On the Impact of Design Flaws on Software Defects

Marco D'Ambros, Alberto Bacchelli, Michele Lanza
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland
{marco.dambros, alberto.bacchelli, michele.lanza}@usi.ch

***Abstract*—The presence of design flaws in a software system has a negative impact on the quality of the software, as they indicate violations of design practices and principles, which make a software system harder to understand, maintain, and evolve. Software defects are tangible effects of poor software quality.**

In this paper we study the relationship between software defects and a number of design flaws. We found that, while some design flaws are more frequent, none of them can be considered more harmful with respect to software defects. We also analyzed the correlation between the introduction of new flaws and the generation of defects.

***Index Terms*—Software quality and design; Software defects**

I. Introduction

In the light of the increased complexity of today's software systems, it is no wonder that maintenance and evolution claim 90% of the total software costs [1]. In this context, much effort has been devoted to find approaches capable of detecting parts of the source code that are likely to be harder to maintain, or to be more related to defects. Source code entities that have design flaws are good candidates, since these are known to have a negative impact on quality attributes such as flexibility or maintainability [2]. However, simple source code metrics are not capable of identifying poorly designed parts, because they must be analyzed and considered in the context in which they appear. For this reason, *meaningful* metric combinations have been devised as so-called *detection strategies* [3] and put in the context of *design (dis)harmony*, [4].

Design disharmonies have been thoroughly analyzed in literature: to find good metrics and thresholds for their classification [3]–[5], to propose correction strategies and refactorings [4], [6], to visualize them [7], and to put them in relation to code evolvability [8] or change-proneness [9]. However, a research, which was never carried out so far, is the analysis of the relationship between design flaws and *software defects*.

In this paper, we investigate this relationship conducting an extensive experiment on six open-source software systems. We analyze the frequency of design flaws in software systems. Then, we analyze the correlation of flaws with post-release defects. Finally, we evaluate whether an addition in the number of flaws in a system can induce bugs. We conduct our experiment not only analyzing each flaw, per se, but also extracting and comparing differences between flaws in all these situations.

We choose to analyze design flaws in six different software systems from two open-source communities (i.e., Apache and Eclipse) because the development environment, the community culture, and the development paradigms are all likely to be different. Our goal is to compare whether and how these characteristics induct, influence, or alleviate different flaws.

Although our goal is clearly defined, we do not expect to be able to fully determine it with six case studies, even though we consider the comprehensive data from all the histories of the software systems analyzed. However, we do expect to provide some useful evidence that can contribute to the analysis of the relationship between different design flaws, and between design flaws and software defects, and also, help delineate appropriate questions to ask in future case studies.

Structure of the paper In Section II we introduce *detection strategy*: the technique we employ to identify design flaws in software systems. In Section III we explain how we extract, link, and process the actual data from source code and bug repositories, that we later use in Section IV to conduct the core of the experiment. In Section V we outline the threats to the validity of this study. In Section VI we analyze the current research on detecting design flaws and in the field of defect analysis and prediction. We conclude in Section VII.

II. Design Flaws and Detection Strategies

As opposed to object-oriented metrics [10], which are simple measures of size (e.g., lines of code, number of methods) or complexity (e.g., McCabe cyclomatic complexity) of software, *detection strategies* [3] provide a formal method to identify design flaws in a given source code, also referred to in literature as "code smells" [11].

To recognize a number of design flaws, we transform informal design rules, guidelines, and heuristics [2], [11], [12] into *detection strategies*, i.e., quantifiable expressions of rules by which design fragments that are conforming to these rules can be detected in the source code [3]. In practice, *detection strategies* are logical conditions, based on source code metrics, that detect violations against design guidelines. We use the design flaws described by Lanza and Marinescu [4].

As it goes beyond the scope of this paper to describe each flaw in detail, we limit ourselves to exemplify one of these design flaws, i.e., *Brain Method*.

Example. The Brain Method design flaw refers to a method that tends to centralize the functionality of a class, in the same way a God Class [12] centralizes the functionality of an entire (sub)system. It can be informally described by the following rules: (1) it is excessively large, (2) it has many conditional branches, computed using the McCabe's cyclomatic complexity, (3) it has a deep nesting level, and (4) it uses many[1] variables.

[1]Refers to a number higher than a human can keep in short-term memory [13], i.e., 6 - 9.

These rules can be transformed into the detection strategy depicted in Figure 1.

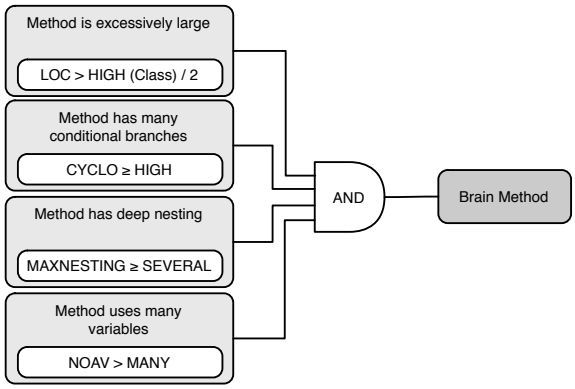

Fig. 1. "Brain Method" Detection Strategy.

