Studying the Relation between Anti-patterns in Design Models and in Source Code

Bilal Karasneh
Leiden Institute of Advanced CS
LIACS
Leiden, the Netherlands
b.h.a.karasneh@liacs.leidenuniv.nl

Michel R.V. Chaudron
Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
chaudron@chalmers.se

Foutse Khomh and Yann-Gaël Guéhéneuc
Génie Informatique et Génie Logiciel
École Polytechnique de Montréal
Montréal, Canada
{foutse.khomh, yann-gael.gueheneuc}@polymtl.ca

***Abstract*—There exists a large body of work on the specification and detection of anti-patterns in the source code of software systems. However, there are very few studies on the origins of the occurrences of anti-patterns in the source code: do the very design of the systems lead to the occurrences of anti-patterns or are anti-patterns introduced during implementation? Knowing when anti-patterns are introduced could help software designers and developers improve the quality of the source code, for example by eliminating fault-prone anti-patterns early during the design of the systems, even before their implementation. Therefore, we detect occurrences of anti-patterns in design models and in the source code of some systems, trace these occurrences between design and implementation, and study their relation and impact on the source code. First, we analyze both the UML design models and the source code of 10 open-source systems and show that anti-patterns exist in design models. We observe that, on average, 37% of the classes in the design models that belong to anti-patterns also exist in the source code and also play roles in the same anti-patterns. Second, we investigate two open-source systems to assess the impact of the anti-patterns in their design models on the source code in terms of changes and faults. We show that classes that have anti-patterns in the design models have more changes and faults in the source code. Our results suggest that the design of the systems lead to anti-patterns and that the anti-patterns impact negatively the change- and fault-proneness of the classes in the source code. Thus, designers should be wary of anti-patterns in their design models and could benefit from tools that detect and trace these anti-patterns into the source code.**

***Index Terms*—Anti-patterns, design models, source code.**

I. Introduction

Many researchers and practitioners propose and use approaches to assess the quality of the source code of software systems and, consequently, do not pay attention to the quality of the design models of the systems. Yet, if the implementations of the systems follow their design models, the quality of the design models is important and could impact the quality of the source code. Therefore, we ask the following question: **do the very design of the systems lead to the occurrences of anti-patterns or are anti-patterns introduced during implementation?** Some studies [1]–[3], investigated the evolution of anti-patterns in systems and observed that anti-patterns are not necessarily introduced only in the source code during development and evolution. They reported that many classes are "born" as anti-patterns. Yet, there are few studies about the impact that anti-patterns occurring in the design models of systems have on their source code, because it is challenging to collect convincing data.

In this paper, towards answering our questions above, we conduct two studies. We conduct these studies using both hand-made, designers' models of software systems and the implementation of these models as source code. To the best of our knowledge, these are among the only studies using designers' models and the corresponding source code. There are also the only studies in which *design anti-patterns* [4] are detected in *true design models* and in models of the *design* of the source code obtained through reverse-engineering.

First, we consider four anti-patterns, Complex, Large, Lazy, and LongMethod classes, and study 10 open-source systems for which design models and source code are available, by detecting and tracing occurrences of anti-patterns in designs and source code. We choose these anti-patterns because they are reported to increase systems' change-and fault-proneness [5] and they are a source of technical debt that should be managed by developers. We report on the possibility to detect these anti-patterns in designs and on their prevalence both in the design models and source code of the systems. We also trace manually the classes in the models with those in the source code to assess whether the same classes experience the same anti-patterns. Results show that 37% of the classes in the design models that belong to anti-patterns also exist in the source code and play roles in the same anti-patterns.

Second, we focus on seven types of anti-patterns, Complex, Large, Lazy, Blob, ClassDataShouldBePrivate, RefusedParentBequest, and BaseClassShouldBeAbstract classes, and study the effect that anti-patterns from the design models have on the change- and fault-proneness of the classes in the source code. For two open-source systems, for which we have different versions of their design models and source code, we compute changes and faults for four categories of classes: classes in source code and–or designs belonging or not to anti-patterns. Results show that there are significant differences in terms of changes and faults between the different categories of classes. Classes appearing in design models (*i.e.*, classes that were modelled during the conception) have more changes and faults than classes that appear only in the source code (*i.e.*, classes that were added only later during the implementation of the system). Classes that have anti-patterns in designs and that

exist in source code have more changes and faults than classes in designs and source code without anti-patterns.

This result suggests that anti-patterns that appear early on during the conception of the system (*i.e.*, in design models) impact negatively the source code. Designers should be wary of these anti-patterns and would benefit from tools that can track and control these anti-patterns as early as possible. Preventing anti-patterns during the design phase could improve the quality of the source code of a system and prevent the system from experiencing problems related to anti-patterns (*i.e.*, more changes and faults [5]).

This paper makes the following contributions: (1) we show that anti-patterns exist in design models and they can be detected; (2) we show that the anti-patterns that occur in design models percolate to the source code; (3) classes that are modelled during the design phase tend to have more changes and faults than others; and, (4) classes that have anti-patterns in designs have more changes and faults in the code than others.

We organize the paper as follows: We present related work in Section II. We provide background information in Section III. We describe the first study and its results in Section IV. We report on the second study in Section V. We discuss our studies in Section VI. We conclude the paper and outline future works in Section VII.

II. Related Work

Since their inception in software engineering, design patterns (i.e., reusable solutions to recurring design problems) [6] and anti-patterns (i.e., poor solutions to design and implementation problems) [4] have been the subject of many research works, which focused on their specification [7], detection [8], and on the analyses of their impact and life-cycle [1]–[3].

There are many research works on the definition, specification, detection, correction, and the life-cycles of code smells and design anti-patterns. For the sake of space and because we focus on code smells and design anti-patterns in this paper, we describe here three works that (1) showed that code smells and design anti-patterns do impact negatively class change- and fault-proneness, (2) studied the introduction and removal of some anti-patterns qualitatively, and (3) reported four lessons on their life-cycles. We describe specification and detection techniques in Section III.

