next section (Section 4) we present the context of the study as well as the addressed research questions. We continue with a brief description of the employed statistical tests followed by the presentation of the performed steps within our study. We end the section by pointing out the results of the study. The threats to the validity are presented in Section 5. In the last section (Section 6) we summarize the results and hint towards future work.

## II. RELATED WORK

To our best knowledge an empirical study which investigates if the clients of classes with identity disharmonies have an increased likelihood to exhibit more defects than clients

of classes that do not reveal identity disharmonies has not already been done. Since there are several works which address different problems related to design flaws and defect proneness of classes in this section we present the ones we consider closest to our study.

*Detection of design flaws.* Fowler defined in [6] a set of 22 design flaws which are considered to hamper the maintenance of object-oriented software systems. Most of the time it is desirable to get rid off the flawed entities and in order to do this the first step is to find the existing flawed entities within the systems. Finding manually flawed entities is time-consuming and this was the main reason automatic detection techniques of design flaws appeared. Probably the most used automatic approach for finding entities affected by various design flaws is the metrics-based technique. Currently there are many design flaws that can be detected automatically (like the ones proposed in [13] [14] [7] [9]) and different tools accompany the extraction of design flaws. Usually the tools parse the source code in order to extract the necessary information (*e.g.*, [15] [16] [17]) and the inspection of the source code is not integrated into the development process. We believe this characteristic is a temporal one and many tools (like the one introduced in [18]) that transform quality assessment and code inspections from a standalone activity, into a continuous, agile process, fully integrated into the development life-cycle will appear. [19] introduces a bayesian approach to detection of code and design smells and it is claimed this approach provides a better precision and recall for the identification of the Blob antipattern than the metrics-based approach from [16]. Since this approach is quite new and further work is needed in order to prove that the precision and recall are also better for the identification of identities disharmonies, we use in the current paper a metrics-based approach for identifying the design flaws.

As it is emphasized in [13], a list of code smells can never be complete due to the specificity of the application. For example, within an enterprise application not every class that contains public fields, accessor methods and detected as being affected by the Data Class design flaw is considered harmful. Based on the specificity of enterprise applications, in [20] is proposed an approach which provides only those Data Classes which are not used in order to carry data between a client (the domain layer) and a server (the data source layer) in order to reduce the number of fine-grained method calls. An approach that captures the extent to which the clients of a hierarchy polymorphically manipulate the objects defined in the hierarchy is presented in [21]. This approach can also be used to identify groups of subtypes that are treated in a particular manner (non-polymorphically) by their clients. Taking into account the starting point of this work (a hierarchy of classes cannot be meaningfully validated in isolation and that it can only be validated in terms of its clients) we also consider that the impact of design flaws on defect proneness of classes cannot be fully validated in isolation and it also should be validated also in terms of its

clients.

*Detection of defects.* Finding defects for software entities using versions archives like CVS, SVN and bug tracking systems like Jira and Bugzilla seems to be the most widespread method. It was used in various work related to the identification of defects [22] [23] [11] and this is the method we employ in the current empirical study. Details about how defects can be found using the two data sources are presented in Chapter 27, Mining Your Own Evidence, from *Making Software* [24]. In [23] a distinction is made between defects which were identified during development and testing of a system (pre-release defects) and defects which were identified after the system has been deployed (post-release defects).

*Prediction of changes and defects based on metrics.* Paper [23] presents a set of experiments that shows that complexity metrics, in combination, can predict defects, suggesting that the more complex source code is, the more defects it has. In [25] the concept of change burst is introduced. Based on metrics related to change burst the defect prone components were predicted with a precision and recall greater than 90%.

In [10] entities revealing Shotgun Surgery, God Class and God Methods design flaws were positively associated with the number of exhibited defects. A new study [11] tackling the same area suggests that God and Brain Classes are positively associated with the number of exhibited defects, without normalization with respect to size. But when classes are normalized with respect to their size, God and Brain Classes exhibit less defects than other classes. In this work, like in [10], [11], we do not intend to take design flaws as better predictors of defects than metrics. Instead, we want to inform, both researchers and practitioners, whether any correlations exist among clients of classes revealing design flaws and defects.

### III. DATA COLLECTION

In this work we inspect the correlations (if any) between clients of flawed classes and defects on three releases (2.0, 2.1 and 3.0) of Eclipse<sup>2</sup>. The mentioned releases of Eclipse were heavily used within various empirical studies like the ones in [23], [10]. Extracting the required data involves two steps, and we dedicate the next two sections to each of these steps.

#### A. Extracting entities from the source code.

In order to extract the design entities (*e.g.*, methods, classes, calls of methods, accesses of attributes) from the source code we use an enhanced version of the IPLASMA[15]<sup>3,4</sup> environment.

IPLASMA is an integrated environment for quality analysis of object-oriented software systems that includes support for different phases of analysis – from model extraction (including scalable parsing for C++, Java and lately C#) up to high-level

<sup>2</sup><http://www.eclipse.org>.

<sup>3</sup>Integrated Platform for software modelling and analysis.

<sup>4</sup>a previous version can be downloaded from <http://loose.upt.ro/iplasma>.