Filtering conditions are expressed in terms of metrics (the left part of the expressions) and related to thresholds[2] (the right part of the expressions).

CYCLO, also known as McCabe's Cyclomatic Complexity, is the number of linearly-independent paths through an operation. MAXNESTING represents the maximum nesting level of control structures within an operation. NOAV is the total number of variables directly accessed from the measured operation. Variables include parameters, local variables, instance variables and global variables. "HIGH" refers to a threshold for methods, while "HIGH (Class)" refers to a threshold for classes.

This detection strategy, functioning on any Java software system, produces a set of candidate entities exhibiting symptoms of the *Brain Method* design flaw. For example, in one version of ArgoUML we were able to detect 120 brain methods.

In addition to *Brain Method*, we consider the following method level design flaws:

- *Feature Envy:* methods more interested in the data of other classes than that of their own class [11], by accessing it directly or via accessors.
- *Intensive Coupling:* methods intensively coupled to other methods located in few other classes: The communication between the client method and (at least one of) its provider classes is excessively verbose.
- *Dispersed Coupling* is complementary to the previous design flaw, and it refers to a method excessively tied to many other methods in the system dispersed among many classes: a single method communicates with an excessive number of classes, whereby the communication with each of the classes is not intense.
- *Shotgun Surgery* denotes that a change in a single method implies many changes to many different methods and classes [11]. This design flaw deals with strong afferent (incoming) coupling, thus concerning the coupling strength and dispersion.

[2]Obtained by measuring 45 Java systems. For more details see [4].

III. Obtaining Disharmonies and Bug Data

To analyze the relationship between software bugs and design flaws we first need to extract the necessary data from source code and bug repositories. Figure 2 summarizes our data extraction process.

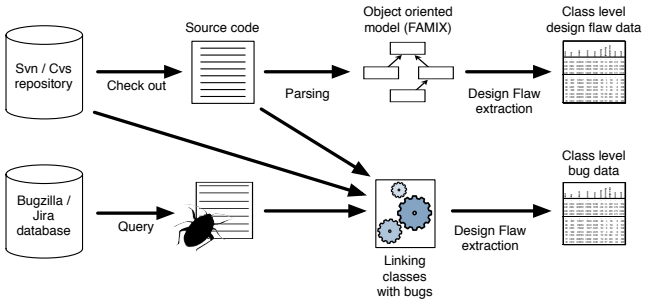

Fig. 2. Extracting Design Flaws and Bug Data.

Extracting Design Flaws. Our starting point, i.e., the input data, is the source code of an object-oriented software system, obtained from a CVS (or Subversion) repository. We parse the source code with the tool inFusion[3] and produce a FAMIX[4]-compliant object-oriented model of the system.

Having the FAMIX-compliant model as input, we use detection strategies to spot the design flaws mentioned previously. This operation is performed within the Moose reengineering framework [15]. The result is a list of method level design flaws that each class contains. We conduct our analysis at class level because classes are the cornerstone of the object-oriented paradigm, and developers perform maintenance and refactoring tasks mostly at this level.

Extracting and Linking Bugs With Classes. Figure 3 shows our approach to link classes with bugs, through versioning system data.

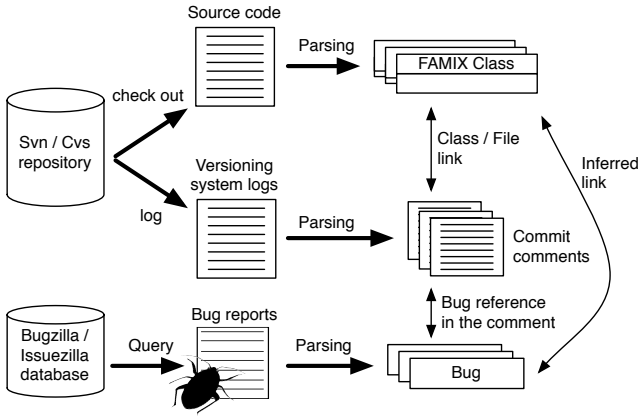

Fig. 3. Linking classes with bugs.

[3]http://www.intooitus.com/inFusion.html

[4]FAMIX is a language independent object-oriented meta-model of software systems [14].

We parse the source code obtaining the FAMIX-compliant model. Then, we extract the list of classes, and we link them with the corresponding files, whose histories (in terms of commits) are included in the log files of the versioning system. Log files contain comments written by developers at commit time. These comments often includes references to problem reports (e.g., "fixed bug 12345"), which allow us to link problem reports with files in the versioning system, and therefore with classes.