Khomh et al. [9] investigated the impact of anti-patterns on classes in object-oriented systems by studying the relation between the presence of anti-patterns and the change- and fault-proneness of the classes. The authors showed that in 50 out of 54 releases of the four systems, classes participating in anti-patterns are more change and fault-prone than others.

Vaucher et al. [2] studied the "God class" design anti-patterns, which describes large classes that "know too much or do too much". Although literature postulated that God classes are created by accident as functionalities are incrementally added by developers to central classes over the course of their evolution, they assumed that, in some systems, a God class is created by design as the best solution to a particular problem because, for example, the problem is not easily decomposable or strong requirements on efficiency exist. They studied the life-cycles of God classes in the source code of Eclipse JDT and Xerces: how they arise, how prevalent they are, and whether they remain or they are removed as the systems evolve over time. They distinguished between those classes that are God classes by design (good code) from those that occurred by accident (bad code). They concluded with prevention refactorings and mechanisms to track the appearance and disappearance of design anti-patterns.

Following this previous study, Tufano et al. [1] studied five design anti-patterns in the evolution histories of the source code of Android, Apache, and Eclipse and drew four lessons from their study regarding the life-cycles of the anti-patterns: (1) classes often play roles in these anti-patterns from their inception in the systems, (2) the metric values of the classes that start to play role in the anti-patterns during the evolution have specific trends, (3) refactoring operations, in addition to other changes, may lead to the introduction of these anti-patterns, and (4) time pressure is the main reason for these anti-patterns. Thus, the authors confirmed that the occurrences of anti-patterns are not necessarily due to developers' lack of skills but could be due to the very design of the systems.

All these studies considered occurrences of code smells and design anti-patterns in the source code of the studied systems (and their revisions) or in design models *reverse-engineered* from the source code of these systems. They did not study the prevalence of the antipatterns in design models created *before* (and–or during) development in comparison to that in the source code implementing these design models. The following study aims at confirming the observations that, in some *designs*, code smells and design anti-patterns are present from the very beginning of the inception of the systems.

A few research studies investigated the effect of using UML on software maintenance. Arisholm et al. [10] conducted a controlled experiment using students to evaluate the impact of UML documentation on the software maintenance. The results show that the availability of UML documentation may improve the functional correctness of changes. Dzidek et. al [11] conducted a controlled experiment using professional software developer and found that using UML significantly improves the functional correctness of changes during maintenance. Fernández-Sáez et al. [12] conducted a family of experiments consisting of one controlled experiment and three replications with students, to investigate whether and how the level of details (LoD) of UML diagrams impacts maintenance tasks. Results show that there is no strong statistical evidence of the influence of different LoDs. However, they conclude that low LoD helps for modification of the source code and high LoD helps to understand the system.

The study of Nugroho et al. [13] is close to our study, where they investigate the impact of using design models (class diagrams and sequence diagrams) on the quality of the implementation measured by defect density. Their study was conducted using industrial systems. The result shows that the classes appearing in the design models have significantly lower defect density than those that are not in the design models.

To the best of our knowledge, we propose the first study on the impact of anti-patterns in design models on the change- and fault-proneness of (corresponding) classes in source code.

III. Background

We conduct our studies using the design models of some open-source software systems, selected randomly from the UML Repository. We use existing tools to detect occurrences of anti-patterns in design models and source code.

A. UML Repository

Many researchers argue that the lack of sharable and searchable design models impairs the ability to perform and replicate empirical studies, thus slowing down the rate at which novel, effective approaches are defined and studied [14]–[16]. The absence of an open community of software designers is also an impediment to the uptake of modeling. The UML Repository[1] attempt to overcome this lack. It is a unique repository containing pairs of designers' models (mostly UML class diagrams) linked to the corresponding source code, when available. It is a free online repository (free registration is required for full use). The two first authors built the UML Repository using design models found on the Internet in open-source repositories and by approaching researchers in our community [17]. The UML Repository aims to grow an active community of researchers and practitioners interested in design modeling: experts, developers, teachers, and students.

The UML Repository contains dozens of pairs of design models and source code written in C++ and Java as well as metadata and other information, including design metrics and occurrences of anti-patterns retrieved on design models and source code. It is searchable to find suitable case studies. It currently[2] contains 810 UML design models with URLs to the corresponding source code if available. It uses a dedicated subsystem, Img2UML, for converting images of UML diagrams, e.g., class diagrams, in JPEG and NMP format into XMI files [18]. Img2UML uses image recognition techniques to recognize UML diagram elements in images, including class names, attributes, operations, and relationships between classes. It generates XMI files that are compatible with many UML CASE tools [19]–[21].

B. Anti-patterns Specification and Detection

For this study, we specify the Lazy Class in terms of the number of methods defined in the classes. We detect Complex Classes using the number of methods and the relationships among classes: a class that defines many methods and that has many relationships (in or out) is inherently complex. Long Method can only be computed on classes from the source code because we need the number of statements in the methods. Finally, we specify Large Classes as the "opposite" of a Lazy Class, in terms of the number of methods in the class. A Blob class is a large class that declares many fields and methods with a low cohesion. We specify ClassDataShouldBePrivate as a class that exposes its field. A RefusedParentBequest class is a class that redefines inherited methods. Finally, a BaseClassShouldBeAbstract anti-pattern is a class that has many subclasses without being abstract.

The details of the specification and detection of the anti-patterns is outside the scope of this paper because we mostly adapt and–or reuse the specifications and detection algorithms used in previous work. Indeed, we use the Ptidej[3] tool suite, which implements the anti-pattern detection approach DECOR (Defect dEtection for CORrection), to identify occurrences of anti-patterns in both design models and source code [7]. DECOR is an approach based on the automatic generation of detection algorithms from rule cards. It proposes the descriptions of different anti-patterns and provides generation and detection algorithms. It converts anti-patterns descriptions automatically into detection algorithms and identifies the occurrences of these anti-patterns in design models describing either class diagrams or the source code of a system.