Name of Data	Description
fileName	location of the class (relative path)
className	name of the class
type	0 - normal class 1 - abstract class 2 - interface
isDataClass	1 - entity is detected as a DataClass, 0 otherwise
isGodClass	1 - entity is detected as a GodClass, 0 otherwise
isBrainClass	1 - entity is detected as a BrainClass, 0 otherwise
isFeatureEnvy	1 - entity is detected as a FeatureEnvy, 0 otherwise
useDataClass	1 - entity make use of a least one class detected as a DataClass, 0 otherwise
useGodClass	1 - entity make use of a least one class detected as a GodClass, 0 otherwise
useBrainClass	1 - entity make use of a least one class detected as a BrainClass, 0 otherwise
useFeatureEnvy	1 - entity make use of a least one class detected as a FeatureEnvy, 0 otherwise
noUsedFlawedClasses	number of distinct used classes revealing design flaws
pre	number of pre-release defects the entity exhibits
post	number of post-release defects the entity exhibits

Fig. 1: The Structure of the Extracted Data.

metrics-based analysis or detection of code duplication. This environment relies on the MEMORIA [26] meta-model which specifies the main entities which are extracted from the source code. MEMORIA is a meta-model that can represent Java, C++ and C# systems in a uniform manner. For Java systems the IPLASMA environment uses the open-source model loader library called Recoder [27] in order to extract all the necessary information in terms of the MEMORIA meta-model. We create within this environment a new analysis which provides for each class having the same name as the file it belongs to the first 12 values described in Figure 1. These values are stored within a csv (comma-separated values) file which is further processed by the R environment <sup>5</sup> [28].

In [29] was performed a manual investigation whose goal was to determine if the classes belonging to the data-source layer were accurately captured by IPLASMA from the source code. The investigation revealed a high degree of precision and a recall close to 100%. An in-depth analysis showed that the only cause for having this recall was due to the Java 5 constructions, which were not properly captured. Since IPLASMA was the subject of many improvements, we consider that investigation deprecated. Consequently, in order to be able to determine the accuracy of the data extracted from the source code we performed a new investigation.

We took as a benchmark the data extracted from Eclipse 2.0 [23] by Zimmermann et al. and compared with the data provided by IPLASMA. In terms of number of packages a perfect match was found (377 packages). Regarding the classes, the benchmark contains 6730 files (classes) while IPLASMA captured only 6328 classes (consequently, a recall of 94%). A manual investigation reveals two main causes for this recall:

- Eclipse 2.0 contains some parallel hierarchies of classes for building the user-interface (e.g., carbon, motif, win32). We noticed multiple classes with the same name,

belonging to the same package, most of the times one for each targeted platform. IPLASMA is able to extract information from java source code which does not contain compilation errors and, consequently, only one hierarchy for building the user-interface was extracted. Probably when Eclipse was compiled, a flag that states exactly which is the targeted platform was set.

- Eclipse 2.0 contains some iterators named enum and since Java 1.5 enum is a keyword (e.g., *EditorRegistry.java*, *CellEditorActionHandler.java*). Consequently the information about those classes could not have been loaded since the used version of iPlasma is able to deal with Java 5 constructs.

We continue the investigation and manually take a look at every class in the source code whose row ID extracted in the csv file is multiple of 100 (63 files were investigated). A perfect match between the extracted data in the csv file and the source code in various terms (e.g., number of attributes and methods the class has) was found. Consequently, our investigation reveals a high degree of precision, near 100%.

The identity disharmonies classes reveals (Data Class, God Class, Brain Class and Feature Envy) are found by using the detection strategies presented in [7]. The same detection strategies were used in many empirical studies (e.g., [10], [11]). Next, we briefly present their main characteristics, but more details about the used detection strategies are presented in [7].

A class is considered as being affected by the Data Class design flaw if the interface of the class reveals data rather than offering services, the class reveals many attributes and is not complex.

A class is detected as a God Class if it uses directly more than a few attributes of other classes, has a very high functional complexity and a low cohesion.

We consider a class being a Brain Class if it is very complex and non-cohesive, contains more than one Brain Method and is very large or contains only one Brain Method but is extremely large and complex. We consider a Brain Method an excessively large method that has many conditional branches, deep nesting and uses many variables.

A class is affected by the Feature Envy design flaw if it has at least one Feature Envy method. A method is Feature Envy if it uses directly more than a few attributes of other classes, uses far more attributes of other classes than its own and the attributes used belong to very few other classes.

We consider that a class (client) makes use of a class which reveal the God Class, Brain Class and Feature Envy design flaws if the client calls <sup>6</sup> at least one method from a flawed class. We consider a client making use of a Data Class if the client accesses at least one attribute for such a class or calls at least one of the exposed methods from the class.

The number of distinct flawed classes used by the current class (*noUsedFlawedClasses*) is the distinct number of flawed classes (Data Classes, God Classes, Brain Classes and Feature

<sup>5</sup><http://www.r-project.org>.

<sup>6</sup>polymorphic calls are excluded.

Envy) used. Each of counted God Classes, Brain Classes and Feature Envy provides at least one service (*i.e.*, the methods of the current class call at least one method for each counted class) while each of the counted Data Class provides at least one service or one value (*i.e.*, the methods of the class access an attribute).

### B. Extracting defects.

The defects were extracted from the CVS version archive of Eclipse and Bugzilla bug tracking system using the approach from [23]. The authors of the approach provides the Eclipse bug data set freely available, and we use the number of pre-release and post-release defects (number of defects reported six months before and after release) provided by this data set.