The linking between a CVS/SubVersion file and a Bugzilla/JIRA issue report is not formally defined. As widely done in practice [16], [17], we use pattern matching techniques on the developer's comments to find a reference to the issue report id. To choose the appropriate algorithm to detect the links, we analyzed a number of sample comments taken from the case studies. We noted that some bug reference ids are just plain numbers, without keywords such as "fix" or "bug". If looking for the mere number, it is possible to obtain false positives. Thus, in our algorithm, each time we find a candidate reference to a bug report, we not only check that a bug with such an id exists, but we also verify that the date in which this bug was reported is antecedent to the timestamp of the commit comment in which the reference was found (i.e., it verifies that the bug is fixed *after* being reported).

Having the links between classes and files and between files and bugs, we infer the links between classes[5] and bugs.

IV. Experiments

We performed *two* sets of experiments to analyze the relationship between design flaws and software defects:

1) Correlation analysis: We studied the correlation between number of defects and class-level design flaws.
2) Delta analysis: We investigated whether increments in the flaws correlate with defects, within a given time window.

Lucene	A high-performance, full-featured text search engine library.
Maven	A software project management and comprehension tool.
Mina	A network application framework.
Eclipse CDT	A C/C++ Integrated Development Environment (IDE) for the Eclipse platform.
Eclipse PDE UI	Models, builders, editors and more to facilitate plug-in development in the Eclipse IDE.
Equinox	An implementation of the OSGi R4 core framework specification.

TABLE I
SOFTWARE SYSTEMS USED FOR THE EXPERIMENTS.

Data Set. We performed our experiments on the six Java software systems detailed in Table I.

For each system, we considered multiple versions: one version every two weeks in the history of the system. To achieve this, given the Subversion repository of a software project, we performed the following two steps:

1) We checked out the source code relative in the repository at a particular date, iteratively incrementing the date by 14 days.
2) We parsed the source code, extracted design flaws and bug data, as previously described.

For each system, Table II shows the period of time considered[6], the number of versions considered, the size of the systems in terms of average number of classes, the average number of design flaws and the number of references to bugs reported in the considered time period.

Since the number of classes and the number of design flaws vary across system versions, we show average numbers over all the considered versions. The number of classes shown in the table is the total number of classes linked with versioning system files (e.g., not considering inner classes). Bug references indicate all the links to classes that a bug report can have (when a single bug affects multiple classes, it is linked with multiple classes: the number of bugs is one, while the number of bug references is equal to the number of affected classes).

Once we have extracted the design flaw data for all the versions of a system, for each flaw we build a design flaw matrix M with the following properties, exemplified in Figure 4:

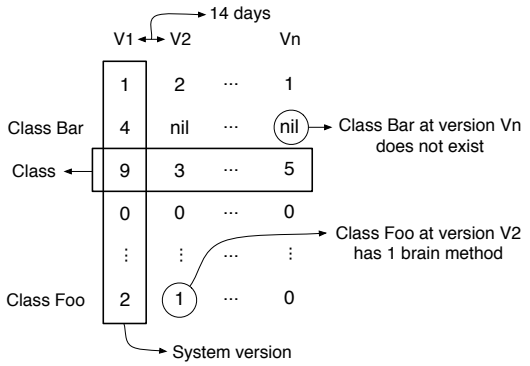

Fig. 4. An example of design flaw matrix for brain methods.

- Each column represents a version of the system.
- Each row represents a class.
- The value of a cell at row r and column c is equal to the number of instances of the design flaw in the class represent by r at the version represent by c.
- Since some classes exist in some versions but not in others, the set of rows is composed by the sum of all the classes existing in at least one version. If a class at row r does not exist at the version in column c, we set the value of the cell c, r to nil.

Having all the pieces of information, the first question that we want to answer is: *Are there design flaws that are more frequent in all the systems, or is each system different with respect to design flaws frequencies?*

[5]Due to the fact that Java inner classes are defined in the same file as their containing class, different classes might point to the same CVS/Subversion file. This implies that a bug linking to a file might actually be linking to more than one class. We are not aware of a fix for this problem, which in fact is a shortcoming of the bug tracking systems employed in the software projects analyzed. For this reason, in our experiments we do *not* consider inner classes.

[6]The end of the time period always corresponds to a release of the software.

Software system	Lucene	Maven	Mina	CDT	PDE UI	Equinox
Time period begin	1 Jan 2005	1 Jan 2005	14 Jan 2006	24 Jun 2006	1 Jan 2005	1 Jan 2005
Time period end	8 Oct 2008	18 Feb 2009	10 Dec 2008	25 Feb 2009	11 Sep 2008	25 Jun 2008
Last release	2.4.0	2.0.10	2.0.0-M4	5.0.2	3.4.1	3.4
Number of versions	99	108	76	70	97	91
Bug refs in the time period	982	1500	629	923	4953	2043
Avg. # classes	513.5	156.2	108.6	217.8	1170.5	242.6
Avg. # Brain method	106.9	24.8	1.6	6.9	29.1	35.5
Avg. # Dispersed coupling	14.2	2.9	1.6	0.4	63.0	31.8
Avg. # Feature envy	943.7	74.7	70.1	90.6	1006.8	450.8
Avg. # Intensive coupling	0.97	6.6	1.3	3.4	1.55	4.3
Avg. # Shotgun surgery	123.7	20.7	19.6	31.5	117.5	36.7

TABLE II
THE DATA SET USED FOR THE EXPERIMENTS.

