Circular dependencies and change-proneness: An empirical study

Tosin Daniel Oyetoyan, Jens Dietrich, Jean-Rémy Falleri, Kamil Jezek

Circular Dependencies and Change-Proneness: An Empirical Study

Conference Paper · March 2015

DOI: 10.1109/SANER.2015.7081834

CITATIONS
7

READS
277

4 authors, including:

Tosin Daniel Oyetoyan
Høgskulen på Vestlandet
27 PUBLICATIONS **111** CITATIONS
SEE PROFILE

Jens Dietrich
Victoria University of Wellington
72 PUBLICATIONS **952** CITATIONS
SEE PROFILE

Kamil Jezek
University of West Bohemia
31 PUBLICATIONS **137** CITATIONS
SEE PROFILE

Circular Dependencies and Change-Proneness: An Empirical Study

Tosin Daniel Oyetoyan
Department of Computer and Information Systems
Norwegian University of Science and Technology
Trondheim, Norway
tosindo@idi.ntnu.no

Jens Dietrich
School of Engineering and Advanced Technology
Massey University
Palmerston North, New Zealand
J.B.Dietrich@massey.ac.nz

Jean-Rémy Falleri
LaBRI, University of Bordeaux
Bordeaux, France
jr.falleri@gmail.com

Kamil Jezek
Department of Computer Science and Engineering
University of West Bohemia
Pilsen, Czech Republic
kjezek@kiv.zcu.cz

***Abstract*—Advice that circular dependencies between programming artefacts should be avoided goes back to the earliest work on software design, and is well-established and rarely questioned. However, empirical studies have shown that real-world (Java) programs are riddled with circular dependencies between artefacts on different levels of abstraction and aggregation. It has been suggested that additional heuristics could be used to distinguish between bad and harmless cycles, for instances by relating them to the hierarchical structure of the packages within a program, or to violations of additional design principles.**

In this study, we try to explore this question further by analysing the relationship between different kinds of circular dependencies between Java classes, and their change frequency.

We find that (1) the presence of cycles can have a significant impact on the change proneness of the classes near these cycles and (2) neither subtype knowledge nor the location of the cycle within the package containment tree are suitable criteria to distinguish between critical and harmless cycles.

***Keywords*—Circular dependency, maintainability, patterns**

I. Introduction

Avoiding circular dependencies between software artefacts is a classic software design principle that can be traced back to Parnas' advise that modules should be organised in a hierarchy with respect to dependency relationships, thereby keeping dependencies "loop free" [31]. In the context of modern object-oriented languages, this is known as the Acyclic Dependencies Principle (ADP): The dependencies between packages must not form cycles [24].

The justification for this principle has often been related to maintenance. For instance, Parnas pointed out that it is undesirable to have systems where "nothing runs unless everything runs" [31]. Later work has related this to testing, where the presence of cycles prevents unit testing and requires the use of expensive methods such as the use of stubs [29].

Empirical studies on a large set of real-world Java programs have shown that these programs are riddled with circular dependencies [25], [8]. This applies to both simple circular dependencies [25] as well as to more sophisticated antipatterns like subtype knowledge [36], [8].

This seems to indicate that not all cycles are as critical for the quality of software as previously thought, and that the notion of cyclic dependencies in software must be re-evaluated. One possible approach taken by Falleri et al [11] is to distinguish between "bad" and "harmless" cycles based on the topology of dependency graph. In a nutshell, the authors argue that cycles forming in branches of the package containment tree evolve when packages grow, and are harmless, while cycles that span across the entire package containment tree are undesirable. Mutawa et al [1] studied the topology of cycles on a large set of real-world Java programs and found that (1) most cycles do form in branches of the package containment tree (and are therefore not critical according to [11]), and (2) that the parent packages are the "hubs" within these circular structures – indicating that cycles grow around these parent packages. This offers an explanation of why circular dependencies are common, and do not necessarily compromise the quality of programs.

However, the question how cycles in general and certain types of cycles in particular relate to the maintainability of programs remains open. In this paper, we present a study that investigates this issue for Java programs. We use the qualitas corpus [40] data set in our study. Maintainability is difficult to measure directly. According to IEEE 610.12, maintenance is "the **process of modifying** a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment" [35]. Following this definition, we use change (frequency of modifications) to approximate maintainability, and therefore set out to answer the following question: *Is there a co-relation between the fact that a Java class is in a (certain kind of) cycle, and the change frequency of this class*. In other terms, do cycles incur a maintenance penalty that can be measured? We will investigate both general circular dependencies between classes and special kinds of circular dependencies that have been portrayed as particularly undesirable in previous research.

This study extends our previous work on dependency

cycles where we have investigated the relationship between cycles and defects [30]. The result of this study revealed that classes within and near cycles account for the most defects in programs. This study did not investigate particular types of cycles and their relationship with change proneness. It used a smaller data set, and did not study the classes directly, but mined the comments in the issue tracking and subversion systems instead.

The rest of this paper is organised as follows: we first present the core concepts used in this paper in Section II. We then discuss related work in Section III. We describe our methodology in Section IV. We present our results in Section V and discuss them in Section VI. Finally we conclude and present the future work in Section VII.

II. Background

A. Cycles and Dependency Graphs

The notion of cyclic dependency corresponds to strongly connected components (SCCs) in dependency graphs. SCCs can be effectively computed with Tarjan's algorithm in linear time [38].

A dependency graph is a simple model representing software artefacts and their relationships. Such a graph can be built on several levels of abstraction and aggregation. For instance, in the case of Java programs, we can consider methods and fields and their invoke and access relationships, classes and interfaces and their uses, extends and implements relationships, packages and their dependencies, and containers (jar files) and their dependencies. Low-level cycles have been associated with potential problems for comprehension, testing, and maintenance [3], [4]. However, to the best of our knowledge no empirical studies on larger sets of real-world programs exist to support this claim, and at least some of the cycles are created by widely-used programming techniques like recursion.

Higher-level dependency graphs are typically obtained from lower-level graphs by means of aggregation. For instance, a package-level dependency graph is built from the dependency graph of the classes contained in this packages. Cyclic dependencies between classes in different packages induce cyclic dependencies in the package graph. Therefore, we focus our attention on SCCs in the class graph. The vertices in this graph represent the classes of a Java program, while the edges represent the relationships between these vertices. Classes here refers to compiled classes, and also include other Java types like annotations, interfaces and enums. Edges are labelled with either *uses*, *extends* or *implements*. The *extends* and *implements* labels are used according to the meaning of the respective keywords defined in the Java Language Specification [15], *uses* covers all other dependencies. We also use the label *inherits* defined as the union of *extends* and *implements*.

