A Study of Cyclic Dependencies on Defect Profile of Software Components

Tosin Daniel Oyetoyan¹, Daniela S. Cruzes¹,²; Reidar Conradi¹
¹Computer and Information Science Department, NTNU, Trondheim, Norway
²SINTEF, Norway
tosindo@idi.ntnu.no, dcruzes@idi.ntnu.no, reidar.conradi@idi.ntnu.no

Abstract - (Background) Empirical evidence shows that dependency cycles among software components are pervasive in real-life software systems, although such cycles are known to be detrimental to software quality attributes such as understandability, testability, reusability, build-ability and maintainability. (Research Goals) Can the use of extended object-oriented metrics make us better understand the relationships among cyclic related components and their defect-proneness? (Approach) First, we extend such metrics to mine and classify software components into two groups - the cyclic and the non-cyclic ones. Next, we have performed an empirical study of six software applications. Using standard statistical tests on four different hypotheses, we have determined the significance of the defect profiles of both groups. (Results) Our results show that most defects and defective components are concentrated in cyclic-dependent components, either directly or indirectly. (Discussion and Conclusion) These results have important implications for software maintenance and system testing. By identifying the most defect-prone set in a software system, it is possible to effectively allocate testing resources in a cost efficient manner. Based on these results, we demonstrate how additional structural properties could be collected to understand component’s defect proneness and aid decision process in refactoring defect-prone cyclic related components.

Keywords - Cyclic Dependencies; Dependency cycle metrics; Software metrics; Empirical Study; Defect proneness; Software components

1 INTRODUCTION

Dependency cycles among components have for long been regarded as symptoms of design decay that should be avoided in software systems (Briand et al., 2001a; Fowler, 2001; Lakos, 1996; Martin, 2000; Martin, 1996; Parnas, 1979). Lakos (1996) argues that cycles among components present a unique problem in terms of understandability since there is no reasonably starting point and no single part of the system can make sense on its own. For Parnas (1979), cycles, referred as “loops in the Uses Relation”, can lead to a situation where nothing works in a system until everything works and that cycles prevent easy extension of software components. Similarly, Fowler (2001) says that cycles are problematic and that it makes system harder to understand “because you have to go around the cycle many times”. In addition, Fowler argues that cycles inhibit the reusability of the class code. Martin (2000) states that cycles inhibit build-ability because to release a module, it should be tested and this implies that all dependent components must compile and build. In addition, many authors have proposed several strategies to optimize stubs in order to break cycles during integration testing, showing that cycles are detrimental to testability (Briand et al., 2001a; Hanh et al., 2001; Kung et al., 1996).

Despite numerous claims that cycles inhibit software quality attributes such as extensibility, understandability, testability, reusability, build-ability and maintainability (Fowler, 2001; Lakos, 1996; Martin, 2000; Parnas, 1979), evidence shows that cycles are widespread in real life software systems (Briand et al., 2001a; Hanh et al., 2001; Kung et al., 1996; Le Traon et al., 2000; Melton and Tempero, 2007a; Parnas, 1979; Tai and Daniels, 1997). The extent to which cycles are pervasive in software systems suggests that design advice (Fowler, 2001; Lakos, 1996; Martin, 2000; Parnas, 1979) regarding cyclic dependencies has not been followed. Melton and Tempero (2007a) argues that, either we have a lot of bad software out there, or the advice is not useful. Intuitively, we would expect that since cycles increase coupling complexities among components (Briand et al., 1998; Briand et al., 2001b), then it should have a positive correlation with defects.

Furthermore, special analysis tools have been developed to facilitate refactoring of code modules, to detect and to warn developers about dependencies that are already cyclic or that could result into cycles. For instance, tools like, JDepend (http://clarkware.com/software/JDepend.html), NDepend (http://www.ndepend.com), JooJ (Melton and Tempero, 2007b), Dependometer (http://source.valtech.com/display/dpm/Dependometer), Classycle (http://classycle.sourceforge.net), Dependency Structural Matrix (Laval et al., 2009) are examples of existing tools and approaches that can be leveraged to dissuade developers from cyclic dependencies. However, in terms of defect-proneness and multiplicity of defects among components, are there undisclosed relationship between cycles and defect-proneness? If yes, we speculate that such evidence can reinforce the need to seek the benefits of such tools, if we want to further discourage dependency cycles among components.

The goal of the study presented in this paper is therefore to investigate the relationships between cyclic dependencies and defect profiles of cyclically dependent components. Although research efforts have focused on breaking cycles during integration testing (Briand et al., 2001a, 2003; Hanh et al., 2001; Kung et al., 1996; Le Traon et al., 2000; Tai and Daniels, 1997) collecting empirical evidence of cycles in software systems (Melton and Tempero, 2007a) and developing tools to
break dependency cycles among components (Hautus, 2002; Melton and Tempero, 2007b; Sangal et al., 2005), there exist gaps regarding empirical evidence of defect proneness of cyclically related components.

We have performed an empirical study on six software systems to provide field evidence of actual cyclic dependencies in object-oriented systems and how such interconnection can be used to discover patterns of defect-prone components. We choose Apache Camel, Apache ActiveMQ, Apache Lucene, Eclipse and openPDC because they are open source and to compare between systems with different development technologies (i.e. Java and C#) and with different functionalities. Lastly, we choose a commercial application to understand if the cyclic effects are the same or different from open source domain.

Thus, the main contribution of this work is an empirical study of cycles and defect proneness of components caught in cycles. Furthermore, we propose metrics that identify cyclic dependency relations among components and use this information to understand the components defect proneness. For instance, it is not sufficient to only know components in cycle; we might be interested in the neighbors (Zimmerman et al., 2011) that depend directly on these cyclically connected components. Our findings will be useful for both practitioners and researchers in the collective effort to minimize defects in real life systems and minimize effort and resource usage in system testing. Also, this study points out additional software structural properties that can be focused for understanding components’ defect proneness. Lastly, this effort is aimed to add significance to the need of collecting cycle metrics and focus on defect-prone cyclically dependent components for refactoring possibilities.

The rest of the work is organized as follows; Section 2 explores related literature to our work. Section 3 describes relationships and dependency concepts among software components and explains cyclic dependencies with examples. Furthermore, we define the terms used in the paper. In section 4, we detail our empirical design setup. In addition, we define the proposed metrics that are used in this paper. Additionally, we define our hypotheses and explain the statistical approaches that we employ. We likewise describe the case studies for this study and explain how our data is collected. In section 5, we present the results of our analysis and provide further discussions of the results and their implications. We draw out threats to the validity of the results in section 6. Finally, in section 7, we conclude the paper with additional notes on future work.

2 RELATED WORK

This study concerns cycles among inter-related components and understanding the defect proneness of components in this cyclically related category by using extended object-oriented (OO) metrics. We thus discuss related work in two areas; the first part covers related work in cyclic dependencies among components and the second part covers related work in OO metrics and approach for determining component’s defect proneness.

2.1 Cyclic dependencies

Over the years, researchers have conducted studies and provided design advice regarding cycles among components. In this section, we review previous work in dependency cycles among classes and packages and present empirical study of cyclic dependencies.

In terms of classes, Parnas (1979) identified “Uses relation” between two components and argues that the loops in the “Uses Relation” are detrimental to extensibility of a software system. Lakos (1996) provided extensive discussion about cyclic dependencies among C++ classes. The author claimed that cyclic physical dependencies among classes of C++ program inhibit understanding, testing and reuse. Other authors also claimed that cycles inhibit system understanding (Fowler, 2001), testing in isolation, integration testing (Briand et al., 2001a; Hanh et al., 2001; Hashim et al., 2005; Kung et al., 1996) and reuse (Martin, 1996). Cyclically connected components are mutually dependent, thus in terms of understanding any of the classes; it is necessary to understand all other classes in the cycle. Furthermore, to test a class in isolation is practically impossible when it is involved in a cycle with other classes (Lakos, 1996). In integration testing, cycles prevent the topological ordering of classes that can be used as a test order (Briand et al., 2001a, 2003; Hanh et al., 2001; Jungmayr, 2002; Kung et al., 1996; Melton and Tempero, 2007a), thereby inhibiting the testability of a system.

From package point of view, in many OO systems developed with language such as Java or C++, package represents a physical organization of software components (Knoernschild, 2012; Lakos, 1996). Packages are used to group classes that perform similar functions. As such they focus on manpower and they represent the granule of release (Martin, 1996). Applications are usually a network of interrelated packages and the work to manage, test, build and release those packages is non-trivial (Martin, 1996). When cycles are formed at the package level, it seriously affects manpower since software engineers working on individual packages need to build with every other dependent package before they can release their package. Cycles among packages have thus been claimed to be detrimental to understandability (Fowler, 2001), production (Lakos, 1996; Martin, 1996), marketing (Lakos, 1996), development (Lakos, 1996; Martin, 1996), usability (Lakos, 1996; Martin, 1996) and reliability (Lakos, 1996).

Although, it has been stated (Briand et al., 2001a; Hashim et al., 2005; Kung et al., 1996; Lakos, 1996) and implied (Jungmayr, 2002; Martin, 1996) that cycles are pervasive in real-life software systems. However, it appears that only Melton
and Tempero (2007a) have performed an elaborate empirical study of cycles on many software systems at the class level. Melton and Tempero carried out an empirical study of 78 Java applications and employed three “Uses Relation” types; “USES”, “USES-IN-SIZE” and “USES-IN-THE-INTERFACE” as stated in Lakos (1996) to describe cyclically connected components within the applications. The result shows that almost all the 78 Java applications contain large and complex cyclic structures among their classes. This study is conducted on open source Java applications showing that further work is still needed to investigate other domains and programming languages before any generalization can be made.