Table II provides a first indication that there are patterns of design flaws frequencies in the analyzed systems. However, we cannot compare the numbers of flaws in Table II, as systems have different number of classes. To better investigate our question, we compare the average number of disharmonies per class (which is also an average over all the versions).

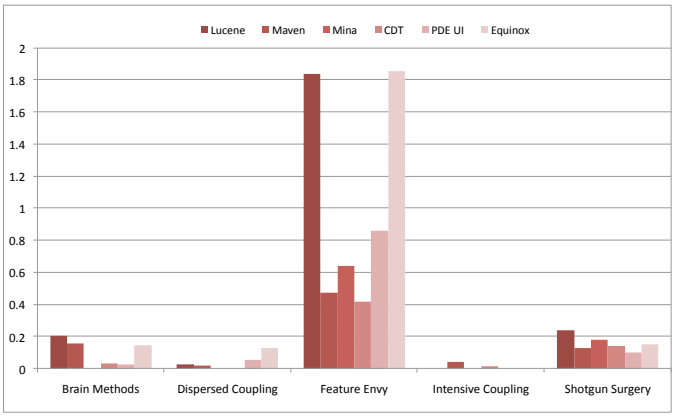

Fig. 5. Average number of design flaws per class, grouped by flaw.

Figure 5 shows the average number of disharmonies per class, grouped by flaw: *Feature envy* is the most frequent in all the systems, followed by *shotgun surgery*, which is stable for all the systems. *Intensive coupling* is relatively low for all systems, while *dispersed coupling* and *brain methods* vary.

A. Correlation Analysis

We found that some design flaws are more frequent than others. Now we want to study the correlation between number of design flaws and number of post release defects at the class level. In particular we aim at answering the following question: *Do design flaws correlate with software defects? Does any flaw consistently correlate more than others in all systems?*

In our analysis we consider post release defects reported after the considered version of the system. For example, if we consider class Foo at version x, post release defects are defects linked to Foo reported after x and within a certain time window. As done by Zimmermann et al. we consider a period of 6 months for post release defects [17]. We consider post-release defects because we want to investigate whether the presence of design flaws generate bugs in the future.

We need two lists to compute the correlation: The first one contains the number of design flaws for each class, and the second one the number of post release defects. The first list corresponds to a column of the matrix M, while the second is mapped to a column in an analogous matrix B. In B rows represent classes, columns represent versions and the value of a cell at position c, r represent the number of post release defects relative to the version c of the class r. To compute the number of post release defects we filter the extracted bug data according to the bugs reporting dates.

To measure the correlation we could use either the Pearson's linear correlation coefficient, which should be used to linear relationships, or the Spearman's rank correlation coefficient, suitable for general associations [18]. To choose which measure is more appropriate we studied the distribution of the data, which resulted to be highly skewed with respect to a normal distribution, in fact, most of the classes have very few or zero design problems and post release defects. For this reason we decided to use the Spearman's coefficient, as it is recommended with data that is skewed or that contains outliers [18].

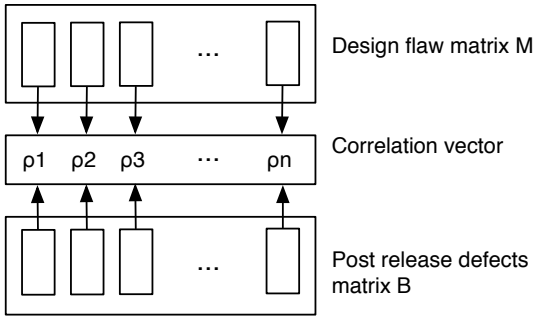

Fig. 6. Computing Spearman's correlations over multiple versions of a system.

Figure 6 shows how we compute the correlation over all the versions of a given system, producing a correlation vector which represents the correlation trend over time. We create the vector by computing the Spearman's coefficient on pairs of columns from the design flaws and bugs matrixes.

Fig. 7. Spearman's correlations between number of design flaws and number of post release defects over multiple versions of software systems.

Figure 7 shows, for each software system in our data set, plots of the correlation vectors for all the considered design flaws. Interruptions in the lines mean that in the corresponding version of the system the Spearman correlation coefficient was not significant. We see that every system is different and there is no flaw which is consistently more correlated with post release defects across the systems. Moreover, not even within systems design flaws are consistently more or less correlated with respect to each other: They oscillate over different versions. To obtain a better grasp at which and how much flaws are correlated with defects, we only consider strong correlation: having a Spearman's coefficient above 0.4, which is the threshold for considering a correlation to be strong in fault prediction studies [19].