We integrate from the Eclipse bug data set the numbers of pre-release and post-release defects into the last two columns of the file whose structure is described in Figure 1. The correspondence between the data extracted by IPLASMA and the data related to defects is established based on the value of the *fileName* column. We take into account only the entities whose type is 0 or 1 (*i.e.*, interfaces were excluded from this study).

## IV. CHARACTERISTICS OF THE EMPIRICAL STUDY

The *goal* of this case study is to make an examination related to the classes that make use of classes affected by design flaws and defect proneness of classes.

The *quality focus* is the defect proneness of classes making use of entities (*e.g.*, methods, classes) revealing design flaws.

The *perspective* of this case study is that both researchers and developers gain knowledge about the correlation between the clients of classes affected by design flaws with respect to the defect proneness of the clients. This correlation may also help the developers in order to support decision making in software engineering.

The *context* is to analyze three versions of Eclipse.

### A. Research Questions

- Are the clients of the classes which reveal design flaws more defect prone than the clients of the classes which do not reveal design flaws?
- Is the number of defects classes that make use of only one flawed class significantly different than the number of defects classes that make use of more than one flawed class?

### B. Employed Statistical Tests

In order to answer to the mentioned research questions we employ the following statistical tests: Chi-Square, Odds Ratio and Man-Whitney U Test. Next, we briefly present them.

The Chi-Square Test, as it is presented in [30], evaluates if within the underlying population represented by the sample in a contingency table ( $rxn$ ), the observed cell frequencies are different from the expected frequencies. We employ the Chi-Square Test for independence applied upon six samples

(for each version of Eclipse we extract two samples, one regarding pre-release defects and one regarding post-release defects) categorized on two dimensions. The tested hypothesis is that the two involved dimensions are independent of one another (*i.e.*, no correlations among them is found). The assumptions one has to check before running the test is that the data fit into the nominal/categorical level of measurement, each subject is only once represented in the sample and the expected frequency into the contingency table is no less than 5.

Odds indicate how much likely is for an event to occur as opposed to not occur. The likelihood for an event to occur (*e.g.*, odds) is computed by dividing the probability that the event will occur by the probability that the event will not occur. The Odds Ratio for a contingency table is the ratio of the two odds characterizing the table. The Odds Ratio is the ratio of the odds of an event occurring in one group to the odds of it occurring in another group. If the value of the Odds Ratio is 1 it means that the probability for an event is equal to occur into both of the groups. If we encounter a value greater than 1 for the Odds Ratio, it means that the probability that an event will occur in the first group is higher than it occurring in the second group. If the value is less than 1, it means that the probability that an event will occur in the first group is lower than it occurring in the second group.

The hypothesis evaluated with the Man-Whitney U Test is if two independent samples represent two populations with different median values [30]. The assumptions of this test is that data come from two independent sample, each sample containing more than 10 values. As Chi-Square Test, Man-Whitney U Test is a nonparametric test and consequently data does not require a normal distribution or any other particular one.

### C. Conducting the Empirical Study

In this paragraph we present the structure of the two-dimensions contingency table used for computing the value for the Chi-Square test.

The first dimension which we call *RevealDesignFlaws*, *UseFlawedClasses* is comprised of four mutual categories, while the second dimension (*RevealDefects*) is comprised of two mutual categories. The *RevealDesignFlaws*, *UseFlawedClasses* dimension is the row dimension (*e.g.*, independent) and the *RevealDefects* is the column dimension (*e.g.*, dependent). Thus, our contingency table is a 4x2 table. The four categories which composed the *RevealDesignFlaws*, *UseFlawedClasses* dimension are:

- *Do not Reveal Design Flaws, Do not Use Flawed Classes* - a class is assigned to this category if it does not reveal the Data Class, GodClass, Brain Class and FeatureEnvy design flaws, neither make use of a class which reveals the above design flaws. As previously mentioned, we consider that a class (*i.e.*, client) make use of a class which reveals the presented design flaws if it calls at least

	Reveal Pre-Release Defects	Do not Reveal Pre-Release Defects	Row Sums		Reveal Post-Release Defects	Do not Reveal Post-Release Defects	Row Sums
Do not Reveal Design Flaws Do not Use Flawed Classes	765	1540	2305	Do not Reveal Design Flaws Do not Use Flawed Classes	189	2116	2305
Do not Reveal Design Flaws Use Flawed Classes	974	1043	2017	Do not Reveal Design Flaws Use Flawed Classes	409	1608	2017
Reveal Design Flaws Do not Use Flawed Classes	77	293	370	Reveal Design Flaws Do not Use Flawed Classes	16	354	370
Reveal Design Flaws Use Flawed Classes	367	189	556	Reveal Design Flaws Use Flawed Classes	231	325	556
Column Sums	2183	3065	5248	Column Sums	845	4403	5248

(a) Contingency Table for Pre-Release Defects.

(b) Contingency Table for Post-Release Defects.

	Reveal Pre-Release Defects	Do not Reveal Pre-Release Defects
Do not Reveal Design Flaws Do not Use Flawed Classes	958.80	1346.19
Do not Reveal Design Flaws Use Flawed Classes	839.00	1177.99
Reveal Design Flaws Do not Use Flawed Classes	153.90	216.09
Reveal Design Flaws Use Flawed Classes	231.27	324.72

(c) Expected Values for Pre-Release Defects.