2.2  **OO Metrics and defect proneness of components**

Object Oriented metrics have been widely used to indicate defect proneness of components. Basili et al. (1996) validated a set of OO metrics proposed by Chidamber and Kemerer (1994). Among these metrics, coupling between object classes (CBO) and response for class (RFC), are shown to correlate significantly to a component’s defect. Briand et al. (1998); (2001b) have also conducted several studies that showed CBO, and especially import and method invocation coupling to be important properties when building an OO quality model.

Additionally, Marinescu (2001) identified code smells (GodClass, ShotgunSurgery, GodPackage, etc.) by defining threshold values and rules based on code metrics. The author showed that code smells violated good design principles of low coupling, high cohesion, manageable complexity, proper data abstraction and standard component reuse (Capretz and Capretz, 1996; Capretz and Lee, 1992; Coad and Yourdon, 1991). Empirical study by Olbrich et al. (2009) on two open source applications further showed that different phases could be identified during the evolution of code smells. In addition, they pointed out that code smell infected components display a different change behavior.

Actual dependencies of a component have also been employed to indicate the defect proneness. For instance, Schroeter et al. (2006) demonstrated that imported components in the eclipse software could predict the defect proneness of their dependent components. Further, we have recently validated this approach on a Smart Grid application (Oyetoyan et al., 2012).

Social network analysis has been explored for defect prediction. Zimmermann and Nagappan (2008) showed that there is significant correlation between their proposed dependency graph metrics and the number of defects in the graph related components. Their results from a study of Microsoft Windows Server 2003 demonstrated that a network-based model could predict the number of defects and could identify critical binaries missed by complexity models. In fact, in their previous study of similar system (Zimmermann and Nagappan, 2007) they made an implicit observation that binaries in dependency cycle have on average twice as many defects as those binaries not in cycle. In another related study of Microsoft Vista and Eclipse, Zimmerman et al. (2011) showed that the properties of a component’s neighbor such as size, code churn; complexity, test coverage and organizational structure can influence the quality of the component. However, Weyuker et al. (2008) disputed the effect of the number of developers’ impact on defect-proneness of components. An elaborate empirical study by the authors concluded that the number of developers is not a major factor that could contribute to a component’s defect-proneness.

Yutao et al. (2010) have proposed a multiple-dependency metric, $m$ based on network analysis. The metric measures the degree of reusability of a component (incoming dependencies) as well as its direct and indirect coupling (reachable set). In the open source systems they analyzed, the authors found that fewer classes have high $m$ value and that correlations exist (though weak) with WMC and LCOM (Lack of Cohesion of Method). Indicating that $m$ may be used as a statistical indicator for defect-prone classes identified by WMC or LCOM.

2.2.1  **Related Metrics**

The following metrics regarding coupling between objects in object-oriented systems are of interest for our work:

- **CBO**: The coupling between object classes (CBO) shows the number of other classes that are directly coupled to the class (Basili et al., 1996; Briand et al., 1998; Briand et al., 1999; Briand et al., 2001b; Chidamber and Kemerer, 1994).

- **RFC**: Response for class (RFC) indicates the set of all methods that can potentially be invoked in response to a message received by the object of the class (Basili et al., 1996; Briand et al., 1998; Briand et al., 1999; Briand et al., 2001b; Chidamber and Kemerer, 1994). It considers both direct and indirect connections. However, it does not show if the connection is cyclic or not.

- **CyclicClassCoupling** (Nagappan and Bhat, 2007; Zimmermann and Nagappan, 2008): This metric counts the number of direct cyclic connections between two classes. For instance, $C_1$ depends on $C_2$ and $C_2$ depends on $C_1$ (see Figure 3d). However, this metric only deal with direct cyclic coupling between two classes and does not consider transitive relationship where components can become cyclic indirectly (see Figure 3a).

- **dwReach** (Nagappan and Bhat, 2007; Zimmermann and Nagappan, 2008): This metric shows the number of components that can reach another component with the distance weighted by the number of steps.

The above metrics measure the number of connections between objects and cannot be used for our purpose. Our proposed cyclic dependency metrics in section IV contains metrics that simply flag a component when it has a cyclic relationship.
Although our work extends these previous studies, it differs in focus. We provide the first empirical study of defect proneness of cyclic dependent components. From this study, we are able to point out additional structural complexities that can be focused for defect tracing and testing activities. Additionally, our work indicates coupling property that can be useful and assessed for building quality models.

3 Relationships and Dependency Concepts

In this section we focus on explaining various types of relationships that exist among components when modeling with UML. Furthermore, we describe how a UML diagram is translated to an Object Relation Diagram (ORD) when the analysis concerns a client to server or supplier relationships. In addition, we present the definitions of an ORD that are appropriate for our study. In addition, we explain the component relationship level at which dependency is stronger and how this influences our choice of analysis decision. Lastly, we present and explain cyclic dependencies with examples and provide definitions that are necessary for our metrics.

3.1 Relationships: From UML to Dependency Graph

In software designs, class interactions are modeled based on the various relationships that exist among them (Bennett et al., 1999; Souza and Wills, 1999). If we concern ourselves with UML modeling, these relationships among the various classes can be modeled in UML as association (uni-directional, bi-directional or reflexive), aggregation, composition, generalization (inheritance) and realization. An association relationship indicates a structural relationship between two class objects. The reasons for this relationship and the rules that govern the relationship are specified in an association relationship. Aggregation typifies a “whole-part” (has-a) relationship, where a class is modeled as a part of an aggregate class (whole). The “part” can exist independently of the “whole” and is therefore not destroyed when the lifecycle of the aggregate class ends. A composition relationship (part-of) is a special type of aggregation where the “part” class can no longer exist once the “whole” lifecycle ends. Generalization illustrates an inheritance (is-a) relationship between a child class and its parent (super or base class). Realization relationship exists when a class implements (realizes) the behavior of another class.

An Object Relation Diagram (ORD) has been widely used to describe components and their relationships (Briand et al., 2001a, 2003; Hahn et al., 2001; Kung et al., 1996; Le Traon et al., 2000; Tai and Daniels, 1997). The term component in this study is used to represent a class or a package. A component $X$ is said to have dependency on another component $Y$ if $X$ requires $Y$ to compile or function correctly (Jungmayer, 2002). Three relationship types are described in ORD, that is: inheritance, $I$, association, $A$s and aggregation, $A$g. Where $I$, is used for both inheritance and realization relationships, and $A$g is used to represent both composition and aggregation relationships$^1$, while $As$ maps to other cases of dependencies and associations. When an ORD is represented as a dependency graph, the relationship labels are usually ignored. Figures 1b and 1c show two design diagrams of the implemented code in Figure 1a. In Figure 1b, the UML relationship diagram shows a reflexive association relationship between class B and itself. In other words, an instance of class B can be related to another instance of B. A generalization relationship exists between classes A and B, since B is the super class of A. A Realization relationship exists between classes A and C since class A implements the behavior or contracts specified in class C. An aggregation relationship is shown between classes B and D and lastly, a composition relationship exists between classes D and E. Class E cannot exist when D’s lifecycle ends.

For formal representation of an ORD, we borrow two definitions from Kung et al. (1996) and state these as follows:

- **Definition 1.** An edge labeled digraph $G = (V, L, E)$ is a directed graph, where $V = \{V_1, ..., V_n\}$ is a finite set of nodes, $L = \{L_1, ..., L_k\}$ is a finite set of labels, and $E \subseteq V \times V \times L$ is the set of labeled edges.

- **Definition 2.** The ORD for an OO program P is an edge-labeled directed graph (digraph) ORD = (V, L, E), where V is the set of nodes representing the object classes in P, $L = \{I, Ag, As\}$ is the set of edge labels, and $E = E_I \cup E_{Ag} \cup E_{AS}$ is the set of edges.

Applying these definitions to the ORD presented in Figure 1c gives $V = \{A, B, C, D, E\}$, $L = \{I, Ag, As\}$ and $E = \{E_{I, B}, E_{I, C}, E_{B, D}, E_{D, E}\}$, where $E_{X,Y}$ denotes an edge that connects node X to node Y in the direction of Y. For the purpose of this paper, we ignore the edge labels L and concern ourselves with the set of nodes and the set of edges. Furthermore, in the data collection section (Section 4.4.4), we describe how the set of edges are determined for each class node and each package node.

---

2. Briand et al. (2001a) maps only composition relationship to $Ag$ with the claim that compositions have a lifetime constraints between the whole and the parts and thus represent tight coupling. Whereas, $As$ maps to simple aggregation (a type that is considered as a special type of association in UML and does not denote strong coupling), dependencies and associations.
public class A extends B implements C {
    private Map<String, String> a1;
    public void ma1(){}
    @Override
    public void mb1(){}
    @Override
    public void mc1(){}
    @Override
    public void mc2(){}
}

public class B {
    private D b1; //Aggregation
    List<B> b2; //Reflexive
    //constructor
    public B(D b1) {
        this.b1 = b1;
    }
    private void mb1(){}
    private int mb2(){}
}

interface C {
    public void mc1(){}
    public void mc2(){}
}

public class D {
    private String d1;
    E d2 = new E(); //Composition
    public void md1(){}
    private void md2(){}
    //inner class
class E {
        private double e1;
        protected void me1(){}
    }
}

(a) – Implemented code
(b) – UML relationships diagram
(c) – Object Relation Diagram (ORD)

Figure 1(a – c) – Representation of component relationships with UML and ORD
3.2 Physical or Logical Dependency

Lakos (1996) differentiated between logical relationships and physical dependencies and states that a physical dependency concerns dependency of the physical entities of a software unit and for instance requires the use of an “#include” directive in C++ in another component and that this type is stronger. Physical design decisions impact on deploy-ability, reusability and maintainability. A physical dependency implies that the dependent component requires the dependee component in order to compile and link. In Java systems, we can imply that to mean relationships between a .java and another .java files. In C#, this is equivalent to a .cs files relationships. Physical dependency thus requires that a physical class file have knowledge of another physical class file. A file can enclose multiple classes, including nested classes and we can define the logical relationships among these classes, for example, the case of class D and E in Figure 1. A physical dependency relationship is formed when there is dependency with classes in another physical file. However, we can infer or imply physical dependencies from logical relationships among components (Lakos, 1996). We concern ourselves in this study with dependency at physical level, that is, both files (top-level classes) and packages as described above since we can infer strong dependencies from it.