We apply DECOR in three steps: first, we reuse/define rule cards describing the anti-patterns of interest through a domain analysis of the literature. From the rule cards, we generate a detection algorithms. Finally, we apply the detection algorithms on the design models and source code of systems to detect the occurrences of the anti-patterns. DECOR has appropriate performance, precision, and recall for our study. It has been reported [7] to achieve a precision of more than 60% and a recall of 100%.

DECOR can be applied on any object-oriented system through the use of the PADL [22] meta-model and POM framework [23]. PADL describes the structure of systems and a subset of their behavior, i.e., classes and their relationships. POM is a PADL-based framework that implements more than 60 structural metrics. We apply DECOR on models obtained either from the class diagrams available in the UML repository, by parsing the corresponding XMI files, or by parsing the corresponding C++ and Java source code.

IV. Origins of Anti-Patterns

For the first study, we randomly selected ten open-source systems from our UML Repository that are available in Github, SourceForge, and Google code. Table I presents the characteristics of the systems used in the study. We now report the results of the detection of anti-patterns in the design models and the source code of the 10 systems. First, we report about some analysis of the numbers of classes in the design models and their source code. Then, we comment about anti-patterns detected in both design models and their source code.

A. Descriptive Statistics

Table II shows an overview of classes in the designs and source code, whose proportions are computed using Equation 1. As expected, the numbers of classes in the source code are higher than in the designs, but we found that some classes in the designs are missing in the source code. We also expected

[1] http://models-db.com/

[2] As of October 17th, 2015.

[3] http://www.ptidej.net

TABLE I
STUDIED SOFTWARE SYSTEMS

Project Name	Descriptions	URLs
ArgoUML	An open source UML modeling tool	http://argouml.sourceforge.net
Annoyme	Adds beautiful typewriter sounds to Desktop keyboards	https://github.com/dedeibel/annoyme
JCoAP	Java implementation of the Constrained Application Protocol (CoAP)	https://github.com/dapaulid/JCoAP
JGAP	Package of Genetic Algorithm and Genetic Programming	http://jgap.sourceforge.net
Kartjax	Cross-platform, tools to create object, animation and map	https://code.google.com/p/kartjax/
Mars_Simulation	Project to create a simulation of future settlements on Mars	http://mars-sim.sourceforge.net
Msv_Poker	Poker Game (poker server and poker client)	https://github.com/mihhailnovik/msvPoker
Neuroph	Lightweight Java neural network framework to develop network architecture	http://neuroph.sourceforge.net
Syntax-analyzer	Syntax analyzer for (*.lng) file	https://github.com/myzone/syntax-analyzer/
Wro4j	Web resource optimizer for Java	http://code.google.com/p/wro4j

TABLE II
CLASSES IN DESIGN MODELS VS. CLASSES IN SOURCE CODE

Project Names	# Classes in Models	# Classes in Source Code	Proportions of Classes
Annoyme	17	59	0.29
ArgoUML	51	909	0.03
JCoAP	21	52	0.40
JGAP	19	191	0.10
Kartjax	29	36	0.80
Mars_Simulation	32	953	0.03
Msv_Poker	22	55	0.40
Neuroph	26	179	0.15
Syntax-analyzer	26	34	0.76
Wro4j	28	289	0.10

this observation because the designs are *conceptual* models and are often refined by developers during implementation. However, these refinements are not always documented back in the designs.

$$\frac{Number\ of\ classes\ in\ a\ project\ models}{Number\ of\ classes\ in\ a\ project\ source\ code} \quad (1)$$

$$\frac{No.\ of\ classes\ exist\ in\ both\ (Models\ and\ Source\ code)}{Number\ of\ classes\ in\ Models} \quad (2)$$

$$\frac{No.\ of\ classes\ exist\ in\ both\ (Models\ and\ Source\ code)}{Number\ of\ classes\ in\ Source\ code} \quad (3)$$

$$\frac{No.\ of\ same\ anti-patterns\ in\ the\ same\ classes\ in\ both\ (Models\ and\ source\ code)}{No.\ of\ classes\ exist\ in\ both\ (Models\ and\ Source\ code)} \quad (4)$$

TABLE III
PROPORTIONS OF CLASSES IN BOTH DESIGN MODELS AND SOURCE CODE

Project Names	# Classes Existing in Models and Source Code	Proportions of Classes according to Equation (2)	Proportions of Classes according to Equation (3)
Annoyme	14	0.82	0.24
ArgoUML	44	0.86	0.05
JCoAP	3	0.14	0.06
JGAP	18	0.95	0.09
Kartjax	1	0.03	0.03
Mars_Simulation	29	0.91	0.03
Msv_Poker	13	0.59	0.24
Neuroph	24	0.92	0.13
Syntax-analyzer	1	0.04	0.03
Wro4j	11	0.39	0.11

Table II shows the ratios of the numbers of classes in the design models over the number of classes in the implementation (*i.e.*, the source code) while Table III shows the ratios of the numbers of classes that exist both in design models and the source code over respectively, the number of classes in the design model and the number of classes in the source code. We did a manual checking for design models and the source code for each system to find classes that exist in both of design models and source code. We chose to report ratios because we observed that, although the majority of classes in design models are also present in the source code, there are still some classes that were modelled during the conception of the system (*i.e.*, present in the design model) but not implemented in the source code. In general, classes in the design models represent only a fraction of the total numbers of classes contained in the source code of a system. In Section IV-B, we show that anti-patterns in design models are transferred to the implementation.

B. Anti-patterns Statistics

We can only detect three of the four considered anti-patterns in the design models: Complex Class, Large Class, and Lazy class. We can detect LongMethod only in the source code because designs are abstract representations of the systems and they contain only method signatures without the implementation details needed to compute the lengths of the methods. This observation shows that other source of information are needed to compute some anti-patterns in design models, e.g., requirements, and that some design-specific anti-patterns may exist and should be studied in future work. Table IV reports the numbers of anti-patterns found in the design models and the source code of the systems.