Several empirical studies on real-world programs suggest that the number of SCCs found in both the class-level and package-level dependency graphs is large [25], [8]. The fact that many of these systems are regarded as functional and widely used suggests that not all cycles are as detrimental to the quality of systems as previously thought. This seems to indicate that it is not sufficient to only study general cycles. Instead, certain types of cycles must be studied as well in order to distinguish between critical and harmless cycles.

B. Subtype Knowledge

Subtype knowledge (STK) is an "antipattern" first studied by Riel [36]. An instance of STK is basically a cycle that has at least one *extends* or *implements* edge, and a back-reference path connecting the target of this edge with its source. Because the Java compiler (as well as most other compilers) enforces that there are no cycles in the supertype (*inherits*) graph, this path must contain at least one *uses* edge. Situations producing inheritance cycles still exist when classes are compiled separately, but they are rare and can be caught by the Java Virtual Machine by means of static analysis during linking.

The intention behind this pattern is that in a well designed program, abstraction and implementation artefacts are separated, and implementation artefacts depend on abstractions, but not vice versa. This is also known as the dependency inversion principle (DIP) [22]. STK cycles directly violate this principle. Surprisingly, STK cycles are still common in real-world programs [8].

Figure 1 depicts a STK cycle found in the Java Runtime Environment, version 1.7.0. This is a class-level cycle, but it also induces a package level cycle between `java.awt` and `javax.swing`. The documentation of `LegacyGlueFocusTraversalPolicy` indicates that this is a `FocusTraversalPolicy` implementation that provides support for legacy applications. Yet, every other implementation of `FocusTraversalPolicy` depends on it as there is a dependency from the abstract type to this particular implementation. This is clearly an undesirable constraint for a modular design.

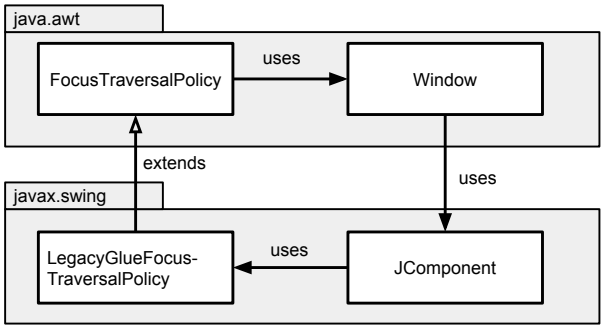

Fig. 1. A STK cycle in the Java Runtime Environment, version 1.7.0

Note that not all STK instances are equally critical. An example is discussed below in section II-D where a STK is a side-effect of using the visitor design pattern. This might still have negative consequences, however, they are outweighed by the benefits of using the design pattern.

C. Cycles and the Package Containment Tree

One possibility to distinguish between critical and harmless cycles is to consider their location within the package containment tree (PCT) [11]. The PCT of a Java program is formed by the hierarchical structure of package names. The Java language specification stipulates that "The hierarchical naming structure for packages is intended to be convenient for organizing related packages in a conventional manner, but has no significance in itself ... " [15, ch 7.1]. However, developers seem

to use some sub-package semantics when organising code. For instance, the package `javax.swing` has circular dependencies with its "child packages" `javax.swing.tree` and `javax.swing.table`. It appears that these cycles forming in branches of the PCT are the result of splitting large packages to facilitate maintainability, but the respective packages retain a high level of cohesion. AWT features a similar structure. However, the core Java interface libraries also provide an example of a critical cyclic dependency spanning *across branches* of the PCT: AWT and Swing mutually depend on each other. Figure 1 also shows this. The critical dependency is caused by references to `javax.swing.JComponent` in several AWT classes, including `java.awt.Window` and `java.awt.Component`. On the other hand, `javax.swing.JComponent` is a subclass of `java.awt.Component`. This design flaw had a significant impact on early versions of the Java platform, and there is evidence that it can be removed without impacting on the functionality of the respective libraries. This is discussed in more detail in [9].

D. Inadvertent Cycles

There are situations where cycles are a direct result of the features and limitations of technologies and methods used in projects. The most simple example in this category are the cycles formed between non-static nested classes and their outer classes in Java byte code. In particular, the compiler generates access fields to reach inner class from outer one and vice-versa.

A more complex case that is common originates from the use of certain design patterns that induce cycles. An example is the use of Visitor, one of the classic gang of four patterns [14]. The pattern consists of abstract and concrete visitors, and abstract and concrete visited "elements". The visitors reference all concrete element types as parameters in the (overloaded) `visit` methods, while the element types (both abstract and concrete) use the abstract visitor type as parameter type in the `accept` methods. Visitor is a very popular pattern, in particular in programs that use hierarchical data structures such as parsers for domain specific languages (DSLs). Such an example is depicted in figure 2. The cycle is even an instance of STK, caused by the inherits relationship between the concrete elements (such as `ASTIdentifier`) and the abstract element (`Node`). Note that the number of concrete elements is typically large, in this example, there are 33 such classes each representing a particular AST node type. This can result in large SCCs.

These cycles can hardly be interpreted as signs of bad design, on the contrary, the use of Visitor is widely seen as good design as it allows developers to "plug-in" functionality into complex object structures. This is also a case of choosing a particular design to overcome limitations of the programming language, in this case the lack of support for multiple dispatch in Java [26]. Acyclic versions of Visitor have been proposed [23]. However, acyclic visitors are even more complex than visitors as additional abstract visitor types are required, and it appears that they are not widely used.

In the velocity example used in figure 2, the Visitor has been manually implemented. However, in many cases parser code is generated by parser generators from abstract grammar

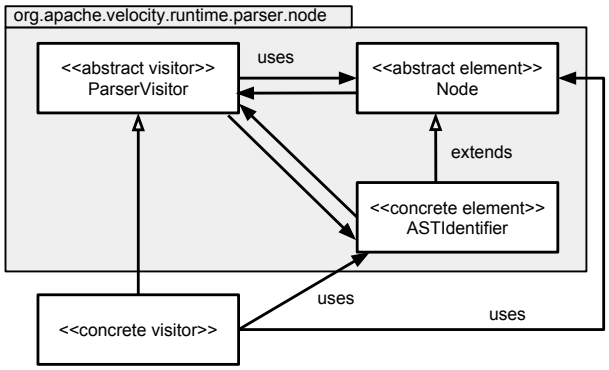

Fig. 2. A cycle caused by the use of the Visitor pattern in Apache Velocity, version 1.6.2

specifications. This is becoming more and more common with the availability of good tools (such as ANTLR), and the popularity of DSLs. Code with generated cycles can have interesting change characteristics, for instance, if the code is regenerated during each iteration as part of automated builds.