3.3 Cyclic dependencies

In Figure 1a, let us say that during system evolution, class E for some reasons is required to pass a message to class A. We introduce an instance variable of class A implemented locally in method mel() of class E (see Figure 2a). Figure 2b shows the dependency relationship that now occurs as a result of the new dependency of class E on class A. What we have is a cyclic dependency between classes B, D, E and A. In graph theory, this type is referred as strongly connected component (SCC) (Jungmayer, 2002). If we assume that class D resides in another package or another file, then a strong physical cyclic dependency exist among the connected classes.

![Diagram of Cyclic Dependency](image)

(a) – Inner class E with instance of class A (b) – Cyclic dependency formed as a result of dependency of E on A

Figure 2 – Cycles and representation with Object Relation Diagram (ORD)

3.3.1 Hypothetical Example

As depicted in Figure 3a, cyclic dependency is formed when components depend on one another in a circular manner. For example, B depends on A, C depends on B, D depends on C and A in turn depends on D. In this network diagram, 2 cyclical paths exist: (i) A-D-C-B-A (ii) A-D-C-F-E-A. This relationship covers both direct and indirect connection between components. Cyclic relationships increase coupling complexities and have the potential to propagate defects in a network (Abreu and Melo, 1996). A hypothetical case as depicted in Figure 3a-b demonstrates such effect. From Figure 3a, assume that component I contains some defects. We can further assume that the rest of the components A – H will have a certain probability to inherit the defect from I, since they are directly and indirectly dependent on I. To reduce the likelihood of defect propagation e.g. in Figure 3b, let us say that, a new component J is created so that components D and C depend on J directly thereby breaking the cyclic effect. By performing such a refactoring, the effect of possible defect propagation is reduced to only component G.

---

1 A component that depends on another component is called a dependent component
2 A component that is dependent upon by other component is called dependee component
For the purpose of this paper, we define some of the terms used henceforth: Assume a component $c \in \text{System } P$ then:

D1. **Component's Children**: Components that are directly and transitively dependent on $c$. E.g. in Figure 3(a), All the components except component $I$ are directly or transitively dependent on $A$. Components $G$, and $H$ have no children. We use $T\text{Children}$ for both direct and transitive children and $D\text{Children}$ for direct children. For example, $D\text{Children}(A) = \{B, E\}$.

D2. **Component's Parent**: All components that $c$ is both directly and transitively dependent upon. We use $T\text{Parent}$ for both direct and transitive parents and $D\text{Parent}$ for direct parents. For instance, $T\text{Parent}(G) = \{A, B, C, D, E, F, I\}$ and $D\text{Parent}(G) = \{C\}$.

D3. **Component In-Cycle**: Component $c$ is said to be in cycle, if it has at least one parent that is the same as one of its children. E.g. $B$ is in cycle because its parent $A$ is also one of its children.

D4. **Component Depend on In-Cycle Component**: Component $c$ is said to depend on another in-cycle component if at least one of its direct parents is in cycle. $G$ and $H$ are examples of components that depend on In-Cycle components $C$ and $E$ respectively.

D5. **Component's Minimum Number of Cycle**: The minimum number of cycle that component $c$ is involved with is defined as the sum of the number of its direct children that are in-Cycle and the number of its direct parents that are in-Cycle minus one. For instance, $D\text{Children}(A) = \{B_{in-cycle}, E_{in-cycle}\}$, and $D\text{Parent}(A) = \{D_{in-cycle}\}$. Therefore, $\text{minCycle}(A) = 3-1 = 2$.

D6. **Associated Defect**: Two components have associated defect if a specific defect affect both components. We use defect ID to track associated defect between components.

D7. **Cyclic Propagated Defect**: Consider component $M$ that directly depend on $K$ (Figure 3c). Let us say that $L$ contains some defects. These defects from $L$ cannot be propagated to $M$. However, if $M$ forms a cycle with $L$ by depending on it (Figure 3d), we can thus infer that the defect from $L$ may be propagated to $M$.

### 4 Empirical Design

Our goal in this work is to explore the defect profiles of cyclic dependent components in a system. As explained in (Fowler, 2001; Lakos, 1996; Martin, 2000; Martin, 1996), cyclic dependencies are better studied at physical design levels such as the source file (compilation unit) and package levels, since physical dependencies are formed at such levels. In addition, previous empirical studies (Melton and Tempero, 2007a) on cyclic dependency have performed analysis at the file levels. Furthermore, when developers resolve defects, they usually log the changes at the file level and thus have file to defect mapping. Based on the above reasons, we identify relationships and dependencies at the compilation units (top-level classes for Java) and at the package level. We perform our evaluation in three ways: First, we propose a set of metrics built around cyclic dependency relationships. Second, we use our proposed metrics to mine software components and classify them into two groups, “Cyclic” and “Non-Cyclic”. Third, we statistically evaluate data from cyclic-related components and non-cyclic related components to determine their defect profiles.

#### 4.1 Proposed Metrics for our study

We describe as follows the cyclic metrics and notations for our study. Consider a set of components, $C$ in an object-oriented system. For each component $c \in C$:

1. **Component In-Cycle**:
   
   $\exists p : p \subseteq (T\text{Parent}(c) \land T\text{Children}(c))$
   
   $\forall c \in \text{inCycle}(c) \iff p \neq \emptyset$

   Where $\text{inCycle}(c)$ denotes $c$ to be in a cyclic dependency

   Example 1. (Figure 3a):
if c = A
TParent(A) = {D, C, I, B, F, E} and
TChildren(A) = {B, C, D, E, F, G, H}
p = {B, C, D, E, F}
Since p ≠ ∅ ∴ inCycle(A) ⇔ True

2. Depend On Cycle:
depOnCycle: boolean
∃ x : x ∈ DParent (c) {∀ c. depOnCycle(c) ↔ (¬inCycle(c) ∧ inCycle(x))}
Where depOnCycle(c) denotes c depends on inCycle component x that is a direct parent of c.

Example 2. (Figure 3a):
if c = H
inCycle(H) = False,
But DParent(H) = {E} and inCycle(E) = True

4.2 Hypotheses
The main goal of this study is to investigate the impact of cyclic dependencies among components regarding their defect proneness. To verify the conjecture that the most defects are concentrated in the components with cyclic dependencies, we define our hypotheses as follows:

H_A: Cyclic dependent components are more defect-prone than non-cyclic dependent components.

To evaluate this hypothesis, we further define two sub hypotheses:

• H_A1: The number of defective components in cyclic relationships is significantly higher than non-cyclic defective components.
  With this hypothesis, we seek to establish the group with the higher number of defective components. In addition, the number of defective components in each group allows us to measure the recall value that shows the ratio of defective components in each group to the total number of defective components in the system.

• H_A2: The proportion (ratio) of defective components in cyclic group is significantly higher than the proportion of defective components in non-cyclic group.
  Using this hypothesis, we aim to establish the group with higher defect propagation among their components. It is not sufficient to know the number of defective components in each group. We are also interested in knowing if defects spread in a group more than the other. The proportion data gives us idea about the concentration of defects in each group. This measures the ratio of defective components to non-defective components within each group and allows us to identify the group with relatively higher number of defective components.

H_B1: The actual number of defects in cyclic dependent components is higher than non-cyclic dependent components.
H_B2: Defect density in cyclic dependent group is higher than non-cyclic dependent components.

Defects can be associated in nature, that is, a defect may propagate to a number of components. Therefore, in terms of number of defects, a component may have many defects and many components may have very few defects. If H_A is true, H_B1 therefore, allows us to verify if the components in cyclic group are defective due to more actual defects than the non-cyclic group. If this hypothesis is not rejected, we can conjecture that cycles probably trigger more defects.

Defect density takes the size of the components into account. We compute defect density as the number of defects in each group per the source line of code in the group. We seek to know if there is an implicit relationship between size and defect in each group.

4.3 Statistical Analysis
For this study, we identify cyclic group and non-cyclic group from each system. In Table 1, we use C to represent all cyclic-related group, inC for group with components that are only in-cycle and NC for “Non-Cyclic” group. We have performed analysis both at the class and package levels. A cyclic group consists of all components (classes or packages) that
are flagged to be (1) in-cycle and (2) cyclic-related, i.e. both in-cycle and also directly dependent on in-cycle components. If we use our Figure 3a, then inC and C groups for this hypothetical example consists of components \{A, B, C, D, E, F\} and \{A, B, C, D, E, F, G, H\} respectively, and non-cyclic (NC) group consists of only \{I\}. We use Table 1 to present how the data for each category and for each hypothesis is computed. For each system, we collect both cyclic dependency data and defect data for multiple versions (Table 2). For each version and each group, we determine the number of components, the number of defects, the number of defective components and the source line of code. Subsequently, we compute the proportion data and the defect density per group as shown in Figure 4.