We observe that, in JCoAP, Kartjax, and Syntax-analyzer, the numbers of anti-patterns in the design models is higher than in the source code, which is unexpected but can be because of (1) the small sizes of these systems and (2) the difference between their design models and implementation. Reason (2) could be due to either the design models being outdated or them being used only as sketches while developers changed the designs during implementation, "on the fly".

TABLE IV
ANTI-PATTERNS IN BOTH DESIGN MODELS AND SOURCE CODE

Project Names	# Anti-patterns in the Models	# Anti-patterns in the Source Code	# Long-Method Anti-patterns in the Source Code	# Same Anti-patterns in the Same classes in Models and Source Code
Annoyme	10	16	0	5
ArgoUML	18	524	270	10
JCoAP	4	2	0	3
JGAP	13	252	130	5
Kartjax	17	4	2	1
Mars_Simulation	22	370	206	3
Msv_Poker	11	18	8	4
Neuroph	12	41	27	4
Syntax-analyzer	9	5	1	1
Wro4j	25	215	135	12

TABLE V
PROPORTIONS OF CLASSES IN DESIGN MODELS THAT TRANSFER THE SAME ANTI-PATTERNS IN SOURCE CODE

Project Names	# Anti-patterns in the Models	# Same anti-patterns in the same classes in CD and Sc	Proportions of same classes have same anti-patterns in CD and Sc
Annoyme	10	5	0.50
ArgoUML	18	10	0.56
JCoAP	4	3	0.75
JGAP	13	5	0.38
Kartjax	17	1	0.06
Mars_Simulation	22	3	0.14
Msv_Poker	11	4	0.36
Neuroph	12	4	0.33
Syntax-analyzer	9	1	0.11
Wro4j	25	12	0.48
Average			**0.37**

Future work should introduce a concept of distance between designs and source code and study the causes for short or long distances.

We compute the proportions of classes that play the same roles in the same anti-patterns in design models and source code based on Equation 4 and Table V shows the proportions of these classes. There is a significant proportions of classes playing the same roles in the same anti-patterns in both design models and source code: **37%** of classes in design models. If we put aside JCoAP, Kartjax, and Syntax-analyzer, the proportions of classes playing the same roles in the same anti-patterns in design models and source code is **39%**.

C. Anti-patterns Details

We now focus on individual anti-patterns and their occurrences in the design models and the source code.

a) Complex Class: Regarding the Complex Class anti-pattern, very few occurrences are found in design models, which means that architects and designers' tend to avoid

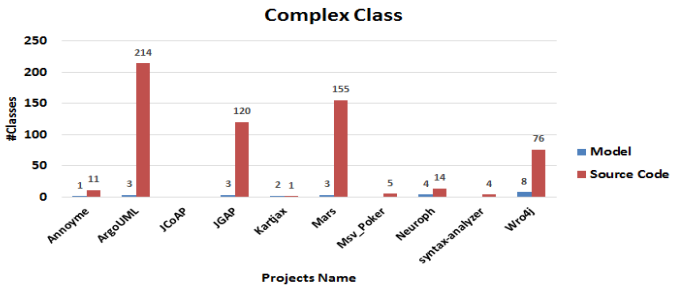

Fig. 1. Numbers of Complex classes in designs and source code

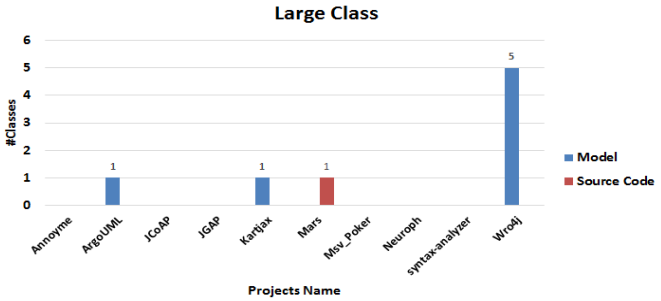

Fig. 2. Numbers of Large classes in designs and source code

excessively complex classes. However, developers do not seem to follow the same care during implementation as we observe proliferations of occurrences of the Complex Class anti-pattern in the source code of ArgoUML, JGAP, Mars, and Wro4j as shown in Figure 1.

We explain this observation by two facts. On the one hand, design models tend to be sketches of the actual implementation and, hence, do not contain all the details and complexity of the source code while the source code must, by its very definition, contain the actual algorithms, which may be intrinsically complex to implement. On the other hand, complex classes in source code tend to arise because of the lack of time for developers to research the best (i.e., simplest) implementation. Hence, it is our experience and observation that source code tends to be inherently more complex than necessary and, therefore, more complex than the design models.

b) Large Class: Occurrences of the Large Class anti-pattern are generally absent from both design models and source code, except for Wro4j, whose design contains five occurrences, as shown in Figure 2. As reported by Vaucher et al. [2], Large Classes are sometimes present in systems because they are the best solution to some problems, for example when the problem is not easily decomposable. Yet, such cases seem to be rare in design models: only one system out of 10 contains occurrences of the Large Class for the same reasons as mentioned above: designers focus on the essentials of classes, developers lack time to introduce proper abstractions and, thus, their tendency to "grow" classes to implement new features.

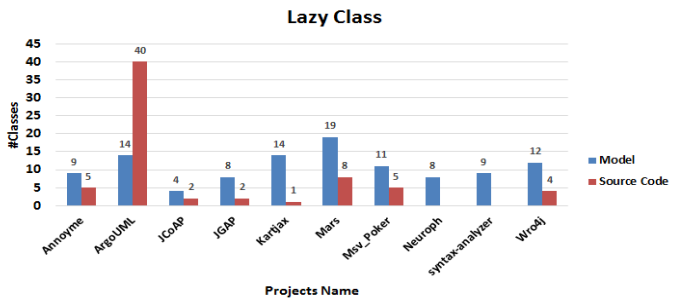

Fig. 3. Numbers of Lazy classes in designs and source code

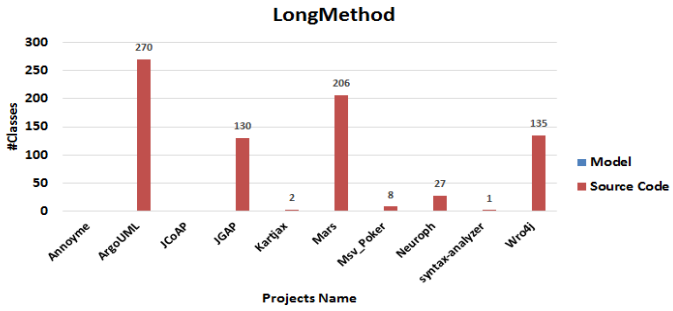

Fig. 4. Numbers of classes with Long Methods in designs and source code

c) Lazy Class: Lazy class, which is the most frequent anti-patterns among the four considered anti-patterns, is more prevalent in designs than in source code as shown in Figure 3.