III. Related Work

Several authors have investigated the relationship between anti-patterns and the change-proneness of software artefacts.

Khomh et al. [20] examined classes involved in anti-patterns and code smells and their change and fault proneness. The study investigated four systems and thirteen anti-patterns. The claims from this study are that classes participating in anti-patterns are more change- and fault-prone than others and that structural changes affect more classes with anti-patterns than others. Romano et al. [37] investigated the impact of anti-patterns on change-proneness using change data from source code analysis. The results of this study is consistent with [20]. In addition, they showed that certain anti-patterns are prone to certain types of changes such as API changes. Olbrich et al. [28] performed a study on two open source applications to study the impact of code smells. Their results show that different phases could be identified during the evolution of code smells and in particular, components infected with code smells display a higher change frequency than others. Fontana et al. [13] investigated the correlations between different smells and antipatterns.

In our study, we have investigated one particular antipattern on the structural/architectural level, and this is different from these studies.

On the other hand, while anti-patterns are claimed to be poor design choices, design patterns are recurring solutions to design problems. A plethora of studies have also investigated the relationships between design patterns and class change-proneness. Bieman et al. [2] investigated the impact of design patterns on the change proneness of classes by using five systems, four small ones and one large system. They have mined the change data from a configuration management system. They concluded that classes participating in design patterns are rather more change-prone. A recent study on mining repository [16], however showed that multiple tangled code changes could result into an incorrect classification of change/fault data.

Di Penta et al. [6] investigated whether certain design pattern roles are more change-prone in general, and whether certain roles are prone to particular types of changes. Their results confirmed that many design pattern roles do undergo changes within the pattern. Vokac [41] analyzed the defect rates of classes that participated in selected design patterns of a large commercial product. The study concluded that Observer and Singleton patterns are correlated with large code structures and can thus serve as indicators for special attention. On the other hand, Factory pattern instances tend to have lower defect counts. Prechelt et al. [33] reported a controlled experiments that showed Observer and Decorator patterns to result in less maintenance time while the results for Visitor pattern were inconclusive. Vokac et al. [42] replicated the experiment by [33]. Their results confirmed the previous results that Observer, Decorator and Abstract Factory patterns favour ease of maintenance. However, the Visitor and Composite patterns had strongly negative results on maintenance. On the contrary, Jeanmart et al. [19] reported a positive relationship between the use of Visitor pattern and maintenance efforts.

In our study we investigate the impact of one particular anti-pattern on maintenance using change data as a proxy. We do not focus on the impact of design patterns in general, however, we discuss the impact of one particular pattern, Visitor, as it results in dependency cycles. To the best of our knowledge, there is no study that has systematically explored the relationship between change proneness and cycles. The key papers of research on cycles in dependency graphs are discussed in the previous section.

IV. Methodology

A. Data Set

We have conducted the study using the Qualitas Corpus dataset [40]. This is a curated dataset of open source real world systems that has been widely used in empirical studies on software quality issues. Using a standard dataset facilitates the replication of our study. The Qualitas Corpus version 20120401 contains 111 programs. The full release (20120401f) combines the standard release (20120401r) with the evolution release (20120401e) which contains multiple versions of programs, a total of 661 versions. We chose programs that had at least 10 versions in the corpus in order to observe evolution over a longer period of time. This means that the following programs were included in this study: ant (21 versions), antlr (20), argouml (16), freecol (28), freemind (16), hibernate (100), jgraph (39), jmeter (20), jung (23), junit (23), lucene (28) and weka (55).

The scripts we have used and developed for this study can be found here: https://bitbucket.org/ootos/scc-project. Table I provides some statistics of the dataset used. A total of twelve (12) systems are analyzed consisting of 389 versions.

B. Experiment Setup

The experiments consist of the following steps to extract, process and analyse data:

1) Graph Extraction: Dependency data is extracted from Java byte code with scripts using the Apache BCEL library [5]. Since the units of maintenance are compilation units, we merge nested classes with their outer, top-level classes. The dependencies of nested classes are aggregated to their top-level classes. These aggregated classes form the vertices of the dependency graph. Extends and inherits edges are created when the respective constructs are encountered in byte code, all other occurrences of a class in the byte code of another class result in the creation of a uses edge.

2) Graph Pre-processing: We sanitise the dependency graphs by removing test classes and generated code. Test cases are removed as tests (1) tend to be more stable[1] due to the fact that in many projects they are used as specification artefacts as suggested by the test-driven development (TDD) methodology, (2) it is unusual to have cross-references between tests, and references from core functional code to tests, making it very unlikely to encounter tests that participate in cycles. We therefore believe that including tests would have skewed the results. We have also tried to remove generated code. In particular, parser APIs generated by ANTLR and similar parser generators are removed. Even minor changes in grammar definitions can produce a large amount of changes as many generated artefacts are regenerated and renamed. But this has nothing to do with whether these artefacts are in cycles or not, this is only caused by the fact that they are generated together. On the other hand, the process of regenerating these classes often does not incur any maintenance effort, as code generation is completely automated. Note that generated parser APIs often use the Visitor pattern and therefore often contain SCCs, as discussed in section II-D.

We use simple naming pattern filters to remove tests (looking for the "Test" token in class names). To remove generated code, we have manually inspected the (ANT, Maven and Gradle) build scripts of the projects for references to code generators and the target packages names used by them. We found two projects where parser generators are used: (1) hibernate uses ANTLR and JAXB, and we excluded the following packages: `org.hibernate.hql.internal.antlr.*`, `org.hibernate.sql.ordering.antlr` and `org.hibernate.internal.jaxb.*`. (2) Weka uses JFlex and CUP, and we excluded the following packages: `weka.core.mathematicalexpression`, `weka.filters.unsupervised.instance.subsetbyexpression` and `weka.core.json`.