<table>
<thead>
<tr>
<th>Group</th>
<th>#Component</th>
<th>#Defect</th>
<th>#Defective Component</th>
<th>#Non-Defective Component</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>inC</td>
<td>inC&lt;sub&gt;i&lt;/sub&gt;</td>
<td>D&lt;sub&gt;inC&lt;/sub&gt;</td>
<td>inC&lt;sub&gt;D&lt;/sub&gt;</td>
<td>inC-inC&lt;sub&gt;D&lt;/sub&gt;</td>
<td>inC&lt;sub&gt;SLOC&lt;/sub&gt;</td>
</tr>
<tr>
<td>C</td>
<td>C&lt;sub&gt;i&lt;/sub&gt;</td>
<td>D&lt;sub&gt;C&lt;/sub&gt;</td>
<td>C&lt;sub&gt;D&lt;/sub&gt;</td>
<td>C-C&lt;sub&gt;D&lt;/sub&gt;</td>
<td>C&lt;sub&gt;SLOC&lt;/sub&gt;</td>
</tr>
<tr>
<td>NC</td>
<td>NC&lt;sub&gt;i&lt;/sub&gt;</td>
<td>D&lt;sub&gt;NC&lt;/sub&gt;</td>
<td>NC&lt;sub&gt;D&lt;/sub&gt;</td>
<td>NC-NC&lt;sub&gt;D&lt;/sub&gt;</td>
<td>NC&lt;sub&gt;SLOC&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Hypothesis H<sub>A1</sub>: Number of defective components

Hypothesis H<sub>A2</sub>: Proportion of defective components

Hypothesis H<sub>B1</sub>: Actual defect

Hypothesis H<sub>B2</sub>: Defect density

Where X can be inC, C, or NC

The next step is to determine what statistical approach is appropriate to test our hypotheses, either a t-test or non-parametric test. We initially perform statistical test to determine if our data sample is from a normally distributed population. For this, we use the Shapiro-Wilk normality test. If the data is normally distributed, we employ a t-test; otherwise we use a non-parametric statistical approach (Fenton and Pfleeger, 1997) such as Wilcoxon signed rank test (see Appendix B).

Lastly, we test the difference in mean between both groups for significant difference that is greater than zero. Four categories are identified for both groups based on our hypotheses:

I. Number of defective components in each group
II. Proportion of defective components in each group
III. Actual defect counts produced in each group
IV. Defect density for each group measured as actual defect in each group per source lines of code in the group

For these four categories, we test the hypothesis (1-tailed significance test):

- H<sub>0</sub>: µ<sub>C</sub> ≤ µ<sub>NC</sub>: The mean of cyclic group is significantly less than or equal to the mean of non-cyclic group
- H<sub>1</sub>: µ<sub>C</sub> > µ<sub>NC</sub>: The mean of cyclic group is significantly higher than the mean of non-cyclic group

4.4 Data Collection

We have performed a study on two Smart Grid systems, an open sourced (openPDC)\(^5\) and a commercial application (commApp) developed with C#. In addition, we choose an integrated development environment (Eclipse)\(^6\), a search engine (Apache Lucene)\(^7\), an integration framework (Apache-Camel)\(^8\) and a messaging and integration pattern server (Apache-ActiveMQ)\(^9\), all developed with java. We have selected very active projects from the open source community and we also considered projects that have different functionalities and different development languages.

Apache Camel is an integration framework that can serve as a routine and mediation engine between applications. Apache-ActiveMQ is a messaging server with the capability to handle various integration patterns.

OpenPDC is a medium-sized Smart Grid open source software (OSS) named openPDC, supported by the Tennessee Valley Authority (TVA). The solution is developed using the .NET Framework and mainly with the C# programming language. The openPDC is a phasor data concentrator software that is designed to process real time data for user-defined actions and for archival purpose.

The commercial application shares the same Smart Grid domain with openPDC. It is a distribution management system designed to allow for monitoring and planning of Grid operations. It provides real-time operational support by continuously receiving status data from the power grid.

---

\(^5\) http://openpdc.codeplex.com/
\(^7\) http://lucene.apache.org/core/index.html
\(^8\) http://camel.apache.org/index.html
\(^9\) http://activemq.apache.org/index.html
Eclipse is a popular open source integrated development environment (IDE), while Lucene is a high-performance search engine. Table 2 details some properties of the applications we have used for this study.

<table>
<thead>
<tr>
<th>System</th>
<th>Language</th>
<th>#Developers</th>
<th>Domain</th>
<th>License</th>
<th>Bug Tracker</th>
<th>Age</th>
<th>Module</th>
<th>Versions Analyzed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-Camel</td>
<td>Java</td>
<td>34</td>
<td>Routing and Mediation Engine</td>
<td>Open</td>
<td>JIRA</td>
<td>5</td>
<td>CORE</td>
<td>2.10.2, 2.10.1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.10.0, 2.9.2, 2.9.1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.9.0</td>
</tr>
<tr>
<td>Apache-ActiveMQ</td>
<td>Java</td>
<td>24</td>
<td>Messaging and Enterprise Integration Pattern Server</td>
<td>Open</td>
<td>JIRA</td>
<td>6</td>
<td>CORE</td>
<td>5.7.0, 5.6.0, 5.5.1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.5.0, 5.4.2, 5.4.1</td>
</tr>
<tr>
<td>Apache-Lucene</td>
<td>Java</td>
<td>31</td>
<td>Search Engine</td>
<td>Open</td>
<td>JIRA</td>
<td>7</td>
<td>CORE</td>
<td>4.0, 3.6, 3.5, 3.4,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.3, 3.2</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Java</td>
<td>IDE</td>
<td>Open</td>
<td>Bugzilla</td>
<td>All</td>
<td></td>
<td></td>
<td>3.0, 2.1, 2.0</td>
</tr>
<tr>
<td>commApp</td>
<td>C#</td>
<td>28</td>
<td>Smart Grid</td>
<td>Commercial</td>
<td>HP-Quality Center</td>
<td>6</td>
<td>All</td>
<td>4.2.4, 4.2.2, 4.1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.0.1SP4, 4.0.1SP2, 4.0</td>
</tr>
<tr>
<td>openPDC</td>
<td>C#</td>
<td>13</td>
<td>Smart Grid</td>
<td>Open</td>
<td>CodePlex</td>
<td>3</td>
<td>All</td>
<td>1.5, 1.4SP2, 1.4</td>
</tr>
</tbody>
</table>

4.4.1 Defects collection from the defect tracking system (DTS)

We have collected defect data from three different DTSs. Some DTSs contain more details than the others and some are more difficult to filter. Defect repository gives typically a high level overview of a problem report. For example, typical attributes of the HP-QC defect tracking system (QC-DTS) are the Defect ID, severity of the defect, the type of defect, date defect is detected, the module containing the defect, the version where defect is detected, and the date the defect is fixed. These fields are similar to the Apache JIRA and CodePlex DTSs.

Our first step is to determine the bugs that affect each version of the system. In Apache JIRA DTS, we readily use the “Affects Version” field to filter all bugs that affect a particular version of the system. For CodePlex, we use the “RELEASE” field and for HP-QC, we use “Detected in Version(s)”. A certain defect may affect multiple versions of a system. By this we mean “hotspot” defects (Li et al., 2011) that keep re-occurring and span several versions of a system. We include these defects in all the versions they affect. Next, we filtered out “duplicate”, “Not a problem”, and “Invalid” cases from the resolution field. The Eclipse dataset that we use in this paper has been mapped in previous study (Zimmermann et al., 2007).

4.4.2 Method of mapping class files to defects

Version repository on the other hand is a configuration management system used by the developers to manage source code versions. The version system provides historical data about the actual file that is changed and/or added as a result of corrective action (defect fixes), adaptive, preventive and perfection actions (Gupta et al., 2010). Thus, the SVN/CVS provides a detailed granularity level to know which source file(s) in the module(s) are changed to fix a reported bug. A common way to figure out what operation is performed on the source file is to look at the message field of the SVN commit. When developers provide this information with the bug number and/or useful keywords (e.g. bug or fix), it is possible to map the reported defect with the actual source file that is modified to fix it (S’liwerski et al., 2005; Schroeter et al., 2006). In some cases, not all bug commits in the version repository contain the bug number or useful keyword in the message field. In the past, researchers have approached this situation by mapping from defect repository to the version repository (C’ubranic, 2004; Schroeter et al., 2006).

We have used both approaches to map defect from JIRA and HP-QC DTSs to the code changes. The resolution date allows us to map some of the untagged commits in the version system to the resolved bugs. The second approach of mapping from defect repository to code repository is found suitable for CodePlex DTS. None of the bug is tagged in the commit log of the openPDC application. The observed style of developers in this community is to include the SVN revision number of the corrected bug in the comment field of the defect repository (e.g. “resolved with change set 79160”). We use the revision numbers from the comment field to identify class files that are changed because of bug fix. Overall, we mapped an average of 89.5% for Apache-Lucene, 90.1% for Apache-Camel, 75.7% for Apache-ActiveMQ, 71.3% for commApp and 81.4% for openPDC.