	Reveal Post-Release Defects	Do not Reveal Post-Release Defects
Do not Reveal Design Flaws Do not Use Flawed Classes	371.13	1933.86
Do not Reveal Design Flaws Use Flawed Classes	324.76	1692.23
Reveal Design Flaws Do not Use Flawed Classes	59.57	310.42
Reveal Design Flaws Use Flawed Classes	89.52	466.47

(d) Expected Values for Post-Release Defects.

Fig. 2: Eclipse 2.0 - Contingency Tables and Expected Values.

one method from a flawed class. Additionally to this rule, we consider that a class makes use of a Data Class if it accesses at least one attribute for such a class.

- *Do not Reveal Design Flaws, Use Flawed Classes* - a class belongs to this category if it is not detected as being a Data Class, GodClass, Brain Class or FeatureEnvy but make use of at least one class revealing at least a design flaw.
- *Reveal Design Flaws, Do not Use Flawed Classes* - a class is mapped into this group if it is identified as exhibiting at least one design flaw, but do not make use of classes affected by design flaws.
- *Reveal Design Flaws, Use Flawed Classes* - a class is assigned into the last category if it reveals at least one design flaw and use at least one class revealing at least a design flaw.

The two categories which composed the *RevealDefects* dimension are:

- *Reveal Defects* - if a class reveal at least a defect, it is mapped into this category.
- *Do not Reveal Defects* - if a class does not reveal a defect, we consider it as belong into this category.

Since we investigate both pre-release and post-release defects, we create two contingency tables for each analyzed versions of Eclipse (2.0, 2.1 and 3.0). For each version the first contingency table (Figure 2a, Figure 3a, and Figure 4a) stores the observations related to the pre-release defects, while the second contingency table (Figure 2b, Figure 3b, and Figure 4b) stores the observations related to the post-release defects.

Each value encountered in each contingency table represents the number of classes we observe as falling into the corresponding categories. For example, Figure 2a shows that Eclipse 2.0 has 765 classes that Do not Reveal Design Flaws, Do not Use Flawed Classes and Reveal Pre-Defects (the upper left value associated to the Cell  $O_{11}$ ), 1540 classes that Do not Reveal Design Flaws, Do not Use Flawed Classes and Do not Reveal Pre-Defects (the upper right value associated to the Cell  $O_{12}$ ), 367 classes that Reveal Design Flaws, Use Flawed Classes and Reveal Pre-Defects (the bottom left value associated to the Cell  $O_{41}$ ). The mentioned figure also shows the Column Sums (e.g., 2183 ( $O_{.1}$ ) classes revealing Pre-Defects) and Rows Sums (e.g., 2305 ( $O_{1.}$ ) classes that Do not Reveal Design Flaws, Do not Use Flawed Classes).

From the contingency tables we can find out that:

- the number of classes which reveal pre-release defects is substantially higher that the classes revealing post-release defects.
- each version of Eclipse contains around 20% of classes revealing design flaws.
- a percent varying from 33 to 53% of classes make use of at least one flawed class.

#### D. Results of the Empirical Study

We consider Chi-Square test of independence as being the most appropriate test to employ. The hypothesis that is evaluated is related to the independence of the two dimensions from the contingency table. Next, we present the Null Hypothesis, respectively the Alternative Hypothesis for the Chi-Square test.

Null Hypothesis:  $H_0 : o_{ij} = \epsilon_{ij}$ , where  $o_{ij}$  represents the observed frequency of  $Cell_{ij}$ ,  $\epsilon_{ij}$  represents the expected frequency of  $Cell_{ij}$  in the underlying population of classes.

	Reveal Pre-Release Defects	Do not Reveal Pre-Release Defects	Row Sums		Reveal Post-Release Defects	Do not Reveal Post-Release Defects	Row Sums
Do not Reveal Design Flaws Do not Use Flawed Classes	416	2038	2454	Do not Reveal Design Flaws Do not Use Flawed Classes	169	2285	2454
Do not Reveal Design Flaws Use Flawed Classes	952	1697	2649	Do not Reveal Design Flaws Use Flawed Classes	330	2319	2649
Reveal Design Flaws Do not Use Flawed Classes	39	386	425	Reveal Design Flaws Do not Use Flawed Classes	19	406	425
Reveal Design Flaws Use Flawed Classes	433	289	722	Reveal Design Flaws Use Flawed Classes	202	520	722
Column Sums	1840	4410	6250	Column Sums	720	5530	6250

(a) Contingency Table for Pre-Release Defects.

(b) Contingency Table for Post-Release Defects.

	Reveal Pre-Release Defects	Do not Reveal Pre-Release Defects
Do not Reveal Design Flaws Do not Use Flawed Classes	722.45	1731.54
Do not Reveal Design Flaws Use Flawed Classes	779.86	1869.13
Reveal Design Flaws Do not Use Flawed Classes	125.12	299.88
Reveal Design Flaws Use Flawed Classes	212.55	509.44

(c) Expected Values for Pre-Release Defects.

	Reveal Post-Release Defects	Do not Reveal Post-Release Defects
Do not Reveal Design Flaws Do not Use Flawed Classes	282.70	2171.29
Do not Reveal Design Flaws Use Flawed Classes	305.16	2343.83
Reveal Design Flaws Do not Use Flawed Classes	48.96	376.04
Reveal Design Flaws Use Flawed Classes	83.17	638.82

(d) Expected Values for Post-Release Defects.