3) SCC Detection and Classification: Once the dependency graph is built, we use an implementation of Tarjan's algorithm [38] to detect the strongly connected components (SCCs). The detected $SCCs$ are classified in categories (STK vs non-STK, Visitor vs non-Visitor), and associated with their PCT diameter relative to the diameter of the entire dependency graph. STK is approximated by the presence of *inherits* edges in a SCC as discussed in section II-B. Visitor instances are detected based on naming patterns.

4) SCC Membership: Finally, we establish the association of a class with a cycle. The most obvious option is to look for whether the vertex representing the class is an element of the respective SCC. However, we are also interested in assessing the impact SCCs have on their direct neighbourhood,

[1] In the context of this study, stability relates to whether a class is frequently changed or not

i.e., classes that are not in a cycle, but depend directly on a class within the cycle (*in-neighbours*), or a class in a cycle that directly depends on such a class (*out-neighbours*). A *neighbour* is either an in-neighbour or an out-neighbour.

5) Extracting Change Data: We use the change data set also used in [7]. This data contains fine-grained, per-class information of change classified by a change category. Details on how this is done can be found in this paper.

C. Research Questions

The general problem we are interested in is the correlation between the presence of certain types of cycles in programs, and the maintainability of these program measured in terms of change frequency, as discussed above. We break this down into the following research questions:

Firstly, we want to investigate whether a class within or near a cycle is more prone to change than a class outside a cycle. Our hypothesis is that the structural complexity associated with cycles could make it easier for change to spread to other classes within the cycle, and classes either directly referencing classes in the cycle, or being directly referenced by classes from within the cycle.

RQ1. Are classes within or near cycles more prone to change than other classes?

Secondly, we want to investigate whether classes that are in or near STK cycles are more prone to change than classes in non-STK cycles as these cycles violate a second principle of object-oriented design (the dependency inversion principle (DIP) [22]). This leads to the following question:

RQ2. Are classes in or near cycles with STK more change prone than classes in cycles without STK?

Finally, we want to investigate whether the $PCT - diameter$ of a cycle is correlated with the change proneness of the classes within this cycle, following the argument made by Falleri et al that PCT-local cycles are less critical than cycles that span across different branches of the PCT [11]. We thus hypothesize that cycles with a large PCT-diameter would be more change-prone than those with a smaller PCT-diameter.

RQ3. Is there a correlation between the $PCT - diameter$ of a cycle and the change frequency of the classes in or near this cycle?

D. Metrics and Measurement

For statistical analysis, we compute data series with data points for each version. The values are change probabilities, and each data series corresponds to a set of classes resulting from a classification, such as whether a class is in or near a particular type of SCC.

1) Computing the change probability of a set of classes: Given a program P, let C be the set of classes in P, and V be the set of versions of P such that for each version $v \in V$ a successor version $succ(v)$ exists. For a given set of classes $S \subseteq C$ and a version $v \in V$ we use $changed(S, v)$ to denote the set of classes in S that have changed from v to $succ(v)$. We then define the change probability of a class in S as a function $p_{change} : 2^C \times V \rightarrow [0, 1]$ defined as:

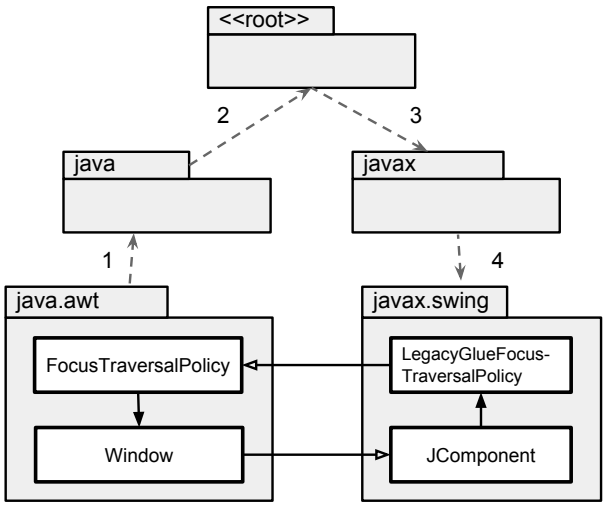

Fig. 3. The PCT-diameter of an SCC

$$p_{change}(S, v) = \frac{|changed(S,v)|}{|S|}.$$

2) Measuring the PCT diameter of an SCC: Given a set of packages P and the package containment tree (PCT) they form (see Section II-C), we compute the PCT-diameter of a set of classes as the diameter of the packages of these classes in the PCT. The PCT-diameter is computed by first computing the shortest distance between each pair of packages in the PCT and then finding the longest of the computed shortest distances. This is referred as the longest shortest path in network analysis [43] We can normalise this value to $[0, 1]$ by dividing this number by the diameter of the set of all packages within the program.

For instance, consider the example depicted in figure 3. We have discussed the same cycle earlier. The longest shortest path between the respective packages has a length of 4 (`java.awt` → `java` → `<root>` → `javax` → `javax.swing`). Note that the PCT shown in this figure is incomplete, there are several core Java packages with 5 tokens, such as `javax.swing.text.html.parser`. Therefore, the diameter of the entire program is 10, and the PCT-diameter of the SCC in figure 3 is 0.4 (4/10). The normalised PCT computation just described defines a PCT function $pct : 2^C \times V \rightarrow [0, 1]$.

3) Detecting SCCs with STK: Finding instances of STK is computationally expensive as the NP-complete subgraph isomorphism problem must be solved. However, STK can be easily approximated by computing SCCs that contain at least one *inherits* edge. The drawback of this approach is that these SCCs may contain both STK and non-STK sub-cycles.

This defines a STK membership function $stk : 2^C \times V \rightarrow \{false, true\}$, where $stk(SCC_i, v) = true$ iff SCC_i is a STK in version v.