We explain this result by the fact that designers try to anticipate future evolutions of the systems, which often leads to many abstract classes that do not contain necessarily enough behavior to justify their existence: they are Lazy classes by very definition (and specification in our detection technique.

However, Figure 3 shows that, in all systems but ArgoUML, these excessive abstractions are "corrected" later by developers during the implementation of the systems.

d) Long Method: Occurrences of the Long Method anti-pattern are also introduced in the source code in large number by developers as shown in Figure 4.

Again, we explain this observation and the difference between design models and source code in two ways. First, designs do not contain all the details necessary to identify Long Methods because of their very nature as sketches. Second, developers tend to implement features as fast as possible, under time pressure, and thus cannot take the time required to refactor their code and to avoid long methods.

D. Discussions

We could detect only three of the four considered anti-patterns in design models. We could not find occurrences of the Long Method anti-pattern in design models because its detection is based on the numbers of Line of Code (LOC) of the methods, which is usually not available in design models. Also, we observed that some anti-patterns appear in designs and the same classes in the source code have different anti-patterns. We relate this to the common practice, in both open-source and commercial software development, whereby developers change the designs and evolve the source code during implementation and do not update the design models.

Table III shows that some classes contained in the design models disappear in the source code, which can be considered in two ways. First, having a class in a design model and not having this class in the source code could be a design violation. For example, a designer could have introduced a Facade between two subsystems, which is later removed by developers for the sake of simplicity of implementation or performance of execution. Such a removal could yield to unintended accesses to some subsystems and also reduce information hiding.

However, having less information in design models and not in source code such as classes, can also be the result of a lack of traceability in the project, because developers may have refined the design during implementation while failing to document the modifications and updating the design models. Indeed, updates and changes to the source code without updating the design models is a common malpractice observed by many researchers and practitioners. Hence, our results confirm the software engineering lore that design models are not synchronized with their source code by developers.

In addition, we observe that most of the classes that are in the design models and disappear in source code are Lazy Classes as shown in Figure 3, which confirms our intuition about developers refining the designs because Lazy classes are the result of excessive abstractions. With a better knowledge of the system under development, developers may decide to remove some of the abstractions that result from designers' speculations about future evolutions of their systems.

The average proportion of the classes that exist in both the design models and the source code, and which have the same anti-patterns in both artifacts is 37%, which represents an important proportion of the total numbers of occurrences of the anti-patterns contained in the design models. Hence, by acting early on these anti-patterns, designers and developers could improve the quality of their systems.

Defects contained in design models are known to be particularly expensive if they are not fixed quickly because classes in the designs are the backbone of the source code and, in most designs, are the most important classes in the source code.

Overall, our results confirm that (1) classes in design models may have anti-patterns, which translate to the source code and (2) classes in both design models and source code have the same anti-patterns for 37% of classes in the design models.

V. Design Anti-patterns Impact on Source Code

For the second study, we selected ArgoUML and Wro4j because we can access the class changes and bug reports for all of their versions. For ArgoUML, we use nine different versions and for Wro4j we use six different versions. This study includes comparing classes that exist in both the design model and the source code, with classes that exist only in the

TABLE VI
MODELED AND NON-MODELED CLASSES

Project Names	Modeled classes	Not-Modeled classes	Total
ArgoUML	88	2,731	2,819
Wro4j	197	876	1,073

TABLE VII
MEANS OF NUMBERS OF CHANGES IN ARGOUML AND WRO4J

	Project Name	Categories	Mean
Changes	**ArgoUML**	**Modeled classes**	22
		Non-modeled classes	7.78
	Wro4j	**Modeled classes**	15.17
		Non-modeled classes	6.56

TABLE VIII
MEANS OF NUMBERS OF FAULTS IN ARGOUML AND WRO4J

	Project Name	Categories	Mean
Bugs	**ArgoUML**	**Modeled classes**	0.82
		Non-modeled classes	0.42
	Wro4j	**Modeled classes**	2.91
		Non-modeled classes	1.04

TABLE XI
ANTI-PATTERNS AND NO-ANTI-PATTERN CLASSES

Project Name	Anti-patterns Category	No-anti-patterns Category	Total
ArgoUML	56	32	88
Wro4j	130	69	199

source code, in terms of changes and faults. We also compare classes from the design models that have anti-patterns with those that do not have anti-patterns.

A. Compare classes in the models and classes that exist only in the source code

We divided classes from the source code into two sets: (1) modeled classes, that exist in the design models and source code, and non-modeled classes, that exist only in the source code. To compare the change- and fault-proneness of these two sets of classes, we collected classes in the design models and their corresponding classes in the source code of the two systems. Table VI summarises both sets for both the ArgoUML and Wro4j systems. We checked the normality of changes and faults in both ArgoUML and Wro4j and observed that they are not normally distributed. Therefore, we use Mann-Whitney test to compare the means of the number of changes and of faults between both sets.