Fig. 3: Eclipse 2.1 - Contingency Tables and Expected Values.

	X-square	p-value
Eclipse 2.0 Pre-Release Defects	306.44	< 2.2e-16
Eclipse 2.0 Post-Release Defects	437.05	< 2.2e-16
Eclipse 2.1 Pre-Release Defects	646.09	< 2.2e-16
Eclipse 2.1 Post-Release Defects	266.54	< 2.2e-16
Eclipse 3.0 Pre-Release Defects	785.02	< 2.2e-16
Eclipse 3.0 Post-Release Defects	553.40	< 2.2e-16

Fig. 5:  $\chi^2$  and p-value for the analyzed systems.

Considering the sample, it means that the observed frequency is equal to the expected frequency for each cell. For the contingency table revealing pre-release defects for Eclipse 2.0 (Figure 2a) the expected values are presented in Figure 2c. Each expected value from Figure 2c was computed according to the formula  $E_{ij} = \frac{(O_{i.})(O_{.j})}{n}$ , where  $(O_{i.})$  represents the sum of observations in the row where the cell appears, while  $(O_{.j})$  represents the sum of observations in the column where the cell appears. The reason behind the above formula comes from the consideration that the two dimensions of the contingency table are independent.

Alternative Hypothesis:  $H_1 : o_{ij} \neq \epsilon_{ij}$ . This formula states that in the underlying population of classes the sample represents, the observed frequency for at least one cell is different than the expected frequency. With respect to the sample it means that the observed frequency is not equal to the expected frequency for at least one cell.

For each of the contingency tables from Figures we compute the values of the Chi-Square test  $\chi^2$  using the R Project for Statistical Computing [28]. In Figure 5 we show the obtained

results for  $\chi^2$  as well as the p-values. Since all p-values are less than a 0.05 level of significance ( $\alpha=0.05$ ), we consider we have enough evidence to reject the null Hypothesis. Consequently, the two dimensions of the contingency table are not independent (*i.e.*, some correlations exist among them).

Next, based on the values of the observed and expected frequencies, we establish the way (positive or negative) in which the involved dimensions are correlated. A trait from the row dimension is positively correlated with a trait from the column dimension if the observed frequency is greater than the expected frequency. Based on this, we discuss below each type of the existing inferred correlation.

*Pre-Release Defects.* Each version of Eclipse reveals a negative correlation between classes that Do not Reveal Design Flaws, Do not Use Flawed Classes and Reveal Pre-Release Defects ( $765 < 958.80$ ,  $416 < 722.45$ ,  $584 < 968.38$ ). Thus, classes having this trait were less likely to exhibit this type of defects. Classes that Do not Reveal Design Flaws, Use Flawed Classes were more likely to Reveal Pre-Release Defects ( $974 > 839$ ,  $952 > 779.86$ ,  $1321 > 1128.22$ ). Classes belonging to the third trait (Reveal Design Flaws, Do not Use Flawed Classes) were less likely to Reveal Pre-Release Defects ( $77 < 153.90$ ,  $39 < 125.12$ ,  $49 < 153.53$ ). Thus, in isolation, a flawed class is not positively correlated with the Pre-Release Defects! In any case, besides the negative impact they have upon the maintenance, we find out that the clients of flawed classes are likely to Reveal Pre-Release Defects. Classes that fall into the last category (Reveal Design Flaws, Use Flawed Classes) are positively correlated with the Pre-Release Defects ( $367 > 231.27$ ,  $433 > 212.55$ ,  $596 > 299.85$ ), meaning that they are more likely to exhibit defects.

*Post-Release Defects.* Comparing the observed values from

	Reveal Pre-Release Defects	Do not Reveal Pre-Release Defects	Row Sums		Reveal Post-Release Defects	Do not Reveal Post-Release Defects	Row Sums
Do not Reveal Design Flaws Do not Use Flawed Classes	584	2639	3223	Do not Reveal Design Flaws Do not Use Flawed Classes	251	2972	3223
Do not Reveal Design Flaws Use Flawed Classes	1321	2434	3755	Do not Reveal Design Flaws Use Flawed Classes	712	3043	3755
Reveal Design Flaws Do not Use Flawed Classes	49	462	511	Reveal Design Flaws Do not Use Flawed Classes	27	484	511
Reveal Design Flaws Use Flawed Classes	596	402	998	Reveal Design Flaws Use Flawed Classes	368	630	998
Column Sums	2550	5937	8487	Column Sums	1358	7129	8487

(a) Contingency Table for Pre-Release Defects.

(b) Contingency Table for Post-Release Defects.

	Reveal Pre-Release Defects	Do not Reveal Pre-Release Defects
Do not Reveal Design Flaws Do not Use Flawed Classes	968.38	2254.61
Do not Reveal Design Flaws Use Flawed Classes	1128.22	2626.77
Reveal Design Flaws Do not Use Flawed Classes	153.53	357.46
Reveal Design Flaws Use Flawed Classes	299.85	698.14

(c) Expected Values for Pre-Release Defects.