---

10 #Developers as used in this study represent all committers to the SVN
nodes (classes) to which form cycles with internal application’s classes. Java API) that developers have no access to their source codes since it is practically shows them to be useful predictors of defect data is easier various types of coupling that is not limited to message passing (method interactions) only. Also, class-level coupling data is easier to collect when using static code analysis and lastly, because ample evidence (Basili et al., 1996; Briand et al., 1998; Briand et al., 2001b; Chidamber and Kemerer, 1994; Zimmerman et al., 2011; Zimmermann and Nagappan, 2008) shows them to be useful predictors of defect-proneness of classes. We use the “USES” relations, which we have defined earlier as DParent and apply them also to the six software applications. We ignore all external library types (e.g., .NET and Java API) that developers have no access to their source codes since it is practically impossible for these external classes to form cycles with internal application’s classes.

Figure 7 shows an example of the actual dependencies for MyClass and mypackage components. In order to collect other nodes (classes) to which MyClass is connected to requires that we scan the text of MyClass. The edge between MyClass and

<table>
<thead>
<tr>
<th>Defect ID/Revision ID</th>
<th>File Changed</th>
<th>Date</th>
</tr>
</thead>
</table>
| id1                   | File 1       | date 1 |...
| id2                   | File 2       | date 2 |...

4.4.3 Aggregating number of defects per class file and per package

In a release, it is possible that multiple reported bugs can be associated to one class file. The unique defect ID is thus appropriate to compute the number of defects fixe that affect a class file and a package. From the mapped change data, we look up each file and determine the total of defects per file by counting the number of unique defect ID in this release. At the package level, we aggregate the unique defect IDs for each class file in the package. As demonstrated in Figure 5, File1, File2 and File3 have 2 defects each, based on the defect ID and Pkg 1 has a total of 3 defects although it contains 3 files with 2 defects each. The unique defect-ID shows that for pkg1, only 3 defects are fixed.

4.4.4 Source code data collection

We have developed a small but very efficient java tool to extract source files meta-data. The source files are downloaded from the version repository. Organizational rules in java source file are substantially different from C# source file. As demonstrated in the relation diagram of a simple F1.java and F1.cs (Figure 6), a java source file has a one-one mapping from file to top-level class and it is not allowed to define another top-level class in a java file. In addition, the top-level class must have the same name as its enclosing file. Also, there is a one-zero or one-one mapping from file to package; a maximum of one package can be defined in a java file. Finally, a java class can contain nested classes (one to many relation). In C#, multiple relations are possible. A file can contain many top-level classes and many top-level namespaces can also be defined in a file. It is also possible that a class contains nested classes and a namespace can equally contain nested namespaces. Unlike java file, the file name does not need to match any of the classes defined in it, although, good practices suggest to have filename as the same as a top-level class.

Since the compilation unit for both Java and C# is the source file and we are considering dependencies at the physical level as explained in section III, we decide for the following:

1. A dependency on any class in a source file implies a dependency on the source file.
2. The cyclic metric for a class is computed using dependencies that cross compilation units (source files). We skip cycles that are formed among classes within a source file.
3. The number of cycles for a compilation unit (source file) is the maximum cycle recorded for any of its classes.

Melton and Tempero (2007a) adapts “USES” relations from Lakos (1996) to a set of Java software to study cyclic dependency among the systems’ classes. These relations have been applied on static code. Identifying coupling among classes using static code analysis has its drawback. As mentioned by Arisholm et al. (2004), because of polymorphism and the common presence of unused code in applications, coupling measures based on static code analysis loose precision, as they do not capture the actual coupling among classes at runtime. This study uses static code analysis because we consider various types of coupling that is not limited to message passing (method-method interactions) only. Also, class-level coupling data is easier to collect when using static code analysis and lastly, because ample evidence (Basili et al., 1996; Briand et al., 1998; Briand et al., 2001b; Chidamber and Kemerer, 1994; Zimmerman et al., 2011; Zimmermann and Nagappan, 2008) shows them to be useful predictors of defect-proneness of classes. We use the “USES” relations, which we have defined earlier as DParent and apply them also to the six software applications. We ignore all external library types (e.g., .NET and Java API) that developers have no access to their source codes since it is practically impossible for these external classes to form cycles with internal application’s classes.
We report the results of mining the software data with the Cycle metrics. In section 5.1, we present the results of the analysis for both package and class mined with the proposed cycle metrics. Sections 5.2 and 5.3 discuss the effect size and sanity checks on the reported results. Lastly, section 5.4 provides further discussions of the results.

We present the results of the statistical analysis of the in-cycle and cyclic-related (both in-cycle and depend-on-cycle) groups versus non-cyclic group. To simplify the replication of this study, we have listed the full results in Appendix A, Tables A.1 (Summary of the systems’ data), A.2 and A.3. Figures 8a-b, 9a-b, 10a-b and 11a-b show the number of defective packages and classes, their proportions, the actual number of defects they produce and their defect densities for the cyclic-related and non-cyclic groups. In addition, in Appendix A, Figures A.1 and A.2, we provide the plots of the outgoing (efferent coupling) and incoming (afferent coupling) dependencies \( <V, E_{\text{out}}, E_{\text{in}} > \) and vertex vs. edge \( <V, E> \) for the cyclic dependency graphs for the last release of each system. As well, we show the diameters (Wasserman and Faust, 1994)\(^\text{11}\) and radius vs. number of cycles for each system. We have used the Floyd-Warshall algorithm (Cormen et al., 2001) to calculate the “all-pairs shortest distance” between the nodes. Also, Table 3 lists the results of data normality tests using Shapiro and the t-tests or non-parametric Wilcoxon-test depending on the Shapiro p-value (see Appendix B, Figure B.3). A very small shapiro-wilk p-value (of less than 0.05) suggests that the data is significantly skewed (positively or negatively) or with significant kurtosis. The p-values of 1-tailed test for in-cycle vs. non-cyclic group is reported in column 2 and the p-values for cyclic-related vs. non-cyclic group are listed in column 3 of Table 3.

\(^{11}\)The diameter of a graph is the length of the shortest path between the most distanced nodes. This is calculated as the maximum of the eccentricities of the nodes or the maximum of the nodes’ geodesic distances in the graph. The eccentricity of a node is the longest geodesic distance between the node and any other node in the graph. A geodesic represents the shortest path between two nodes.
5.1 Distribution of defect and defect-prone components (DPCs) in cyclic and non-cyclic groups

We provide a break down of the results on the four categories of our data using Figures 8, 9, 10, 11 and Table 3. Statistical results at the package level show that the number, proportion and actual defect count of defective components in the cyclic-related group are consistently higher in most cases than those in the non-cyclic. We use Figures 8 and 9 to present the package results as follows:

- For Camel: defective components in cyclic-related group are 4.75 times higher than those in the non-cyclic group (Figure 8a). 31% of packages in the cyclic-related group are defective while 11% are defective in the non-cyclic group (Figure 8b). Furthermore, the cyclic-related group has 6.1 times the number of defect in the non-cyclic group (Figure 9a). Finally, the defect density in cyclic-related group is 0.5 times lower than the non-cyclic group (Figure 9b).

- In ActiveMQ, the defective components in the cyclic-related group are about 8.11 times higher than those in the non-cyclic group (Figure 8a). 45% of the packages in cyclic-related group turn defective while 12% are defective in non-cyclic group (Figure 8b). Defects produced by the cyclic-related group are 8.8 times higher than those produced in the non-cyclic group (Figure 9a). Finally defects per a 1000-LOC in the cyclic-related group are 3.7 times higher than those in the non-cyclic group (Figure 9b).

- In Lucene, the defective components in the cyclic-related group are 11 times higher than defective components in the non-cyclic group (Figure 8a). Over 30% of the packages in cyclic-related group turn defective while 6% of packages in non-cyclic group are defective (Figure 8b). The cyclic-related group has 14 times more defects than the non-cyclic group (Figure 9a). Finally defects per 1000-LOC in the cyclic-related group are twice as higher as those in the non-cyclic group (Figure 9b).

- In commAPP, defective components in cyclic-related group are 2.15 times higher than those in the non-cyclic group. In terms of proportion of defective components in each group, 34% of components in in-cyle group are defective while 9% of components in non-cyclic are found with defects. In addition, the total defects produced by the cyclic-related group are 2.9 times higher than those in the non-cyclic group. The defect density in the in-cycle group is 1.72 times higher than the non-cyclic group.

- For openPDC: defective components in cyclic-related group are 14.2 times higher than those in the non-cyclic group. In terms of proportion, 15% of packages in cyclic-related are defective, whereas, 1% of packages in non-cyclic group turn out to be defective. Also, the defects produced by the cyclic-related group are 20.2 times higher than the non-cyclic group. The defect density in cyclic-related group is 7.3 times higher than the non-cyclic group.

- For Eclipse, defective components in cyclic-related group are about 11 times higher than the non-cyclic group. In terms of proportion, over 50% of components in cyclic-related group are defective whereas 30% in non-cyclic are found with defects.

At the class-file level, Figures 10 and 11 reveal that for:

- Camel: defective components in the cyclic-related groups are 13.7 times more than defective components in the non-cyclic group (Figure 10a). 5.5% of in-cycle classes are defective while the non-cyclic group has 1.7% defective classes. (Figure 10b). The in-cycle classes have about 9.6 times more defects than the non-cyclic group classes (Figure 11a). Lastly, defects per 1000-LOC in the in-cycle group are 1.4 times more than the non-cyclic group (Figure 11b).

- ActiveMQ: defective components in the cyclic-related group are about 3.6 times higher than those in the non-cyclic group (Figure 10a). 12% of the classes in in-cycle group turn defective while 2% of the classes in non-cyclic group are defective (Figure 10b). Defects produced by the cyclic-related group are approximately 4.6 times higher than those in the non-cyclic group (Figure 11a). Finally, defects per a 1000-LOC in the in-cycle group are about 2.74 times more than the non-cyclic group (Figure 11b).