We use IntelliJ IDEA[4] to find the corresponding classes in the source code to the classes in the models.

e) Results: Tables VII and VIII show the means of the numbers of changes and faults, respectively, for both modeled classes and non-modeled classes in both ArguUML and Wro4j. There are significant differences between modeled and non-modeled classes in both ArgoUML and Wro4j for both changes and and faults.

f) Discussions: Classes in the designs with corresponding classes in the source code have more changes and faults than classes that exist only in the source code in both ArgoUML and Wro4j among their different versions. We explain this result: classes in designs describe the core classes in source code, so they have more changes between versions.

Also, when it comes to faults, there is a positive relation between the numbers of changes and the number of faults [24]. Therefore, because classes among the Modeled classes have more changes, they tend to have more faults. Moreover, if the design of a system has problems, these problems could be transferred to the source code. Indeed, the implementation should follow the design, which results in developers transferring problems from the design to the source code. Next, we show the relation between the numbers of changes and two code metrics, Line of Code (LOC) and average Cyclomatic Complexity (AvgCyc). Table IX shows the correlations between numbers of changes in ArgoUML and Wro4j with LOC and AvgCyc. It shows that the numbers of changes in Modeled classes have significantly higher correlations with LOC than Non-modeled classes in both ArgoUML and Wro4j: Modeled classes that have have higher LOC have more changes.

Table X shows the correlation between faults, LOC and AvgCyc: the numbers of faults in Modeled classes have significantly higher correlations with LOC than Non-modeled classes in both ArgoUML and Wro4j. More faults exist in Modeled classes that have higher LOC. AvgCyc does not have any correlations with Modeled classes or Non-modeled classes.

B. Comparison of classes that have/do not have anti-patterns in the design

We now divide classes in design models with corresponding classes in the source code into two sets: (1) anti-pattern classes, which contains classes that have anti-patterns in the designs and (2) no-anti-patterns classes, which contains classes that do not have anti-patterns in the designs.

We again use the Mann Whitney test to compare the means of the numbers of changes and faults between both sets because the changes and faults are not normally distributed in the both systems. Table XI summarizes both sets for ArgoUML and Wro4j. We collected anti-patterns, changes, and faults for each class, then entered this data into a database[5].

g) Results: Table XII shows the means of numbers of changes for both sets for both ArguUML and Wro4j. Table XIII shows the means of numbers of faults for both sets, for ArguUML and Wro4j.

Mann Whitney tests show that there are significant differences between the anti-patterns and no-anti-patterns sets in ArgoUML and Wro4j in terms of changes and faults.

h) Discussions: We observe that classes that have anti-patterns in the designs and corresponding classes in the source code of ArgoUML and Wro4j have more changes and faults in the implementation. The broken windows theory [25] states

[4]http://www.jetbrains.com/idea/

[5]http://Models-db.com/SANER2016/ArgoUML_Wro4j.zip

TABLE IX
CORRELATIONS BETWEEN CHANGES, LOC, AND AVERAGE CYCLOMATIC COMPLEXITY

	Systems	Categories	Correlations		R^2	Formulas
			AvgCyc	LOC		
Changes	**ArgoUML**	**Modeled classes**	0.30	0.74	0.54	Y= 0.727 + 0.018(LOC)
		Non-modeled classes	0.21	0.43	0.19	Y=1.352+0.004(LOC)+0.092(AvgCyc)
	Wro4j	**Modeled classes**	0.06	0.62	0.39	Y=-1.937+0.315(LOC)
		Non-modeled classes	-0.00	0.38	0.17	Y=6.191+0.104(LOC)(Avg)(Cyc)

TABLE X
CORRELATIONS BETWEEN FAULTS, LOC, AND AVERAGE CYCLOMATIC COMPLEXITY

	Systems	Categories	Correlations		R^2	Formulas
			AvgCyc	LOC		
Bugs	**ArgoUML**	**Modeled classes**	0.35	0.53	0.32	Y=-0.54+0.001(LOC)+0.041(AvgCyc)
		Non-modeled classes	0.13	0.26	0.07	Y=0.07+0.000(LOC)
	Wro4j	**Modeled classes**	0.02	0.60	0.35	Y=-0.266+0.060(LOC)
		Non-modeled classes	-0.01	0.40	0.18	Y=0.953+0.026(LOC)-0.417(AvgCyc)

TABLE XII
MEANS OF NUMBERS OF CHANGES IN ARGOUML AND WRO4J

	Project Name	Categories	Mean
Changes	**ArgoUML**	**Anti-patterns**	30.8
		No-anti-patterns	06.59
	Wro4j	**Anti-patterns**	16.60
		No-anti-patterns	12.26

TABLE XIII
MEANS OF NUMBERS OF FAULTS IN ARGOUML AND WRO4J

	Project Name	Categories	Mean
Bugs	**ArgoUML**	**Anti-patterns**	1.20
		No-anti-patterns	0.15
	Wro4j	**Anti-patterns**	3.16
		No-anti-patterns	2.38

that a broken window may lead to a general degradation of the whole environment and we argue that developers should solve these design problems before transferring them to the source code to reduce implementation and maintenance effort.

Similarly, we argue that when design problems are not fixed quickly, they tend to propagate in the system causing other problems. It is therefore, important to track and fix design problems as early as possible in the development cycle. The results of this study show that software organizations can make use of anti-patterns detection tools like Ptidej during the design phase and track and fix anti-patterns in their software system as early as the design phase. Thus, anti-patterns detection tools will help prevent defects that could occur because of anti-patterns. Indeed, the refactoring of anti-patterns should be easier and less costly at modeling level than during implementation.

Table XIV and Table XV show the correlations between numbers of changes and faults in Anti-patterns classes and No-anti-patterns classes with both LOC and AvgCyc. Table XIV shows that the numbers of changes have significantly higher correlations with LOC in No-anti-patterns classes than Anti-patterns classes in both ArgoUML and Wro4j: in systems with anti-patterns, size is not the only factor affecting change-proneness. The occurrence of anti-patterns also contributes to the occurrence of changes.