	Reveal Post-Release Defects	Do not Reveal Post-Release Defects
Do not Reveal Design Flaws Do not Use Flawed Classes	515.71	2707.28
Do not Reveal Design Flaws Use Flawed Classes	600.83	3154.16
Reveal Design Flaws Do not Use Flawed Classes	81.76	429.23
Reveal Design Flaws Use Flawed Classes	159.68	838.31

(d) Expected Values for Post-Release Defects.

Fig. 4: Eclipse 3.0 - Contingency Tables and Expected Values.

	Defects	No Defects
Do not Reveal Design Flaws Do not Use Flawed Classes	-	+
Do not Reveal Design Flaws Use Flawed Classes	+	-
Reveal Design Flaws Do not Use Flawed Classes	-	+
Reveal Design Flaws Use Flawed Classes	+	-

Fig. 6: Types of the inferred correlations.

the contingency table to the expected values, we establish the same types of correlations as within the Pre-Release Defects. We summarize these correlations in Figure 6, which holds for both kinds of defects.

*Conclusions.* Our findings shows that if a class makes use of a class that reveals design flaws, then it is more likely to exhibit defects. Thus, when a developer is aware of a class revealing design flaws within the system, it should be also aware that the clients of the class are more likely to exhibit defects. We consider it important to mention that the provided results do not allow us to draw the conclusion that classes exhibiting design flaws are the cause of encountering more defects among their clients; we only provided evidence about an existing positive correlation.

In order to bring a stronger evidence related to the correlation between clients of classes revealing design flaws and defects, we calculate the Odds Ratio for the last three rows category, with respect to the first row category.

The computations of the Odds Ratio firstly require the

computation of the Odds. For example, the odds that a class revel pre-release defects in the first condition (Do not Reveal Design Flaws, Do not Use Flawed Classes) is computing by dividing 765 to 1540. In Figure 7 we present the values of the Odds for each contingency table associated with the three analyzed versions of Eclipse. For example, from this figure we can find out that the Odds of Revealing Pre-Release Defects for classes that Do not Reveal Design Flaws, Do not Use Flawed Classes are 0.4967 for Eclipse 2.0, 0.2041 for Eclipse 2.1 and 0.2213 for Eclipse 3.0 and the Odds of Revealing Post-Release Defects for the same type of classes are 0.0893 for Eclipse 2.0, 0.0739 for Eclipse 2.1 and 0.0844 for Eclipse 3.0.

From the mentioned figure (Figure 7) we can notice that:

- the probability for a class in the last row category (Reveal Design Flaws, Use Flawed Classes) to reveal Pre-Release Defects is greater than one half (the value of the Odds are greater than 1 (1.9417 for Eclipse 2.0, 1.4982 for Eclipse 2.1 and 1.4825 for Eclipse 3.0)).
- the probability for a class in all of the remaining conditions to reveal defects is lower than one half (all of the Odds values are lower than 1).

The positive correlation between classes falling into the Use Flawed Classes category and Reveal Defects category is also emphasized by highest values of the *Odds*.

The Odds Ratio is computed by dividing two values of the Odds computing for a contingency table. Since from the four categories existing within the rows of the contingency table, it is advisable for practitioners to develop classes targeting the first category (Do not Reveal Design Flaws, Do not Use Flawed Classes), we compute the Odds Ratio for the last three category with respect to the first category (see Figure 8). The Odds Ratio for a class in the second condition (Do

	ODDS Eclipse2.0	ODDS Eclipse2.1	ODDS Eclipse3.0
	Pre-Release Defects ----- Post-Release Defects	Pre-Release Defects ----- Post-Release Defects	Pre-Release Defects ----- Post-Release Defects
Do not Reveal Design Flaws Do not Use Flawed Classes	0.4967 ----- 0.0893	0.2041 ----- 0.0739	0.2213 ----- 0.0844
Do not Reveal Design Flaws Use Flawed Classes	0.9338 ----- 0.2543	0.5609 ----- 0.1423	0.5427 ----- 0.2339
Reveal Design Flaws Do not Use Flawed Classes	0.2627 ----- 0.0451	0.1010 ----- 0.0467	0.1060 ----- 0.0557
Reveal Design Flaws Use Flawed Classes	1.9417 ----- 0.7107	1.4982 ----- 0.3884	1.4825 ----- 0.5841

Fig. 7: The Values of the Odds.

	ODDS RATIO Eclipse2.0	ODDS RATIO Eclipse2.1	ODDS RATIO Eclipse3.0
	Pre-Release Defects ----- Post-Release Defects	Pre-Release Defects ----- Post-Release Defects	Pre-Release Defects ----- Post-Release Defects
Do not Reveal Design Flaws Use Flawed Classes	1.8800 ----- 2.8477	2.7481 ----- 1.9255	2.4523 ----- 2.7713
Reveal Design Flaws Do not Use Flawed Classes	0.5288 ----- 0.5050	0.4948 ----- 0.6319	0.4789 ----- 0.6599
Reveal Design Flaws Use Flawed Classes	3.9092 ----- 7.9585	7.3405 ----- 5.2557	6.6990 ----- 6.9206

Fig. 8: The Values of the Odds-Ratio.

not Reveal Design Flaws, Use Flawed Classes) revealing pre-release defects versus a class in the first condition (Do not Reveal Design Flaws, Do not Use Flawed Classes) is 1.88 (0.9338/0.4967) for Eclipse 2.0, 2.7481 for Eclipse 2.1 and 2.4523 for Eclipse 3.0. The Odds Ratio for a class in the same condition revealing post-release defects is 2.8477 for Eclipse 2.0, 1.9255 for Eclipse 2.1 and 2.7713 for Eclipse 3.0. These results show us that:

- the chances for a pre-release defect to occur in the group of classes that Do not Reveal Design Flaws, Use Flawed Classes are 1.88 to 2.74 times higher than the chances for a pre-defect to occur in the group of classes that Do not Reveal Design Flaws, Do not use Flawed Classes.
- the chances for a post-release defect to occur in the group of classes that Do not Reveal Design Flaws, Use Flawed Classes are 1.92 to 2.84 times higher than the chances for a post-defect to occur in the group of classes that Do not Reveal Design Flaws, Do not use Flawed Classes.
- the chances for pre and post release defects to occur in the third group of classes are smaller than the chances they have to occur in the first group.
- the values of Odds Ratio for classes falling into the last category (Reveal Design Flaws, Use Flawed Classes) reveal higher chances for defects to occur in this group compared to the first group (Do not Reveal Design Flaws, Do not use Flawed Classes) – from 3.90 to 7.34 higher

chances for pre-release defects and 5.25 to 7.95 for post-release defects.

*Conclusions.* The values of the Odds Ratio indicate greater chances for the clients of flawed classes to exhibit defects than for the clients of classes that do not reveal design flaws. We also find out the chances for a class to exhibit post-release defects to be higher than the chances to exhibit pre-release defects if the class make use of flawed classes.

In order to check if classes (*i.e.*, clients) which make use of more than one flawed class reveal a significant different number of defects than classes which make use of only one flawed class, we divide the classes from the second condition (Do not Reveal Design Flaws, Use Flawed Classes) and the last condition (Reveal Design Flaws, Use Flawed Classes) into two groups. Since we make a distinction between pre and post-release defects, for each version of Eclipse (2.0, 2.1 and 3.0) we establish eight groups. The first group of each mentioned condition contains the classes which make use of only one flawed class, while the second contains the classes which make use of more than one flawed class.

Null Hypothesis:  $H_0 : \theta_1 = \theta_2$ , where  $\theta_1$  represents the median of the population the first group represents, while  $\theta_2$  represents the median of the population the second group represents. With respect to the sample data, the null hypothesis states that the means of the ranks of the two samples are equal.

Alternative Hypothesis:  $H_1 : \theta_1 \neq \theta_2$ . This formula states that in the underlying population of classes the sample represents, the median of the population from the first group is not equal with the median of the population from the second group. With respect to the sample data, the alternative hypothesis states that the means of the ranks of the two groups are not equal.

For each of the mentioned groups, we run in the R environment the Mann-Whitney U Test both for the pre-release and post-release defects. The p-values of the test are presented in Figure 9. Since most of the p-values are greater than 0.05 we consider that the samples we have are not strong enough to warrant rejection of the Null Hypothesis. Consequently, until a stronger evidence is brought, we assume that the population of classes making use of only one class has an equal median both for pre-release defects and post-release defects with the population making use of more flawed classes.

*Conclusions.* The data we have suggest that the number of pre-release defects and post-release defects are about the same for classes which use only one flawed, as well as for classes which use more than one flawed class. Consequently the use of a single flawed class positively correlates with increased chances for the classes which Do not Reveal Design Flaws, as well as for the classes which Reveal Design Flaws to exhibit defects (pre-release and post-release).

## V. THREATS TO VALIDITY

In this section we present the threats to validity associated to our empirical study, following the guidelines from [31] [32].

	p-value	Do not Reveal Design Flaws Use Flawed Classes	Reveal Design Flaws Use Flawed Classes
Eclipse 2.0	Pre-Release Defects	0.0522	0.9935
Eclipse 2.0	Post-Release Defects	0.1938	0.4327
Eclipse 2.1	Pre-Release Defects	< 0.0500	0.3504
Eclipse 2.1	Post-Release Defects	< 0.0500	0.1853
Eclipse 3.0	Pre-Release Defects	< 0.0500	0.5710
Eclipse 3.0	Post-Release Defects	< 0.0500	0.7805

Fig. 9: p-values for the Mann-Whitney U Test.

*Construct validity.* This type of threats are connected to the extent the operational measures for the concepts being studied were established correctly [31]. Within the case study presented in this paper they are mainly related to the errors performed during the data extraction. The possible errors are due to the extraction of:

- design entities from the source code. We consider they were extracted with a high degree of precision and recall.
- classes exhibiting design flaws. The approach according to they are extracted is not new and was evaluated in various previous case studies and, consequently, well known accepted approach in the field.
- defects from the CVS version repository and Bugzilla bug tracking system. We use the free Eclipse bug data set containing the defects involved in the study presented in [23]. Consequently, all the encountered threats to the construct validity we can find in the mentioned study are also found in our study.

We use only non parametric statistical tests and all the assumptions required by the used tests were satisfied.

*Internal validity.* This aspect of validity is related to the causal relations that are inferred. Since our study is an exploratory one, this aspect is not relevant.

*External validity.* The reported results are obtained by analyzing three versions of the well known Eclipse open source software system. This system is the subject of many empirical studies (e.g., [10] [23] [25]) and, like the previous studies, we do not suggest generalizing our research results to other systems unless further case studies are performed.