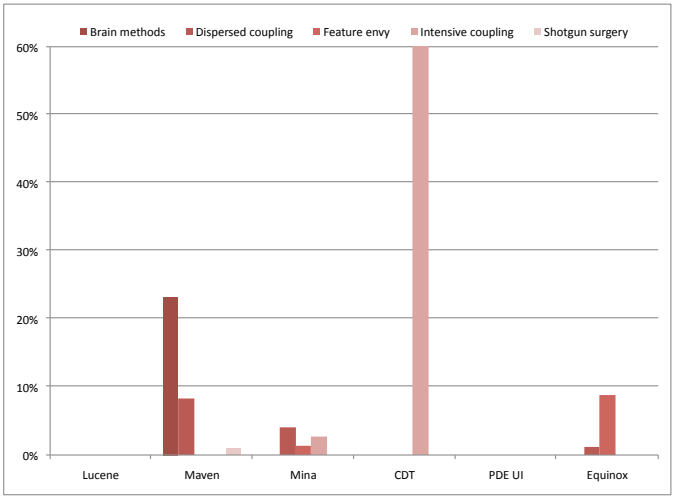

Fig. 8. Percentages of systems versions with a strong correlation (Spearman > 0.4) with post release software defects.

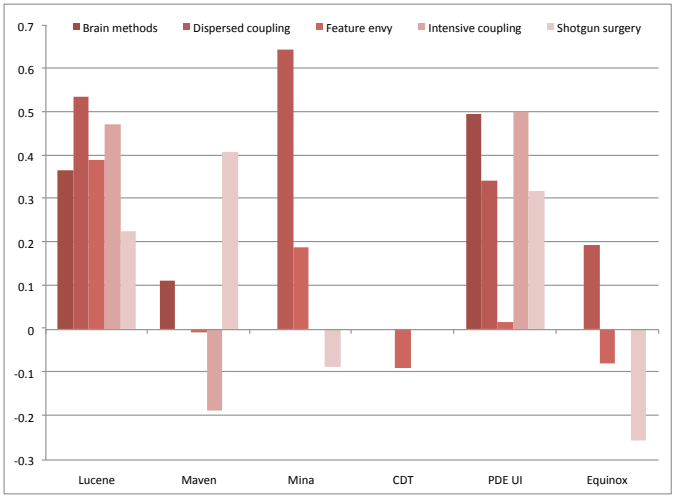

Fig. 9. Spearman's correlations between additions of design flaws and number of generated defects.

Figure 8 shows the percentage of versions with a strong correlation, by design flaw and by software system.

The percentages are computed from the correlation vectors as number of versions with a strong correlation (greater than 0.4) divided by total number of versions. We see that there are two types of systems: (1) systems where no design flaw is strongly correlated with defects (PDE and Lucene) and (2) systems in which one design flaw is frequently strongly correlated with defects (Equinox, CDT, and Maven). Mina does not belong to any group, as some design flaws rarely have a strong correlation with defects, but none of them is much more frequent than the others (as in Equinox, CDT, and Maven).

From the correlation analysis we conclude that:

- Design flaws correlate with post release software defects, but not strongly in all the analyzed systems.
- There is no design flaw which consistently correlates more than others in all the systems.
- In some systems there is no design flaw which strongly correlates with defects.
- Some systems are characterized by a particular design flaw, i.e., one flaw is strongly correlated with defects much more frequently than all the others.

B. Delta Analysis

In the second part of our experiments we want to investigate whether an addition of design flaws in a class generates bugs. In particular, our goal is to answer the following question: *Do design flaws additions correlate with software defects? Is there a design flaw for which additions consistently correlate more than addition of any other design flaw? What is the relationship between "flaws-defects" correlation and "flaws additions-defects" correlation?*

To study design flaws additions, we need to detect, extract and measure these addition events. We do that by analyzing each row in the design flaw matrix M, as depicted in Figure 10. Given a row, which includes the number of design flaws in all the considered versions of a class, we first compute the deltas between each consecutive pair of versions: A positive delta represents an addition of design flaw in the given class.

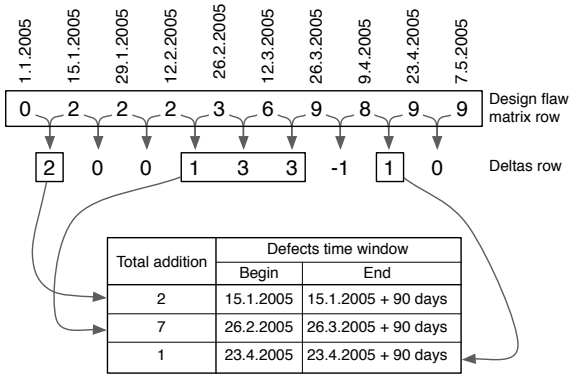

Fig. 10. Extracting and measuring design flaw addition events.