Table XV shows the correlations between numbers of faults, LOC, and AvgCyc in ArgoUML and Wro4j: the numbers of faults in No-anti-patterns classes have significantly higher correlations with LOC than Anti-patterns classes in both ArgoUML and Wro4j. More faults occur in bigger No-anti-patterns classes. For Anti-pattern classes, there is no strong correlation with LOC, which means that faults exist no matter the size of the classes. For AvgCyc, the correlations with changes is higher in Anti-patterns classes, which means that complex classes have more changes.

VI. DISCUSSIONS

This section discusses the threats to validity of our studies following common guidelines for empirical studies [26]. It also provides preliminary recommendations.

i) Threats to construct validity: concern our implicit assumption that each anti-pattern is of equal importance. Future work must study the impact of the anti-patterns found in models in well-used dependent variables, such as class change- and fault-proneness, to assert whether all anti-patterns in models have similar impact in the source code during implementation and maintenance.

j) Threats to internal validity: concern our selection of systems, tools, and analysis method. The accuracy of Ptidej impacts our results. However, Ptidej has been successfully used in multiple studies [1], [2], [8], [9] and has been reported to achieve high precision and recall [7]. However, other anti-pattern detection techniques and tools should be used to confirm our results.

In addition, the level of details of UML models affects the detection of anti-patterns in the models, for example we could not detect occurrences of the Long Method anti-pattern in models. It is possible that the lack of detailed information in models also affected the detection of other anti-patterns. However, because the detection of these anti-patterns require only high-level information about methods, relations, and hierarchies (which are contained in the model), we are

TABLE XIV
CORRELATIONS BETWEEN CHANGES, LOC, AND AVERAGE CYCLOMATIC COMPLEXITY

	Systems	Categories	Correlations		R^2	Formulas
			AvgCyc	LOC		
Changes	**ArgoUML**	**Anti-patterns classes**	0.41	0.67	0.44	y=0.997+0.017
		No-anti-patterns classes	0.33	0.85	0.72	Y=0.149+0.028(LOC)
	Wro4j	**Anti-patterns classes**	0.06	0.59	0.35	Y=0.989+0.245(LOC)
		No-anti-patterns classes	0.02	0.70	0.48	Y=-0.698+0.465(LOC)

TABLE XV
CORRELATIONS BETWEEN FAULTS, LOC, AND AVERAGE CYCLOMATIC COMPLEXITY

	Systems	Categories	Correlations		R^2	Formulas
			AvgCyc	LOC		
Bugs	**ArgoUML**	**Anti-patterns classes**	0.48	0.61	0.42	Y=-0.90+0.001(LOC)+0.054(AvgCyc)
		No-anti-patterns classes	0.35	0.81	0.65	Y=-0.072+0.001(LOC)
	Wro4j	**Anti-patterns classes**	-0.01	0.57	0.49	Y=0.377+0.046(LOC)
		No-anti-patterns classes	0.01	0.70	0.48	Y=-1.534+0.097(LOC)

confident about the validity of our results but will replicate our study with other techniques and tools in the future.

Finally, we manually traced classes in design models in the source code of the studied systems. It is possible that other traceability links could have been established, which could impact the results of our study. Future work includes devising automated designs-to-source code traceability algorithms.

k) Threats to reliability validity: concern the possibility of replicating this study. Every result obtained through empirical studies is threatened by potential bias from the used data sets [27]. To mitigate these threats, we performed our study using 10 systems. In addition, we attempted to provide all the necessary details required to replicate our study. The UML Repository and the source code repositories of the 10 systems are publicly available and so is the source code of the Ptidej tool suite used to compute metrics and detect the occurrences of the anti-patterns.

l) Threats to external validity: concern our use of ten open-source systems and our conclusion based on this data set. These systems that used in our studies are available in the UML Repository. It has different sizes and belong to different domains. In the studies (2, 3) we used two open source software projects, and made our conclusion based on these two systems. Nevertheless, further validation on a larger set of systems is desirable, considering systems from different domains as well as systems from same domains.

m) Recommendations: Kitchenham et al. [28] presented preliminary guidelines for empirical research in software engineering. Following these guidelines, to assist researchers, we recommend the use of the UML Repository because it is open and contains UML models and URLs to some of their source code when available. Thus, researchers can reuse and share their results on a common base of designs and source code. We encourage researchers to join the UML Repository community and share their models.

Our study is a preliminary study on the impact anti-patterns in design models. More empirical studies are needed using more systems to refine our conclusions. We hope that others will use the UML Repository to bring new results.

We recommend that designers be wary of anti-patterns in their designs because they transfer and negatively impact change- and fault-proneness of the corresponding classes in source code. They should use tools to identify and remove anti-patterns in their designs.

VII. CONCLUSION

In this paper, we performed two studies to investigate the relation between quality of the design and the source code. In the first study, we investigated whether models produced by designers before the implementation of software systems contain anti-patterns. We also examined whether the occurrences of the anti-patterns in models translate into the source code, affecting the same classes in models and the source code. We conducted an empirical study on the prevalence of four anti-patterns: Complex Class, Large Class, Lazy Class, and Long Method, using both the architects and designers models of the ten systems (selected from the UML Repository) and the source code of these systems. Our results showed that, on average, 37% of the classes in the models that belong to anti-patterns also exist in the source code and also play roles in the same anti-patterns. Hence, we showed that anti-patterns appear very early. Designers would benefit from help to identify and control these anti-patterns as early as possible. Therefore, it would be wise for maintenance teams to detect these anti-patterns early in the design phase to save time and effort.

In the second study, we conducted two sub-experiments. In the first, we find that classes in the designs have more changes and faults than others that exist only in the source code. This provides evidence that the quality of the classes that appear in the design is important. In the second, seven types of anti-patterns detected in the design phase: (1) Complex class, (2) Large class, (3) Lazy class, (4) Blob class, (5) ClassDataShouldBePrivate, (6) RefusedParentBequest, (7) BaseClassShouldBeAbstract. We find that classes in the designs that have anti-patterns have more changes and faults in the source code.

We conclude that the quality of the design is important, and in our case we measure the quality based on anti-patterns. We

should detect anti-patterns early at design stage, and solve it to: (1) Avoid transfer it to the implementation. (2) Reduce number of changes in classes in the source code. (3) Reduce number of faults in classes in the source code.