- Lucene: defective components in the cyclic-related group are 4.28 times higher than defective components in the non-cyclic group (Figure 10a). 3% of the classes in cyclic-related group turn defective while 1% of classes in non-cyclic group are defective (Figure 10b). The cyclic-related group has 3.1 times more defects compare to the non-cyclic group (Figure 11a). Finally, defects per a 1000-LOC in the cyclic-related group are the same as that of the non-cyclic group (Figure 11b).

- commAPP: defective components in cyclic-related are 5.2 times more than those in the non-cyclic group. In terms of proportion of defective components in each group, 15% of components in in-cycle are defective while 3% of components in non-cyclic are found with defects. In addition, the total defects produced by the cyclic-related group are 3.2 times higher than those in the non-cyclic group. The defect density in the in-cycle group is 1.5 times higher than those in the non-cyclic group.

- openPDC: defective components in cyclic-related group are 1.43 times higher than the non-cyclic group. In terms of proportion, 2.2% of classes in cyclic-related are defective, whereas, 1.4% of classes in non-cyclic turn out to be defective. Also, the defects produced by cyclic-related group are approximately 0.86 times lower than the non-cyclic group. The defect density in cyclic-related group is about 0.58 times lower than the non-cyclic group.

- Eclipse: defective components in cyclic-related group are about 4.3 times higher than the non-cyclic group. In terms of proportion, 19% of components in the in-cycle group are defective whereas 11% are found with defects in the non-cyclic group.
Figure 8 – (a) #Defective packages and (b) their proportions in in-cycle (inC), Cyclic (inC U DC) and non-cyclic (NC) groups

![Graph](image1)

<table>
<thead>
<tr>
<th>Group</th>
<th>inCd</th>
<th>Cam</th>
<th>ActiveMQ</th>
<th>Lucene</th>
<th>commApp</th>
<th>openPDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>inCd</td>
<td>199</td>
<td>12.5</td>
<td>24.33</td>
<td>3.67</td>
<td>6.50</td>
<td>3.33</td>
</tr>
<tr>
<td>Cd</td>
<td>210.67</td>
<td>12.67</td>
<td>24.33</td>
<td>3.67</td>
<td>14.00</td>
<td>4.67</td>
</tr>
<tr>
<td>NCD</td>
<td>18.67</td>
<td>2.67</td>
<td>3.00</td>
<td>0.33</td>
<td>6.50</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Figure 9 – (a) #Defects and (b) Defect Densities of Packages in in-cycle (inC), Cyclic (inC U DC) and non-cyclic (NC) groups

![Graph](image2)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D(inC)</td>
<td>20</td>
<td>0.40</td>
<td>0.15</td>
<td>0.05</td>
<td>0.29</td>
</tr>
<tr>
<td>D(&amp;C&amp;)</td>
<td>20.17</td>
<td>0.10</td>
<td>0.05</td>
<td>0.09</td>
<td>0.29</td>
</tr>
<tr>
<td>D(NC)</td>
<td>3.33</td>
<td>0.00</td>
<td>0.00</td>
<td>0.09</td>
<td>0.59</td>
</tr>
</tbody>
</table>

Figure 10 – (a) #Defective class-files and (b) their proportions in in-cycle (inC), Cyclic (inC U DC) and non-cyclic (NC) groups

![Graph](image3)

<table>
<thead>
<tr>
<th>Group</th>
<th>inCd</th>
<th>Cam</th>
<th>ActiveMQ</th>
<th>Lucene</th>
<th>commApp</th>
<th>openPDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>inCd</td>
<td>724.67</td>
<td>35.17</td>
<td>50.33</td>
<td>6.33</td>
<td>37.83</td>
<td>2.33</td>
</tr>
<tr>
<td>Cd</td>
<td>905.60</td>
<td>36.5</td>
<td>58.83</td>
<td>7.83</td>
<td>54.50</td>
<td>6.67</td>
</tr>
<tr>
<td>NCD</td>
<td>212.33</td>
<td>2.67</td>
<td>16.33</td>
<td>1.83</td>
<td>10.50</td>
<td>4.67</td>
</tr>
</tbody>
</table>

Figure 11 – (a) #Defects and (b) Defect Densities of class-files in in-cycle (inC), Cyclic (inC U DC) and non-cyclic (NC) groups

![Graph](image4)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D(inC)</td>
<td>22.3</td>
<td>0.47</td>
<td>0.15</td>
<td>0.05</td>
<td>0.34</td>
</tr>
<tr>
<td>D(&amp;C&amp;)</td>
<td>20.83</td>
<td>0.20</td>
<td>0.12</td>
<td>0.12</td>
<td>0.32</td>
</tr>
<tr>
<td>D(NC)</td>
<td>2.33</td>
<td>0.00</td>
<td>0.00</td>
<td>0.09</td>
<td>0.23</td>
</tr>
</tbody>
</table>
stands for the pooled standard deviation derived from the standard deviations \( s_X \) Where:

\[
2007 \text{ apply the H}\
\]

our results.

We discuss in this section the effect size check performed on the statistical data. As noted in Kampenes et al. (2007), effect size quantifies the size of the difference between two groups and allows us to judge whether the conclusions drawn from our hypotheses testing are meaningful or not. It is possible that the effect is small even when the statistical test is significant and vice versa. Therefore, for practical use of the results drawn from this study, we are compelled to carry out an effect size check on our results. In this study, we are concerned with two groups; the cyclic and the non-cyclic groups. We apply the Hedges, \( g \) standardized effect size measure. Hedges, \( g \) is calculated as (Kampenes et al., 2007):

\[
Hedges, g = \frac{X_1 - X_2}{s_p}
\]

Where:

\( X_1 \) and \( X_2 \) represent the sample mean for each defect measure from cyclic and non-cyclic groups and \( s_p \) stands for the pooled standard deviation derived from the standard deviations \( s_1 \) and \( s_2 \) of cyclic and non-cyclic groups respectively as:
Fig. 1. Population and estimated effect size measures, which we will call and in the estimators of the single effect size for non-cyclic group (equals the number of analyzed releases)

\[ n_1 = \text{Sample size for the cyclic group (equals the number of analyzed releases)} \]
\[ n_2 = \text{Sample size for the non-cyclic group (equals the number of analyzed releases)} \]
\[ N = n_1 + n_2 \]
\[ n_1 = n_2 = 6, \text{ for Camel, ActiveMQ, Lucene and commApp.} \]
\[ n_1 = n_2 = 3, \text{ for openPDC and Eclipse} \]

For the effect size test, we are mostly concerned with the number of defect-prone components (DPCs) and the percentage of DPCs (proportion of DPCs*100) in both cyclic and non-cyclic groups. We therefore, in Table 5, report the Hedges, \( g \) for the two measures and for each system.

**Interpretation**

There are different ways to interpret effect size results as described in Kampenes et al. (2007). We choose to compare our effect size results to the reported results in Software Engineering empirical studies and categorized in Kampenes et al. (2007) under Table 9. In this Table, the size category for 284 estimated values for Hedges, \( g \) is given as: Small: 0.00 – 0.376, Medium: 0.378 – 1.000 and Large: 1.002 – 3.40

As shown in Table 5, the effect sizes as measured by the Hedges, \( g \) for both number of DPC and percentage of DPC for the package results are in the “large” category. At the class level, the effect size for openPDC is in the “small” category while in Lucene, it ranges between “medium” (0.52) and “large” (1.12) categories. For the remaining four systems, the effect size falls in the “large” category. It can be explained that these two systems, openPDC and Lucene have very small number of DPCs (see Table A.1). We speculate that if the number of analyzed releases is increased, the results for these two might be somewhat different. Overall, the effect size test suggests that a random selection of defect-prone components in these systems has a higher probability to originate from the cyclic related group, either from in-cycle or both in-cycle and depend-on-cycle groups.

<table>
<thead>
<tr>
<th>System</th>
<th>Package</th>
<th>Class-File</th>
</tr>
</thead>
<tbody>
<tr>
<td>#DPC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camel</td>
<td>X1</td>
<td>S1</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>X2</td>
<td>S2</td>
</tr>
<tr>
<td>Lucene</td>
<td>cf</td>
<td>S3</td>
</tr>
<tr>
<td>commApp</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>openPDC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eclipse</td>
<td>199.33</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>57.74</td>
<td>6.56</td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>41.09</td>
</tr>
<tr>
<td></td>
<td>3.51</td>
<td></td>
</tr>
<tr>
<td>Percentage of DPC</td>
<td>X1</td>
<td>S1</td>
</tr>
<tr>
<td>Camel</td>
<td>31</td>
<td>11</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>Lucene</td>
<td>4.8</td>
<td>12.8</td>
</tr>
<tr>
<td>commApp</td>
<td>17.3</td>
<td>9.1</td>
</tr>
<tr>
<td>openPDC</td>
<td>15.1</td>
<td>0.8</td>
</tr>
<tr>
<td>Eclipse</td>
<td>54</td>
<td>30.2</td>
</tr>
<tr>
<td></td>
<td>2.9</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>2.62</td>
</tr>
<tr>
<td></td>
<td>7.27</td>
<td></td>
</tr>
</tbody>
</table>

5.3 **Sanity Check**

We want to verify if the proportion of defect-prone components (DPCs) in the cyclic group is of interest or not. Earlier, we demonstrated that the cyclic group contains the higher number of DPCs than the non-cyclic group. However, this proportion can be a very small number since the distribution of defects and DPCs in software system is usually skewed (Fenton and Ohlsson, 2000) and the proportion in each group (cyclic or non-cyclic) is relative to the proportion of DPCs in the entire system. As listed in Table A.1, the systems we analyzed agree with this observation because the DPCs are indeed few in number relative to the entire systems’ components.