4) Measuring Nearness of a Cycle: We also want to find out whether classes that are in the neighbourhood of a cycle of a certain type are penalized by increased change-proneness. We differentiate between outward nearness (fan-outs of the classes in cycles) and inward nearness (fan-ins of the classes in cycles). In many cases, multiple cycles can have the same

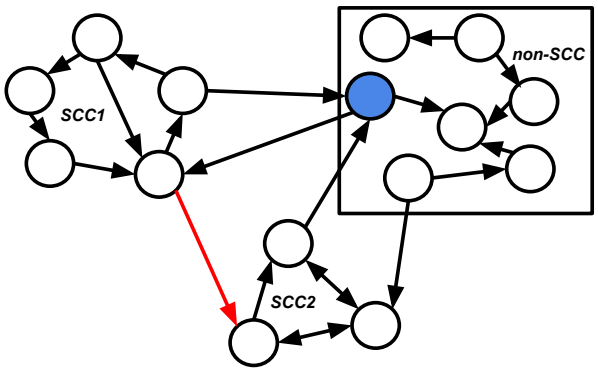

Fig. 4. Neighborhood to an SCC

TABLE II. AVERAGE PERCENTAGE OF CLASSES IN $SCCs$

Systems	% of Classes		
	SCCs	In-Neighbor(incl)	In/Out-Neighbor(incl)
ant	35.2%	76.3%	83.6%
antlr	34.0%	56.9%	75.9%
argouml	31.8%	55.7%	74.5%
freecol	80.7%	82.9%	92.9%
freemind	55.3%	80.3%	92.6%
hibernate	62.8%	76.1%	93.6%
jgraph	77.0%	79.5%	98.0%
jmeter	23.0%	73.3%	83.7%
jung	10.3%	75.0%	80.4%
junit	19.7%	46.1%	64.4%
lucene	29.5%	51.3%	73.5%
weka	13.4%	66.9%	77.9%

neighbours. For instance, figure 4 shows an example where two cycles $scc1$ and $scc2$ share the same outward neighbour. In order to avoid assigning a class to multiple cycles, we use the following set of rules when a class is near multiple cycles:

1) If the class changes, prioritize cycles with change. If there are multiple cycles that change, pick one randomly.
2) If the class does not change, prioritize cycles without change. If there are multiple cycles that change, pick one randomly.
3) Otherwise randomly select a cycle.

E. Statistical analysis

1) Analysis Method: The input data for the statistical analysis are provided by the three functions p_{change}, stk and pct that associate SCCs version pairs with information representing change probability, STK classifications and PCT values.

We want to investigate (1) the change proneness of SCCs against non-SCCs, (2) the change proneness of $SCCs$ with STK over $SCCs$ without STK and (3) whether the PCT diameters of $SCCs$ are correlated with change proneness.

a) Analyzing Change Proneness of SCCs vs. Non SCCs: We analyse two data series for the two sets of classes: the classes in SCCs, and the classes not in SCCs. The hypothesis here is that classes in SCC are more change-prone and they propagate change more to their neighbourhoods because of their structural complexity. It is easy to expand this investigation to include neighbourhoods of an SCC, by also considering neighbours (in-neighbours out-neighbours) as elements of SCCs as described above.

b) Analyzing Change Proneness of STK vs Non-STK : Here we analyse two data series: the classes within STKs, and the classes in non-STK SCCs. Note that we do not directly compare STK instances with non-SCCs, however, this relationship can be inferred by combining the results of this and the previous experiment.

c) Analysing the Correlation between PCT Diameter and Change Proneness: To answer this question, we use a slightly different method. The input data are not just two data series, but consist of two matrices where we map pairs consisting of versions and individual SCCs to a change probability using the formula defined above, and to the PCT diameter value, respectively.

2) Testing of the Hypotheses: We have employed two different statistical analysis methods to test our hypotheses. The choice of either one depends on the measurement type of the variables under investigation. To analyse the correlation between two data series (RQ1 and RQ2), we used a non-parametric test. To test the hypotheses in this category, the data is first tested for normality using the Shapiro test. It turned out that each dataset deviates strongly from normality. Subsequently, we use a non-parametric test (Wilcoxon rank-sum)[12] for analysis.

For interval variables used in the experiment for RQ3, we have used Pearson and Spearman correlation.

3) Measuring interactions among experimental factors: It is the goal to also understand if there are interactions among the two factors being investigated in this study. We suspect that classes with high PCT-diameter could also be prone to STK anti-pattern. It is thus appropriate to treat the two factors as a competing treatments and use one factor as a blocking factor in the experiment [12]. A nested design is chosen where the factor STK is selected as a blocking factor, since it is nominal in its scale whereas PCT-Diameter is interval. Next, the $sccs$ are grouped into hasSTK -$True$ or $False$ groups and a statistical analysis is performed between PCT-Diameter and change-probability (dependent variable) in each group.

V. RESULTS

A. System Properties

Table I shows the average values for several system properties while Table II reports the (average) percentage of classes in and near cycles. Averages are computed over all versions of the respective program in the data set. The distribution of classes within SCC range from 10.3% to 80.7%. For some of the systems, a surprisingly high number of classes is within cycles, including freecol (80.7%), jgraph (77%), hibernate (62.8%) and freemind (55.3%). Two systems, jgraph and freecol, have relatively large PCT-diameter values. Freemind has the largest percentage of changed classes (53.6%) as shown in p_{change} column, while the rest of the systems have change probabilities between 10.8% (jung) to 35.3% (freecol).

TABLE I. SUMMARY OF SYSTEM PROPERTIES, AGGREGATED VALUES ARE OBTAINED BY AGGREGATING OVER THE VALUES FOR EACH SCC AND EACH VERSION v

Systems	Versions	Num of classes	PCT-diameter		Size of STK-SCCs		Size of Non-STK SCCs		Size of SCCs		Size of Non-SCCs		$avg(p_{change}(C,v))$
		Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	
ant	21	162.7	1	0.17	5	3.79	6	3.21	205	113.95	357	211.47	0.285
antlr	20	120.9	1	0.05	6	3.22	10	5.06	166	81.89	328	159.83	0.184
argouml	16	891.6	1	0.09	4	2.92	17	11.46	855	568.92	1705	1214.31	0.325
freecol	28	189.1	1	0.67	2	1.04	2	0.46	473	305.12	90	73.12	0.353
freemind	16	45.1	1	0.33	3	1.14	9	1.79	162	49.86	333	40.29	0.536
hibernate	100	513.2	1	0.19	21	5.06	11	4.12	1406	653.65	1191	372.66	0.160
jgraph	39	25.3	1	1	1	1.00	0	0.00	39	39.00	14	11.67	0.107
jmeter	20	286.5	0.71	0.21	4	2.79	8	6.21	188	132.16	576	440.89	0.283
jung	23	152.7	0.6	0.07	4	1.91	8	6.95	48	31.82	415	273.55	0.108
junit	23	39.6	1	0.09	3	0.76	8	4.62	27	15.52	152	63.62	0.233
lucene	28	163.4	0.75	0.07	6	2.96	9	6.00	143	95.85	339	230.92	0.211
weka	55	325.3	0.875	0.05	13	5.23	26	14.53	263	86.83	969	563.87	0.139