*Reliability validity.* This aspect concerns the fact that a later investigator that conduct the same case study like the one presented here should obtain the same results and, consequently, reach the same conclusions. We consider we provided enough information about the conducted study in order to let it be replicated. The source code of Eclipse is freely available, as well as the number of bugs it exhibits. The detection strategies according to classes revealing design flaws are extracted are presented in [7] and the all the steps performed are presented in this paper.

## VI. CONCLUSIONS. FUTURE WORK

In this paper we present an empirical study performed upon three releases of Eclipse that provides evidence about a

positive correlation between the clients of flawed classes and the defects the clients exhibit. We show that, taken in isolation, classes exhibiting the identity disharmonies design flaws (e.g., Data Class, God Class, Brain Class, Feature Envy) do not have an increased likelihood to exhibit defects that classes which do not reveal design flaws. When those classes are used by their clients the likelihood for the clients to exhibit defects greatly increases, especially for the post-release defects.

We do not want to suggest that using a flawed class is the cause of an increased likelihood to exhibit defects. However, we do provide evidence that the usage of flawed classes is statistically correlated with defects. We consider that the practitioners should be aware of this inferred correlation.

We will focus our future work on the next fronts:

- we intend to replicate this study against other systems in order to see if the results obtained in this study can be generalized.
- in this study we consider only the identity disharmonies design flaws. We aim to address in future work more design flaws (e.g., collaboration and classification disharmonies).
- when we established the clients of a class, we did not take into account the polymorphic calls. We considered an inspected class to be a client of an explicit class which provided services if the client calls at least a method (service) from the explicit class. Using the metric-based approach which captures the extend to which the clients of a hierarchy polymorphically manipulate the hierarch from [21] and using fuzzy rules, we can further refine the rules according to a class is considered to be used by another class.

## ACKNOWLEDGMENTS

We would like to thank Thomas Zimmermann, Rahul Premraj and Andreas Zeller for making available the Eclipse bug data referred in [23].

We also owe a lot to Ioana Verebi and George Ganea for contributing, over the years, to the high degree of scalability IPLASMA reveals now.

## REFERENCES

- [1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [2] A. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
- [3] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006.
- [4] R. C. Martin, *Clean Code. A Handbook of Agile Software Craftsmanship*. PrenticeHall, 2008.
- [5] K. Henney(editor), *97 Things Every Programmer Should Know*. O'Reilly, 2010.
- [6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [7] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer Verlag, 2006.
- [8] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 65, 2003.

- [9] F. Khomh, M. D. Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *16th Working Conference on Reverse Engineering*, 2009.
- [10] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, July 2007.
- [11] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *IEEE International Conference on Software Maintenance*, 2010.
- [12] G. Booch, R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, and K. A. Houston, *Object-Oriented Analysis and Design with Applications*. Addison Wesley, 2007.
- [13] E. van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," in *WCRE 2002 proceedings*, 2002.
- [14] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in Java source-code," in *IEEE International Symposium on Software Metrics*, 2005.
- [15] C. Marinescu, R. Marinescu, P. Mihancea, D. Rațiu, and R. Wettel, "iPlasma: An integrated platform for quality assessment of object-oriented design." in *Proc. IEEE International Conference on Software Maintenance (ICSM Industrial and Tool Volume)*, Budapest, Hungary. IEEE Computer Society Press, 2005.
- [16] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, 2010.
- [17] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an extensible language-independent environment for reengineering object-oriented systems." in *Proc. International Symposium on Constructing Software Engineering Tools*, 2000.
- [18] R. Marinescu, G. Ganea, and I. Verebi, "inCode: Continuous quality assessment and improvement," in *14th European Conference on Software Maintenance and Reengineering(CSMR)*, 2010.
- [19] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *9th International Conference on Quality Software*, 2009.
- [20] C. Marinescu, "Identification of design roles for the assessment of design quality in enterprise applications." in *Proc. IEEE International Conference on Program Comprehension (ICPC)*, Athens, Greece. IEEE Computer Society Press, 2006.
- [21] P. F. Mihancea, "Type highlighting: A client-driven visual approach for class hierarchies reengineering," in *8th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008.
- [22] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller, "If your bug database could talk," in *In Proceedings of the 5th International Symposium on Empirical Software Engineering, Volume II: Short Papers and Posters*, 2006, pp. 18–20.
- [23] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering*. IEEE Computer Society, 2007.
- [24] A. Oram and G. Wilson(editors), *Making Software. What Really Works, and Why We Believe It*. O'Reilly, 2010.
- [25] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, 2010.
- [26] D. Rațiu, *Memoria: A Unified Meta-Model for Java and C++*. Master Thesis, "Politehnica" University of Timișoara, 2004.
- [27] C. Team, *Recoder Project*, <http://recoder.sourceforge.net/>. University of Karlsruhe.
- [28] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Web page: <http://findbugs.sourceforge.net>, 2010, ISBN 3-900051-07-0. [Online]. Available: <http://www.R-project.org>
- [29] C. Marinescu and I. Jurca, "A meta-model for enterprise applications." in *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE Computer Society Press, 2006.
- [30] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, 4th edition*. Chapman&Hall/CRC, 2007.
- [31] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, 2009.
- [32] R. K. Yin, *Case Study Research: Design and Methods., 3rd edition*. SAGE Publications, 2002.