Then, given the deltas row, we group all the sequences of positive values, where a sequence indicate a "longer" addition (see Figure 10). To analyze the relationship between the detected sequences and software defects, we count the number of bugs (linked to the considered class) reported from the beginning to the end of the sequence plus a time window of 90 days[7]. In case the sequence is composed of a single delta, the begin of the sequence coincides with its end. We finally build two lists that we use to compute the correlation: Each element in the lists represents a design flaw addition (a sequence): the first list measures the total addition value, the second one counts the number of defects reported during the addition period (the sequence) over a time window of 90 days

Figure 9 shows, for each system and for each design flaw, the Spearman's correlations between the two lists (the total addition

[7]To be conservative, we use a time window which is half of the post release defect time window.

values and the number of reported defects). For some flaws in some systems, not enough addition events were detected to obtain a significant correlation: In these cases we do not show the correlation.

We conclude that:

- Additions of design flaws correlate with software defects, i.e., introducing a design flaw in a class is likely to generate bugs that affect the class. However, this does not hold for all design flaws in all software systems.
- There is no design flaw addition which consistently correlates more than others in all the systems.

Comparing correlations of defects with absolute numbers of design flaws and design flaws addition, we notice the following facts: In PDE and Lucene no design flaw is strongly correlated with defects (see Figure 8), but at the same time, adding any design flaw in these systems is likely to introduce bugs, as the correlations between flaw additions and defects is relevant for all the design flaws (with the exception of feature envy in PDE). For systems characterized by a particular flaw, an addition of that flaw is not likely to introduce defects or, at least, not as much as other flaws. For example, in Maven brain method is far more frequently correlated with defects than other flaws; However, an addition of brain methods in Maven is not as likely to introduce defects as an addition of shotgun surgery. The same happens for Equinox with shotgun surgery and CDT with intensive coupling.

C. Wrapping Up

The goal of our experiments was to answer a number of questions concerning the frequency of design flaws in the analyzed system, their correlation with software defects and whether their introduction to software entities is likely to generate bugs. The experiments showed that feature envy is the most frequent design flaw, but it is not the most correlated with software defects. Our correlation analysis demonstrated that none of the analyzed design flaws is more correlated with defects than others consistently across systems. Similarly, adding design flaws is likely to introduce defects in many (but not all) software systems, but no design flaw addition correlates with defects more than the others consistently across systems.

Moreover, we found that some software systems are characterized by a specific design flaw "*f*" (different from system to system), in the sense that in these systems the flaw *f* is strongly correlated with defects much more frequently (across versions) than all the others. Interestingly, by performing deltas analysis, we discovered that an addition of *f* in these systems is not likely to introduce defects, or at least not as much as other flaws which are less correlated with defects than *f*. A possible explanation of this finding is that these systems have the flaw *f* in their "nature" and therefore developers know better how to deal with it without generating defects.

We also found systems in which no design flaw is strongly correlated with defects. Adding any flaw in such systems is likely to introduce defects. This corroborates our previous explanation: As long as no design flaw is not in the "nature" of these systems, an addition of any of them introduce defects. We can hypothesize that this happens because developers do not know how to deal with them. However, this is a speculation which needs to be backed up by interviewing the developers of these software systems.

V. Threats to Validity

Threats to construct validity regard measured variables that may not actually measure the conceptual variable. A first construct validity threat concerns the linking of bugs with versioning system files and, subsequently, with classes. The algorithm we use cannot guarantee that all the links are retrieved: For example those links that do not have a bug reference id in the commit comment are not found. However, this is the state of the art in linking bugs to versioning system files, and it is broadly used in literature [16], [17]. A second threat concerns inner classes: As stated in Section III, it is not possible to distinguish reports linking inner classes from reports linking their containing class, because of the file based nature of the content management systems used in the software projects we consider, For this reason, we do not consider inner classes. One way to avoid this problem would be to use heuristics that can detect which class (i.e., the container class or one of its inner classes) is affected by the bug. Such heuristics would be based on bug descriptions and the source code of the classes. A last construct validity threat concerning defects is due to the noise affecting Bugzilla/JIRA repositories. Antoniol et al. showed that a considerable fraction of problem reports marked as bugs are problems not related to corrective maintenance [20]. As part of our future work, we plan to apply the approach that was proposed by Antoniol et al. to filter "non bugs" out.

A construct validity threat concerning design flaws is that we defined them using the *detection strategies* suggested by Marinescu [3] to detect violations against design guidelines. These guidelines are based on thresholds statistically assessed on 45 Java systems. Although such a comprehensive number of software systems, from various projects and domains, was considered, they do not cover all the possible cases, and this can vary the effectiveness of the thresholds in identifying design flaws.

Threats to statistical conclusion validity concern the relationship between the treatment and the outcome. In our approach we use the Spearman's correlation coefficient to evaluate the relationship between design flaws and software defects, and all the correlations are significant at the 0.01 level.