TABLE III. WILCOXON TEST: P-VALUES OF SCCS VS. NON-SCCS ($\alpha = 0.05$)

Systems	SCCs	+ In-Neighbor	+ In/Out-Neighbor
ant	0.5	**0.035***	**0.037***
antlr	0.665	0.233	0.238
argouml	0.147	0.075	0.178
freecol	**0.004***	**0.001***	**5.27E-05***
freemind	0.198	**0.009***	**8.81E-04***
hibernate	0.052	**4.70E-05***	**0.021***
jgraph	**3.39E-10***	**9.18E-10***	**2.24E-11***
jung	0.742	**0.038***	**0.041***
junit	0.435	**0.003***	**0.010***
lucene	0.142	0.108	0.078
weka	0.511	**0.005***	**0.005***
jmeter	0.420	**0.007***	**0.022***

B. RQ1 Are classes within or near cycles more prone to change than other classes?

The results for RQ1 are presented in Table III. In column 2, the significance test results for classes within SCC against those outside SCC are listed. While columns 3 and 4 show the results when we investigated the neighborhood of the SCCs. Only two systems (freecol and jgraph) have significant change proneness for the SCC group. However, when we considered the SCC direct neighbourhood, 75% of the systems showed significant change proneness. As shown in the results, the change frequencies of the classes increase as the size of the neighbourhood expands. This is not surprising giving that the size of the class set increases as shown in Table II. However, what is surprising is the big impact of SCCs on their neighbourhood. Investigation of the actual changes revealed that in many cases, SCCs and their direct (in-) neighbours account for more than 90 % of the total change. For instance, Ant has the average of 76.3% classes in SCCs and its direct in-neighbours, but these classes account for 94% of the total change volume. We can therefore confirm the hypothesis that the presence of SCCs could have a significant impact on the stability of the classes near those SCCs (Table VII column 3).

This may indicate a significant increase in maintenance costs, in particular as many test cases would be required to achieve sufficient coverage of the many unstable classes in the neighbourhood of cycles.

C. RQ2:Are classes in or near cycles with STK more change prone than classes in cycles without STK?

Table IV presents the results of testing this hypothesis. Column 2 of the table presents the p-values of testing $SCCs$ with STK against $SCCs$ without STK. The 3rd column presents the results when the *in-neighbours* are included in the SCC graph and the 4th column presents the results when both *in-neighbours* and *out-neighbours* are included in the SCC graph.

Out of the 12 systems we have studied, only 3 systems have $SCCs$ with STK that show significant change proneness over $SCCs$ without STK (see Table VII for summary of the results of the hypothesis).

Hibernate presents an interesting case because we detected instances of the Visitor pattern in many of its cycles. The Visitor cycles all have the STK property and the results show that in hibernate the STK cycles are more change prone than non STK cycles. To understand the role of cycles with Visitor pattern in this category, we removed the Visitor SCCs and observed that the mean values of the change probability increased from 17.9% to 19.6%. That means that the Visitor SCCs are relatively stable and as a result, removing them produces an increased change ratio. For us, this is an interesting result in the sense that, although using the Visitor pattern produces instances of an "anti-pattern" in the sense that it violates certain object-oriented design principle, nevertheless, it is stable.

A study of trade-offs between design patterns and the anti-patterns is an interesting topic for future studies.

D. RQ3:Is there a correlation between the $PCT - diameter$ of a cycle and the change frequency of the classes in or near this cycle?

The results of testing this hypothesis is presented in table V. All values in asterisks have a correlation of 0.5 or greater and are significant at $\alpha = 0.05$. We report both the Pearson and Spearman correlation results. Only one (freecol) of the systems has a fair correlation between the PCT-diameter and the change probability. As earlier reported in Table I, freecol has a very large relative PCT-diameter. We have no result for jgraph because it only contains one SCC and as a result, one data point. We detect no consistent pattern in the relationship between the PCT-diameter of class cycles and their change

TABLE IV. WILCOXON TEST: P-VALUES OF CHANGE PRONENESS OF STK-SCCS VS. NON-STK SCCS ($\alpha = 0.05$)

Systems	SCC	+ in-neighbor	+ in/out-neighbor
ant	**0.009***	**0.013***	**0.008***
antlr	0.550	0.210	0.196
argouml	0.171	0.185	0.229
freecol	**9.08E-11***	**5.45E-11***	**4.80E-11***
freemind	0.224	0.111	0.080
hibernate	**8.68E-08***	**1.38E-08***	**2.44E-09***
jgraph	-	-	-
jung	0.627	0.837	0.843
junit	0.994	0.996	0.992
lucene	0.374	0.354	0.371
weka	0.733	0.304	0.247
jmeter	0.648	0.453	0.121

TABLE V. CORRELATION TEST BETWEEN PCT-DIAMETER AND CHANGE-PROBABILITY

	SCC groups		+ in-neighbor		+ in/out-neighbor	
Systems	Pearson	Spearman	Pearson	Spearman	Pearson	Spearman
ant	0.01	0.28	0.04	0.20	0.03	0.20
antlr	-0.02	0.16	-0.16	-0.08	-0.09	-0.01
argouml	0.20	0.22	0.13	0.11	0.24	0.28
freecol	0.46	**0.64***	**0.54***	**0.78***	**0.50***	**0.69***
freemind	-0.08	-0.10	0.04	-0.08	0.10	0.04
hibernate	0.21	0.48	0.19	0.44	0.26	0.49
jgraph	-	-	-	-	-	-
jung	-0.04	-0.01	0.00	0.29	0.00	0.32
junit	0.07	0.00	0.24	0.29	0.22	0.30
lucene	-0.02	0.18	0.08	0.21	0.07	0.17
weka	0.08	0.19	0.08	0.21	0.11	0.22
jmeter	-0.02	0.08	0.06	0.17	0.25	0.32

proneness (see Table VII). This result is also surprising as we expected that cycles spanning across branches of the PCT would be more prone to change.