In addition, we expect that problems in the design can propagate to other problems in the source code, which could occur in the same classes or their corresponding classes. Therefore, avoiding anti-patterns may help to avoid much problems in the source code.

Future work includes analyzing more pairs of models and their corresponding source code as well as analyzing more projects to propose prevention techniques. Because refactoring are easier at design level, we aim to propose a technique to automatically refactor anti-patterns detected in models.

ACKNOWLEDGMENT

This work has been partly funded by the Canada Research Chair on Patterns in Mixed-language Systems and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the* 37^{th} *International Conference on Software Engineering*. ACM Press, May 2015. [Online]. Available: https://dibt.unimol.it/staff/oliveto/pubs/c80.pdf

[2] Stéphane Vaucher, Foutse Khomh, Naouel Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering (WCRE)*, G. Antoniol and A. Zaidman, Eds. IEEE Computer Society Press, October 2009, pp. 145–154, 10 pages. [Online]. Available: http://www.ptidej.net/publications/documents/WCRE09b.doc.pdf

[3] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, Sept 2010, pp. 106–115.

[4] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998. [Online]. Available: www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase_theantipatterngr/103-4749445-6141457

[5] Foutse Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering (WCRE)*, G. Antoniol and A. Zaidman, Eds. IEEE Computer Society Press, October 2009, pp. 75–84, 10 pages. [Online]. Available: http://www.ptidej.net/publications/documents/WCRE09a.doc.pdf

[6] B. Venners, "Leading-edge java – a conversation with erich gamma," May–June 2005, part I–V. [Online]. Available: http://www.artima.com/lejava/articles/gammadp.html

[7] Naouel Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *Transactions on Software Engineering (TSE)*, vol. 36, no. 1, pp. 20–36, January–February 2010, 16 pages. [Online]. Available: http://www.ptidej.net/publications/documents/TSE09.doc.pdf

[8] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aimeur, "Smurf: A SVM-based incremental anti-pattern detection approach," in *Proceedings of the* 19^{th} *Working Conference on Reverse Engineering (WCRE)*, R. Oliveto and D. Poshyvanyk, Eds. IEEE Computer Society Press, October 2012, pp. 466–475, 10 pages. [Online]. Available: http://www.ptidej.net/publications/documents/WCRE12a.doc.pdf

[9] Foutse Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering (EMSE)*, vol. 17, no. 3, pp. 243–275, August 2012, 27 pages. [Online]. Available: http://www.ptidej.net/publications/documents/EMSE11b.doc.pdf

[10] E. Arisholm, L. Briand, S. Hove, and Y. Labiche, "The impact of uml documentation on software maintenance: an experimental evaluation," *Software Engineering, IEEE Transactions on*, vol. 32, no. 6, pp. 365–381, June 2006.

[11] W. Dzidek, E. Arisholm, and L. Briand, "A realistic empirical evaluation of the costs and benefits of uml in software maintenance," *Software Engineering, IEEE Transactions on*, vol. 34, no. 3, pp. 407–432, May 2008.

[12] A. M. Fernández-Sáez, M. Genero, D. Caivano, and M. R. Chaudron, "Does the level of detail of uml diagrams affect the maintainability of source code?: a family of experiments," *Empirical Software Engineering*, pp. 1–48, 2014.

[13] A. Nugroho and M. R. Chaudron, "The impact of uml modeling on defect density and defect resolution time in a proprietary system," *Empirical Softw. Engg.*, vol. 19, no. 4, pp. 926–954, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9243-2

[14] R. P. L. Buse and T. Zimmermann, "Information needs for software development analytics," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 987–996. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337343

[15] R. France, J. Bieman, and B. H. Cheng, "Repository for model driven development (remodd)," in *Models in Software Engineering*. Springer, 2007, pp. 311–317.

[16] R. France, J. Bieman, S. Mandalaparty, B. Cheng, and A. Jensen, "Repository for model driven development (remodd)," in *Software Engineering (ICSE), 2012 34th International Conference on*, June 2012, pp. 1471–1472.

[17] B. Karasneh and M. R. V. Chaudron, "Online img2uml repository: An online repository for UML models," in *Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, USA, October 1, 2013.*, 2013, pp. 61–66. [Online]. Available: http://ceur-ws.org/Vol-1078/paper8.pdf

[18] B. Karasneh and M. Chaudron, "Img2uml: A system for extracting uml models from images," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, Sept 2013, pp. 134–137.

[19] E. Architect, "2011 sparx systems pty ltd., creswick, victoria, 3363, australia," 2000.

[20] V. Paradigm, "Visual paradigm for uml," *Visual Paradigm for UML-UML tool for software application development*, 2014.

[21] U. StarUML, "modeling tool. multilingual project. version 5.0. 2.1570," 2005.

[22] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multi-layered framework for design pattern identification," *Transactions on Software Engineering (TSE)*, vol. 34, no. 5, pp. 667–684, September 2008, 18 pages. [Online]. Available: http://www.ptidej.net/publications/documents/TSE08.doc.pdf

[23] Y.-G. Guéhéneuc, H. Sahraoui, and Farouk Zaidi, "Fingerprinting design patterns," in *Proceedings of the* 11^{th} *Working Conference on Reverse Engineering (WCRE)*, E. Stroulia and A. de Lucia, Eds. IEEE Computer Society Press, November 2004, pp. 172–181, 10 pages. [Online]. Available: http://www.ptidej.net/publications/documents/WCRE04.doc.pdf

[24] D. Radjenovic, M. Hericko, R. Torkar, and A. ivkovic, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397 – 1418, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913000426

[25] J. Q. Wilson and G. L. Kelling, "Broken windows," *Atlantic monthly*, vol. 249, no. 3, pp. 29–38, 1982.

[26] R. K. Yin, *Case study research: Design and methods*. Sage publications, 2013.

[27] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, January 2007. [Online]. Available: http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=4027141

[28] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 721–734, Aug 2002.