What is therefore of interest is to see if a standard classifier can find precision/recall over (100 – “actual percentage of DPCs in cyclic group”) or false alarm rates under “actual percentage of DPCs in cyclic group”. If either of these conditions is fulfilled, we can conclude that the proportion of DPCs in the cyclic group is important in this data set. To achieve this objective, we use Naïve Bayes...
(http://www.cs.waikato.ac.nz/ml/weka/) classifier because of its simplicity (Hall et al., 2011) and Random Forest because of its ability to generalize well on small dataset (Breiman, 2001). For the classification task, we employ three independent variables; the source lines of code (SLOC), the weighted method for a class (WMC) and coupling between class objects (CBO) (Efferent, e and afferent, c couplings) metrics because our tool already measures them and because they are shown to be good predictors of defect-proneness of components (Basili et al., 1996; Briand et al., 1998; Chidamber and Kemerer, 1994; Zhang, 2009). We trained the models by using cross validation method on the dataset for each group. In this approach, a training dataset is divided into 10-folds and the model is trained on each fold with the result cross-validated on the rest folds in each iteration. By doing this, we achieve both training of the model using each fold as training set and at the same time testing the model’s performance on the entire dataset. For the purpose of the sanity check, we consider that this approach of model training and testing suffices. Also, since we are not focused on building a reusable model, we therefore do not concentrate on thorough training of each of the models. For these reasons, we have used default classifier parameters for Naïve Bayes and have only changed the default number of trees in Random Forest from 10 to 500.

Table 6 lists the precision, recall, false alarm rates and the actual percentage of defect-prone components in the cyclic group averaged over the number of releases. As shown, in all the cases, the false alarm rates are lower than the actual percentage of defect-prone components in the cyclic group (i.e. Actual %C\(_{DPC}\)). In addition, the precisions for Camel, ActiveMQ, Lucene and Eclipse are over (100-Actual %C\(_{DPC}\)) at the package level. Also, the recalls for Camel, ActiveMQ, Lucene and Eclipse are over (100-Actual %C\(_{DPC}\)). At the class-file level, the false alarm rates for Camel, ActiveMQ, commApp and Eclipse are lower than the actual percentage of defect-prone components in the cyclic group. But, for Lucene and openPDC, the classifiers could not divide between the DPCs and non-DPCs in some of the releases in these dataset because of the few number of DPCs recorded in these two systems. We therefore decided to exclude them from the results. As listed in Table A.2, Lucene has an average of 9.3 DPCs out of 501 class-files and openPDC has an average of 11.3 DPCs out of 616 class-files. Although, the small sample sizes of these two systems and the decision to exclude them based on the above stated reason do not override/invalidate the claims in this study. We however, learn a great deal that sanity checks can guide our decisions regarding where to focus such expensive cyclic dependency analysis efforts in software systems both for research and for industrial practices.

<table>
<thead>
<tr>
<th>System</th>
<th>Precision</th>
<th>Recall</th>
<th>FP Rate</th>
<th>Actual %C(_{DPC})</th>
<th>(100-Actual %C(_{DPC}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camel*</td>
<td>69.6</td>
<td>51.1</td>
<td>8.8</td>
<td>30.5</td>
<td>69.5</td>
</tr>
<tr>
<td>ActiveMQ*</td>
<td>75.2</td>
<td>43.5</td>
<td>10.8</td>
<td>44.9</td>
<td>55.1</td>
</tr>
<tr>
<td>Lucene+</td>
<td>77.1</td>
<td>57.9</td>
<td>8.3</td>
<td>32.4</td>
<td>67.6</td>
</tr>
<tr>
<td>commApp*</td>
<td>67.8</td>
<td>55.1</td>
<td>5.1</td>
<td>19.2</td>
<td>80.8</td>
</tr>
<tr>
<td>openPDC+</td>
<td>55.6</td>
<td>19.4</td>
<td>3.8</td>
<td>15.1</td>
<td>84.9</td>
</tr>
<tr>
<td>Eclipse+</td>
<td>67.9</td>
<td>71.2</td>
<td>34.1</td>
<td>50.0</td>
<td>50.0</td>
</tr>
<tr>
<td>Class-File</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camel*</td>
<td>25.2</td>
<td>26.2</td>
<td>3.6</td>
<td>4.5</td>
<td>95.5</td>
</tr>
<tr>
<td>ActiveMQ*</td>
<td>33.8</td>
<td>35.5</td>
<td>6.8</td>
<td>8.9</td>
<td>91.1</td>
</tr>
<tr>
<td>Lucene</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>97.8</td>
</tr>
<tr>
<td>commApp*</td>
<td>42.3</td>
<td>45.7</td>
<td>6.4</td>
<td>10.1</td>
<td>89.9</td>
</tr>
<tr>
<td>openPDC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>97.8</td>
</tr>
<tr>
<td>Eclipse+</td>
<td>55.7</td>
<td>26.1</td>
<td>3.3</td>
<td>13.8</td>
<td>86.2</td>
</tr>
</tbody>
</table>

* Naïve Bayes
+ Random Forest

5.4 Discussion

Clearly, the results show interesting trends of significant higher defect profiles for cyclic dependent components in the systems. As revealed in Table 4, at the package level the null hypotheses for \(H_{A1}\) and \(H_{A2}\) are rejected for all the systems indicating that the results are all significant. Similarly, the null hypotheses for \(H_{B1}\) are rejected for all the 5 systems that we have their actual defect dataset. We fail to reject the null hypotheses of \(H_{B2}\) for Camel and openPDC. At the class level the null hypotheses for \(H_{A1}\), \(H_{A2}\) and \(H_{B1}\) are equally rejected for all the systems except for openPDC and \(H_{A2}\) for Eclipse. However, for Eclipse, the effect size in Table 5 shows a large effect. This confirms that, even though the statistical test is not significant which largely can be due to the small sample size (number of releases). The effect size shows that the difference between the two groups for Eclipse is not negligible. The null hypotheses for \(H_{B2}\) are rejected for 3 out of the 5 systems. In all the cases where we fail to reject the null hypothesis for \(H_{B2}\), it is either there are higher number of cyclic components than non-cyclic or that the cyclic group’s size (LOC) is higher than the non-cyclic group.

openPDC shows a contrasting result to the rest of the systems at the class level. It is hard to imply any pattern from the C# applications’ results at the class level because of the sample size (i.e. Number of
systems analyzed) and also because of the number of defect-prone components available for this study. Further studies will be necessary to observe patterns in this direction.

5.4.1 Multiplicity of Defect
A clear observation from the results is that the cyclic-related group has significantly more defective components and accounts for higher number of defects than those in the non-cyclic group. In addition, the proportion of defective components in cyclic group is higher than those in the non-cyclic group except for openPDC (class-file level). It thus means that components in cyclic relationships tend to be more defect-prone and that possibly, defects propagate more among components in cyclic relationships. Cyclic dependencies increase the probability of defect propagation and the tendency to make the system fragile, thus leading to possible increase in the number of system’s defects. While we cannot claim exclusively that cyclic relationships is unequivocally responsible for this behavior, the results from this study, however, lend support to this pattern. This effect poses huge maintenance challenge as the system evolves. Defects become difficult to trace and system become become more strenuous to test, thus resulting into higher maintainability cost.

5.4.2 Cycle-Size Relationship
We discover a positive correlation between LOC and minimum number of component’s cycle for both the packages and the classes in all the 6 systems. In many cases, there is a correlation of more than 0.5 between the size and the minimum number of component’s cycle.

A look at the cyclic-related group distribution against the size (KLOC) in each group (Appendix A, Tables A.2 and A.3) reveals for example that:

- For ActiveMQ: the in-cycle group has about 32.5% (inc/N) of the total classes but accounts for 55% of the total size (KLOCinc/(KLOCc+KLOCnc)).
- For commApp: Packages in the cyclic group are 12.6% of the total packages and this number account for a total of 32% of the total size. Relatedly, the cyclic group at the class level contains 25.2% of the total classes and contributes 46.8% of the total size.
- For Lucene: Classes in the cyclic-related group are 38.8% of all the classes but account for 57.5% of the total size.

This effect accounts for the mixed results for hypothesis \( H_R_2 \). The correlation values also show that some large classes have many cycles and thus seem to promote cyclic relationships among interconnected components. (Melton and Tempero, 2007a) made similar observations about the presence of cycles in some large classes. Lines of Code (LOC) and degree of coupling have long been validated to correlate to defect-proneness (Basili et al., 1996; Briand et al., 2001b; Menzies et al., 2007). The large components we found in the cyclic group contain many cycles and a high number of incoming and outgoing couplings. A beneficial approach would then be the need to reduce the cyclic connections. Performing such refactoring would invariably reduce both the size and tight coupling in these large components.

5.4.3 Number of Defect Prone Components in cyclic vs non-cyclic group
In terms of number of defective components, cyclic relationships are convincingly important. As observed in Figure 12, the number of defective classes located in the cyclic group is very high. For instance, Apache Camel has 90% of all the defective classes (82% at the package level) in the in-cycle group and 93% (83% at the package level) when combined with depend-on-cycle, that is, the cyclic-related group. We observe however that both applications developed with C# (commApp and openPDC) give the least results in the in-cycle group. Further investigation of many C# systems will thus be necessary to study the defect patterns of components in the cyclic group. Overall, this is a very useful finding that can be employed during system testing to effectively allocate testing resources in a software development and maintenance project. Furthermore, we suggest that based on these results, it is possible to investigate the cyclic metrics for improving existing quality models. Finally, since cyclic related components account for the highest number of defects and defect-prone components in these systems, we argue that focusing on defect-prone cyclically related components for refactoring could be a positive step. Our speculation therefore is that since cycles increase structural complexities (Briand et al., 1998; Briand et al., 2001b), performing such refactoring by taking advantage of existing refactoring tools could reduce the defect-proneness of components and consequently improve the reliability of the system.