E. Interaction between STK and PCT-diameter

The results in table VI shows the correlation between PCT-diameter and change when grouped in the STK category

TABLE VI. CORRELATION TEST BETWEEN PCT-DIAMETER AND CHANGE-PROBABILITY BLOCKED BY STK/NON-STK

Systems	STK		Non-STK	
	Pearson	Spearman	Pearson	Spearman
ant	-0.05	0.24	-	-
antlr	-0.09	0.17	0.04	0.09
argouml	0.22	0.31	0.12	0.07
freecol	-0.23	-0.23	-	-
freemind	**0.60***	**0.56***	-	-
hibernate	0.26	**0.61***	0.12	0.02
jgraph	-	-	0.00	0.00
jung	0.10	0.18	-0.16	-0.12
junit	0.33	0.40	-0.05	-0.04
lucene	-0.09	0.18	-	-
weka	0.07	0.17	0.13	0.20
jmeter	0.08	0.18	0.05	0.05

TABLE VII. SUMMARY OF HYPOTHESES TEST: Y DENOTES H_0 IS REJECTED

	RQ1		**RQ2**		**RQ3**	
Systems	in-SCC	in/near SCC	in-SCC	in/near SCC	in-SCC	in/near SCC
ant	N	**Y**	**Y**	**Y**	N	N
antlr	N	N	N	N	N	N
argouml	N	N	N	N	N	N
freecol	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**
freemind	N	**Y**	N	N	N	N
hibernate	N	**Y**	**Y**	**Y**	N	N
jgraph	**Y**	**Y**	-	-	N	N
jung	N	**Y**	N	N	N	N
junit	N	**Y**	N	N	N	N
lucene	N	N	N	N	N	N
weka	N	**Y**	N	N	N	N
jmeter	N	**Y**	N	N	N	N

and non STK category. The STK category is represented in columns 2 and 3, while the non-STK category is represented in columns 4 and 5. The results indicate that there are just two systems (freemind and hibernate) with fair correlation (see table VI). This result is different from the correlation results in Table V that reports only freecol with a relatively high and significant correlation. We therefore conclude that there is no relationship between the STK property of a cycle and the PCT-diameter of the cycle in this dataset.

VI. DISCUSSION

A. Cycles and the Shape of Java Programs

Overall, the results are somehow surprising, and we do not have an ultimate explanation for all the findings. However, the results seem to be consistent with some other recent research on the shape of software. Several authors have studied the networks formed by software artefacts and their relationships and found that they are scale-free, and have a heavy tail distribution with a very few nodes with high connectivity [44], [17], [18], [32].

A commonly used model to explain how scale-free networks come to exist is preferential attachment [34] – in a nutshell, this model stipulates that nodes that are added to the network have a higher probability to link to nodes with an already high degree. In particular, in the case of software that would mean that there are classes with a high in-degree based on their popularity (because they provide useful utilities, or because they are widely known by developers), and the in-degree of these classes increases further as new classes are added to the program that use these utilities. On the other hand, classes with a lot of incoming dependencies have a high responsibility, and therefore tend to be more stable. It has been demonstrated that such a model can explain the network topology found in Java programs [39]. Conversely, this model suggests that high coupling is unavoidable [39]. This is in a way similar to the finding we made here: we found evidence that software evolution follows a pattern that leads to properties that are traditionally regarded as indicators of a bad design.

The results we obtained could therefore be explained by a model where cycles form in the heavy tail of the distribution. In particular, this would explain the results for RQ1: classes in cycles are relatively stable, but not the classes that reference the cycles (we called them "in-neighbours"). This could also offer an explanation for RQ2: developers may abstract from classes

providing useful utilities, but eventually these abstractions themselves reference these utilities as they are useful, for instance, in order to provide defaults for certain services. An example where this happens is the combination of abstraction and the Singleton design pattern [14], where an abstract service class references a single instance of one of its subclasses. There are several case of this kind in the Java Runtime Environment, all with a high in-degree, including `java.lang.Runtime` and `java.awt.Toolkit`.

Note that this model is supported by the results of earlier research that many cycles form around hubs (nodes with betweenness centrality, usually corresponding to a high degree) [1], and that there are a few dependencies that support a large percentage of cycles and other antipattern instances, and therefore present high-impact refactoring opportunities [9].

However, this model does not offer an explanation for the results for RQ3. But we notice that package naming is sometimes influenced by considerations not related to the semantics of the actual code. Examples are the use of different package branches in the Java Developer Kit (such as `java.*`, `javax.*`, `sun.*`, `org.w3c.*`, ..) based on intellectual property rights, and the use of `org.junit` and `junit` branches in junit to provide older versions for backward compatibility.

But at this stage, this is only one model that could be used to explain the observations we have made. Further research is needed to assess the validity of this explanation.

B. Threats to Validity

Graph extraction: Our tools cannot recognise weak uses relationships created by reflection. This is a common limitation for tools based on static analysis.

Graph pre-processing: Our method to recognise and remove tests is prone to both false positives and false negatives. We expect that it may make non-SCCs to appear slightly more stable for the reasons discussed in section IV-B. Our method to detect generated code may be incomplete as other scripts and tools could have been used in some projects. We think that this is unlikely as most successful open source projects automate routine tasks using build scripts.

SCC Membership: The mechanism to assign vertices to the neighbourhood of cycles is not deterministic, and this could influence the outcome of the respective experiments. However, we executed these experiments at least 10 times, and found that the impact of this on the outcome of the experiments is negligible. In addition, we did not detect any siginificant difference by using a different mechanism (e.g. random assignment of neighborhood).

Detecting STK: As described above in section IV-D3, we use an approximation to detect STK mainly for performance reasons. The result of this is that we may classify some larger STKs cycles as STK even though they are *predominantly* non-STK.

Detecting Visitors: Instances of the visitor design pattern are detected using naming patterns. This might yield both false positives and false negatives. However, in our experience the accuracy of this method is very high.

Controlling for size and dependencies: We have not controlled for the size of classes and the size of their dependencies within each group. Both metrics have been shown to correlate with the change/fault-proneness of components [45], [27], [21]. By investigating the size/dependencies of classes in cycle and their neighborhood, we can further understand the association between the fact that classes in and near cycles are more change-prone as reflected in the results (Table VII, column 3) and whether those classes account for the significant size and dependencies in the systems.