Threats to internal validity concern external factor that may affect an independent variable. We are not aware of any external factor that can affect independent variables of our case studies. However, since these threats are often caused by human intervention, we decided to limit the human factor as more as possible. For this reason, we choose to not consider bug severity as a weight factor when evaluating the number of defects in the experiment. In fact, Ostrand et al. reported how these severity ratings are highly subjective and inaccurate in industrial settings, because of political considerations not related to the importance of the fix to be made [21]. This is also the case for open-source software systems: For example the

severity of a defect can be changed to increase the reputation of the developer who fix it; or defects can be reported as more severe than necessary, so that developers who actually make the change focus on it quickly.

Threats to external validity concern the generalization of the findings. In our experiment, there are two threats regarding to this category: first we considered open-source software systems only. Differences between open-source and industrial development could change our results. However, as also reported by Lanza and Marinescu [4], design flaws also appear in industrial software systems and can be effectively detected using the employed *detection strategy*. Second, we only considered software systems developed in the Java programming language. This affect the generalization of our findings. However, having the possibility to use the same parser for all the case studies ensure that all the code metrics are defined identically for each system, and that we can avoid threats due to behavior differences in parsing, a known issue for reverse engineering tools [22]. Nevertheless, we plan to conduct our analysis to industrial systems as well as systems written in other object-oriented languages.

VI. Related Work

We divide the related work in two parts: In the first we analyze the current research on detecting design flaws and we compare our approach against it, while in the second we look at other related work in the field of defect analysis and prediction.

A. Design flaw detection

A number of approaches have been devised to address the problem of detecting and correcting design flaws in object-oriented software systems. Marinescu proposed the *detection strategies* that we use in our work, as metrics-based composed logical conditions [3]. Studying various large-scale software systems, Marinescu provided evidence that these strategies accurately spot design issues in object-oriented programs [3].

Raţiu and Gîrba [23] applied such detection strategy concept to find design problems revealed by the software system history. Trifu et al. propose correction strategies to refactor design problems detected using the suite of detection strategies defined by Marinescu [6].

Our research is especially engendered by the work by Lanza and Marinescu [4]. They expand the *detection strategies* previously proposed by Marinescu [3], and -analyzing statistical information from many industrial projects and generally accepted semantics- they deduce many single and combined threshold values for the detection. They show in detail how to identify disharmony patterns in code, which solution strategies can be used, and how to devise possible remedies.

Salehie et al. proposed a metric-based heuristic framework to detect and locate object-oriented design flaws similar to those illustrated by Marinescu [5].

Wettel and Lanza continue the research on "design disharmonies" through *disharmony maps*: A visualization-based approach to locate, in large systems, software artifacts that are flawed according to the *detection strategies* mentioned above [7]. They display software systems using a 3D visualization technique based on a city metaphor, and they enrich such visualizations with the results returned by a number of detection strategies. They both render the static structure and the design problems that may affect a subject system, and they evaluate their approach on various open-source Java systems.

When considering the research on design flaws and on metrics-based logical conditions that are able to identify them, the first concern is assessing whether these metrics can detect real bad design quality; followed by how bad design can be refactored. In our case, we "stand on the shoulders" of this previous research and, having enough evidence that detection strategies actually identify bad design in the code, we want to see how design flaws are related to software defects: the tangible effect of poor software quality.

B. Defect analysis and prediction

In 2003 Subramanyam and Krishnan pointed out that empirical evidence linking the object-oriented programming paradigm and project outcomes was scarce [24]. After that, much work has been done on evaluating the capability of source code characteristics (i.e., metrics) to predict and correlate to the outcome of projects, in terms of defect prediction (e.g., [21], [25]–[27]), integration failure prediction (e.g., [28]), or post-release defect correlation (e.g., [17], [19]).

One of the first approaches to prove that object-oriented metrics correlate with defects was proposed by Basili et al. [29]. Later Subramanyam et al. provided empirical evidence, through eight industrial case studies, that object-oriented metrics are significantly associated with defects [30]. Then other object-oriented metrics have been extensively assessed (e.g., [17], [31]), and other approaches to bug prediction exploiting various measures of coupling and cohesion, or use the information contained in software relationships, have been proposed. For example, Marcus et al. introduced a measure of cohesion, called Conceptual Cohesion of Classes, and used it to predict faults in three large open-source systems [32].

As opposed to the mentioned research that considers single source code metrics, or a sum of them, to find a relation with defects, our work is the first one considering the presence of design flaws, identified with detection strategies.

Code smells have been also evaluated by Mäntylä and Lassensius, in order to see to what extent they can be used as a basis for subjective evaluation of *code evolvability* [8]; while Khomh et al. showed that classes containing code smells change more frequently than others, and that specific smells are more correlated than others to *change-proneness* [9]. Differently, our approach evaluate if class design flaws are related to *defects*.