VII. Conclusion

We have investigated whether classes in and near dependency cycles are more likely to change than other classes. We did this in order to investigate whether cycles are related to poorer maintainability as change ripple effects propagate easier through cycles. We used change frequency as an indicator for maintainability. We found no evidence that classes in cycles are more change prone. However, classes in and near cycles have an increased change probability.

We also investigated two heuristics that had been proposed to distinguish between critical and harmless cycles: subtype knowledge and location of the cycle within the package containment tree (PCT). We found no strong correlation between these criteria and change proneness.

We believe that our findings indicate the need for more research to describe and detect cycles as well as other types of anti-patterns that are truly detrimental to the maintainability of a program. A particularly interesting open problem is the relationship between cycles and the scale-free property of class dependency graphs.

In addition, it would be interesting to control for the size of classes and their dependencies as it has been shown to have a confounding effect on the validity of metrics [10]. We plan to investigate this in future work.

References

[1] Hussain A Al-Mutawa, Jens Dietrich, Stephen Marsland, and Catherine McCartin. On the shape of circular dependencies in java programs. In *Software Engineering Conference (ASWEC), 2014 23rd Australian*, pages 48–57. IEEE, 2014.

[2] James M Bieman, Greg Straw, Huxia Wang, P Willard Munger, and Roger T Alexander. Design patterns and change proneness: An examination of five evolving systems. In *Software metrics symposium, 2003. Proceedings. Ninth international*, pages 40–49. IEEE, 2003.

[3] David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 177–186. IEEE, 2005.

[4] David Binkley and Mark Harman. Identifying'linchpin vertices' that cause large dependence clusters. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 89–98. IEEE, 2009.

[5] Markus Dahm. The Apache bytecode engineering library (BCEL). *URL: http://jakarta.apache.org/bcel*, 2010.

[6] Massimiliano Di Penta, Luigi Cerulo, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 217–226. IEEE, 2008.

[7] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 64–73. IEEE, 2014.

[8] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M Ali Shah. Barriers to modularity-an empirical study to assess the potential for modularisation of java programs. In *Research into Practice–Reality and Gaps*, pages 135–150. Springer, 2010.

[9] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M Ali Shah. On the existence of high-impact refactoring opportunities in programs. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference-Volume 122*, pages 37–48. Australian Computer Society, Inc., 2012.

[10] K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, July 2001.

[11] Jean-Rémy Falleri, Simon Denier, Jannik Laval, Philippe Vismara, and Stéphane Ducasse. Efficient retrieval and ranking of undesired package cycles in large software systems. In *Objects, Models, Components, Patterns*, pages 260–275. Springer, 2011.

[12] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[13] Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bartosz Walter, and Pawel Martenka. Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 260–269. IEEE, 2013.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java ™ Language Specification 7th Edition*. Oracle, Inc., California, USA, 2012.

[16] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 121–130. IEEE, 2013.

[17] David Hyland-Wood, David Carrington, and Simon Kaplan. Scale-free nature of java software package, class and method collaboration graphs. In *Proceedings of the 5th International Symposium on Empirical Software Engineering, Rio de Janeiro, Brasil*. Citeseer, 2006.

[18] Makoto Ichii, Makoto Matsushita, and Katsuro Inoue. An exploration of power-law in use-relation of java software systems. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pages 422–431. IEEE, 2008.

[19] Sebastien Jeanmart, Yann-Gael Gueheneuc, Houari Sahraoui, and Naji Habra. Impact of the visitor pattern on program comprehension and maintenance. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 69–78, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Foutse Khomh, MassimilianoDi Penta, Yann-Gal Guhneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[21] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[22] Robert C Martin. The dependency inversion principle. *C++ Report*, 8(6):61–66, 1996.

[23] Robert C Martin. Acyclic visitor. In *Pattern languages of program design 3*, pages 93–103. Addison-Wesley Longman Publishing Co., Inc., 1997.

[24] Robert C Martin. Design principles and design patterns. *Object Mentor*, 1:34, 2000.

[25] Hayden Melton and Ewan Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007.

[26] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *Acm sigplan notices*, volume 43, pages 563–582. ACM, 2008.

[27] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[28] Steffen M Olbrich, Daniela S Cruzes, and Dag IK Sjoberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[29] Jan Overbeck. *Integration testing for object-oriented software*. PhD thesis, Vienna University of Technology, Vienna, Austria, 1994.

[30] Tosin Daniel Oyetoyan, Daniela S Cruzes, and Reidar Conradi. A study of cyclic dependencies on defect profile of software components. *Journal of Systems and Software*, 86(12):3162–3182, 2013.

[31] David Parnas. Designing software for ease of extension and contraction. *Software Engineering, IEEE Transactions on*, (2):128–138, 1979.

[32] Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry in oo programs. *Communications of the ACM*, 48(5):99–103, 2005.

[33] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brossler, and Lawrence G. Votta. A controlled experiment in maintenance: comparing design patterns to simpler solutions. *Software Engineering, IEEE Transactions on*, 27(12):1134–1144, 2001.

[34] Derek de Solla Price. A general theory of bibliometric and other cumulative advantage processes. *Journal of the American Society for Information Science*, 27(5):292–306, 1976.

[35] Jane Radatz, Anne Geraci, and Freny Katki. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610121990:121990, 1990.

[36] Arthur J Riel. *Object-oriented design heuristics*. Addison-Wesley Publishing Company, 1996.

[37] Daniele Romano, Paulius Raila, Martin Pinzger, and Foutse Khomh. Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 437–446. IEEE, 2012.

[38] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[39] Craig Taube-Schock, Robert J Walker, and Ian H Witten. Can we avoid high coupling? In *ECOOP 2011–Object-Oriented Programming*, pages 204–228. Springer, 2011.

[40] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, December 2010.

[41] Marek Vokac. Defect frequency and design patterns: An empirical study of industrial code. *Software Engineering, IEEE Transactions on*, 30(12):904–917, 2004.

[42] Marek Vokáč, Walter Tichy, Dag IK Sjøberg, Erik Arisholm, and Magne Aldrin. A controlled experiment comparing the maintainability of programs designed with and without design patternsa replication in a real programming environment. *Empirical Software Engineering*, 9(3):149–195, 2004.

[43] S. Wasserman and K Faust. *Social network analysis : methods and applications. Structural analysis in the social sciences*. Cambridge University Press, 1994.

[44] Richard Wheeldon and Steve Counsell. Power law distributions in class relationships. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 45–54. IEEE, 2003.

[45] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540. ACM, 2008.