VII. Conclusion

Design flaws are known to have a negative impact on quality attributes of software systems, for example in their flexibility, or maintainability. In this paper, we proposed an extended analysis of the relationship between design flaws and software defects: We studied design flaws and software defects in six different

software systems developed by independent development teams and emerging from the context of two unrelated communities.

Looking at the frequencies of design flaws in these projects, we discovered that *Feature Envy* is consistently the most recurring design flaw. The Feature Envy disharmony refers to methods that access more the data of other classes, than the data of the class containing it. It might be a sign that the method was misplaced and that it should be moved to another class. The most significant aspect about Feature Envy is that it is a sign of an improper distribution of a systems intelligence.

Afterward, our analysis, spreading over the data from a minimum of two years in the history of the chosen systems, showed that design flaws do correlate with software defects. Also, no flaw consistently correlates more than others across all the different systems.

Finally, we found that an increase in the number of design flaws is likely to generate bugs, and still, there is no design flaw addition that consistently correlates more than others across the totality of the systems.

To make our experiments extendible and reproducible by other researchers, our data set, and in particular the design flaws and post release defects matrixes, is publicly available[8].

A. Future Work

In the future we plan to extend our experiments by analyzing more software systems. We also want to investigate whether programming languages play a role by analyzing software systems written in different object-oriented languages, such as C++ and Smalltalk. Finally, we plan to corroborate the speculations done in this paper by conducting qualitative research with software developers.

Acknowledgements. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA" (SNF Project No. 118063).

REFERENCES

[1] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 02, no. 3, pp. 17–23, 2000.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison Wesley, 1995.

[3] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance (ICSM'04)*. Los Alamitos CA: IEEE Computer Society Press, 2004, pp. 350–359.

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

[5] M. Salehie, S. Li, and L. Tahvildari, "A metric-based heuristic framework to detect object-oriented design flaws," in *Proceedings of ICPC 2006 (14th IEEE International Conference on Program Comprehension*, 2006, pp. 159–168.

[6] A. Trifu, O. Seng, and T. Genssler, "Automated design flaw correction in object-oriented systems," in *Proceedings of CSMR 2004 (the 8th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2004, pp. 174–183.

[7] R. Wettel and M. Lanza, "Visually localizing design problems with disharmony maps," in *Proceedings of Softvis 2008 (4th ACM International Symposium on Software Visualization)*. ACM Press, 2008, pp. 155–164.

[8] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Enggineering*, vol. 11, no. 3, pp. 395–431, 2006.

[9] F. Khomh, M. D. Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," *Proceedings of WCRE 2009 (16th IEEE Working Conference on Reverse Engineering)*, pp. 75–84, 2009.

[10] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[12] A. Riel, *Object-Oriented Design Heuristics*. Boston MA: Addison Wesley, 1996.

[13] S. Pinker, *How the Mind Works*. W. W. Norton, 1997.

[14] S. Demeyer, S. Tichelaar, and S. Ducasse, "FAMIX 2.1 — The FAMOOS Information Exchange Model," University of Bern, Tech. Rep., 2001.

[15] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of Moose: an agile reengineering environment," in *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*. ACM Press, 2005, pp. 1–10.

[16] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings International Conference on Software Maintenance (ICSM 2003)*. Los Alamitos CA: IEEE Computer Society Press, Sep. 2003, pp. 23–32.

[17] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of ICSEW 2007 (29th International Conference on Software Engineering Workshops)*. Washington, DC, USA: IEEE Computer Society, 2007, p. 76.

[18] M. Triola, *Elementary Statistics*. Addison-Wesley, 2006.

[19] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of ICSE 2008 (30th International Conference on Software Engineering)*, 2008.

[20] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of CASCON 2008*. ACM, 2008, pp. 304–318.

[21] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *Proceedings of ISSTA 2004 (ACM SIGSOFT International Symposium on Software testing and analysis)*. ACM, 2004, pp. 86–96.

[22] R. Kollmann, P. Selonen, and E. Stroulia, "A study on the current state of the art in toolsupported uml-based static reverse engineering," in *Proceedings of WCRE 2002 (9th IEEE Working Conference on Reverse Engineering*, 2002, pp. 22–32.

[23] D. Raţiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proceedings of CSMR 2004 (the 8th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2004, pp. 223–232.

[24] R. Subramanyam and M. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, April 2003.

[25] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the ICSE 2006 (28th International Conference on Software Engineering*. New York, NY, USA: ACM, May 2006, pp. 452–461.

[26] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[27] ——, "Automating algorithms for the identification of fault-prone files," in *Proceedings of ISSTA 2007 (ACM SIGSOFT International Symposium on Software testing and analysis*. ACM, 2007, pp. 219–227.

[28] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," *SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 55–64, 2002.

[29] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[30] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[31] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 771–789, 2006.

[32] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[8] http://www.inf.usi.ch/phd/dambros/tools/bugs-disharmonies.php