5.4.4 Package vs Class
The burden of cyclic dependency is high as it increases the cost of software testing to trace or track a defect. As noted in the results, package level results are more significant than class level. openPDC has significant results for cyclic group at the package level even though at the class level the results are mostly significant for the non-cyclic group. This reinforces the Acyclic Dependencies Principle as proposed by Martin (2000). Package to package dependency also implies for instance, in java that an
“import” directive is used. Additionally, it translates to a strong cyclic physical dependency as mentioned by Lakos (1996). Cyclic dependencies among packages will result into strong structural complexity by making the modules to be tightly coupled together and thereby increasing the tendencies of defect propagation. As Martin (2000) states, “a dependency upon a package is a dependency upon everything within the package” The implication is increasing cost of testing and maintenance as the system evolves. Is it then necessary to focus on this property? Our empirical results in this study suggest an affirmative yes. Empirical evidence shows that cycles in real-life systems mostly grow as these systems evolve (Melton and Tempero, 2007a), our results also agree with this pattern leaving us with strong doubt that refactoring option hardly focus on breaking dependency cycles. This study has very useful implications for maintenance engineers and system testers. The more information we have about the groups or subsets within a software system with the most defect-prone components, the better we can allocate quality assurance resources and efforts to trace and test components in the system.

6 Threats to Validity

We have analyzed and evaluated two Smart Grid systems, an integrated development environment, a search engine application, a versatile routing and mediation engine and a messaging and integration pattern server. Although, these six systems vary in terms of properties such as domain, functionality, programming language, size, age, usage, context and study period, we cannot claim that the observed defect patterns or related will hold for other systems. As it is with most case studies, we cannot generalize these results across all systems. Further studies will be necessary to compare results across several systems and domains.

Our study is based on static coupling measurements and not dynamic coupling measurements (Arisholm et al., 2004); as such actual coupling among classes at runtime may not be completely captured. This imprecision can occur due to polymorphism, dynamic binding and dead code in the software. This as it may, static code analysis has been found to be practically useful and less expensive to collect (Basili et al., 1996; Briand et al., 1998; Briand et al., 2001b; Chidamber and Kemerer, 1994; Zimmerman et al., 2011; Zimmermann and Nagappan, 2008). Our study collects coupling types that are not only based on method invocation. In addition, static coupling measures reflects to a very high degree the coupling among classes at runtime. We do not think the data collected based on static code analysis can bias our result in any significant manner.

For this study, we have relied on the defects logged in the defect tracking systems of each application. Our approach of defect data extraction is similar to what other researchers have used in the past (C’ubranic, 2004; S’liwerski et al., 2005; Schroeter et al., 2006). Nevertheless, a common threat is whether defects logged in the DTS are accurately tagged in the respective code changes in the version systems. In addition, we cannot be sure if all defects are logged in the DTS especially in cases where the defects are discovered by the developers. Also, there could be cases that the message log of the file that consists a change is not tagged with the bug numbers of the resolved defect. Furthermore, there could be cases of typographical error in the recording of the bug number in the version systems (C’ubranic, 2004) and lastly, it is still possible that duplication will occur. For instance, in cases where the commits in the log is not tagged with the bug number from the defect tracking system, we can never be sure that a commit with a particular bug fix is not “re-commit” in the version control system with the same bug fix. All these are common threats to the internal validity of studies that use mapped data from both the DTS and the version control system. Comparably, independent defect dataset of Eclipse yield results in the same direction as the defect dataset that we collected.

We address construct validity using four different hypotheses. These hypotheses measure in detail the number, ratio and size of defect profiles of components in both groups. All dimensions to establish which particular group has higher defect profile are adequately captured with the stated hypotheses.
7 CONCLUSIONS AND FUTURE WORK

We have carried out, to the best of our knowledge the first and an extensive investigation into cyclic properties of software components and their defect profiles. Using our proposed cycle metrics, we divided the mined data into two groups; cyclic group and non-cyclic group. Statistical analysis reveals that components in cyclic relationships are more defect-prone, having more number of defects and containing more defective components. In addition, it shows that defect propagation in the cyclic group is significantly higher than non-cyclic group. This study shows additional structural component property that may impact on the defect proneness of software components.

Furthermore, it reinforces the results from previous studies on coupling complexity and the impacts on system quality. A noteworthy observation is the presence of cycles in all the systems that we analyzed. Evidence from previous studies supports our result that cycles are indeed pervasive in real-life systems. This further supports our hypothesis that cyclic dependencies should be considered when collecting structural properties of software components.

These results have implications for software maintenance. By focusing on the cyclic group, it is possible to discover most defects and defective components in the system. Testing resources can therefore be effectively allocated to trace defects and test components in a cost efficient manner.

As further study, we seek to analyze a large number of versions in each system we have analyzed so as to understand the evolution behavior of dependency cycles and defect proneness. We seek to know, if defective components increase in cyclic group as the system evolves and if certain factors have some effect (such as refactoring) on the evolution of defect in the cyclic group.

In this study, we have used all types of dependency relationships that result in cycles. Some dependencies are stronger than the other in terms of their coupling characteristics (Briand et al., 1999; Kung et al., 1996). Can we identify which dependency relationship (Inheritance, Aggregation or Composition) contribute most defects in a cyclically dependent components? Lakos (1996) explained that intrinsic cyclic dependencies are those cycles that cannot be avoided giving example of a Node and Edge in a graph, with node having information about the edge and vice-versa. Are there cycles we may care less about regarding their tendencies to propagate defects among inter-related components and thus prune the cyclic group to those with higher probability of defect proneness? We would investigate these in our future work.

In addition, we plan to investigate the most common types and severity of defects involved in cyclic dependencies and compare to non-cyclic group. Also, we will investigate how these results can be used in combination with other approaches to improve defect prediction models. Based on the current results, it is positive that we can employ the cycle variables as predictors of a component’s defect-proneness.

REFERENCES


APPENDICES

Appendix A - Software data

Table A.1 – Summary of software source code and defect data

<table>
<thead>
<tr>
<th>Release-Version</th>
<th>Date</th>
<th>#Pkg</th>
<th>#Class-File</th>
<th>#Class</th>
<th>KLOC</th>
<th>#Defective Class-File</th>
<th>#Defect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-Camel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.10.2</td>
<td>Oct 15 2012</td>
<td>67</td>
<td>991</td>
<td>1108</td>
<td>78.91</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>2.10.1</td>
<td>Aug 28 2012</td>
<td>67</td>
<td>991</td>
<td>1108</td>
<td>78.55</td>
<td>20</td>
<td>9</td>
</tr>
<tr>
<td>2.10.0</td>
<td>Jul 01 2012</td>
<td>67</td>
<td>991</td>
<td>1108</td>
<td>78.23</td>
<td>59</td>
<td>30</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Apr 17 2012</td>
<td>65</td>
<td>959</td>
<td>1074</td>
<td>74.97</td>
<td>31</td>
<td>12</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Mar 05 2012</td>
<td>65</td>
<td>955</td>
<td>1068</td>
<td>74.32</td>
<td>23</td>
<td>17</td>
</tr>
<tr>
<td>2.9.0</td>
<td>Dec 31 2011</td>
<td>65</td>
<td>952</td>
<td>1063</td>
<td>73.43</td>
<td>86</td>
<td>49</td>
</tr>
<tr>
<td>Apache-ActiveMQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.7.0</td>
<td>Nov 22 2012</td>
<td>82</td>
<td>1517</td>
<td>1665</td>
<td>136.22</td>
<td>35</td>
<td>68</td>
</tr>
<tr>
<td>5.6.0</td>
<td>Jun 15 2012</td>
<td>83</td>
<td>1505</td>
<td>1649</td>
<td>133.25</td>
<td>88</td>
<td>102</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Oct 16 2011</td>
<td>78</td>
<td>1331</td>
<td>1472</td>
<td>118.27</td>
<td>54</td>
<td>76</td>
</tr>
<tr>
<td>5.5.0</td>
<td>Apr 01 2011</td>
<td>78</td>
<td>1331</td>
<td>1472</td>
<td>118.27</td>
<td>115</td>
<td>105</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Dec 02 2010</td>
<td>77</td>
<td>1258</td>
<td>1393</td>
<td>113.01</td>
<td>80</td>
<td>66</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Sept 21 2010</td>
<td>77</td>
<td>1256</td>
<td>1386</td>
<td>112.20</td>
<td>79</td>
<td>63</td>
</tr>
<tr>
<td>Apache-Lucene</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.0</td>
<td>Oct 12 2012</td>
<td>20</td>
<td>620</td>
<td>1115</td>
<td>76.60</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>3.6</td>
<td>Apr 12 2012</td>
<td>18</td>
<td>503</td>
<td>810</td>
<td>73.78</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>System</td>
<td>N</td>
<td>N_d</td>
<td>inC</td>
<td>DC</td>
<td>C</td>
<td>NC</td>
<td>inC_d</td>
</tr>
<tr>
<td>----------</td>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>Camel</td>
<td>973.17</td>
<td>39.17</td>
<td>641.00</td>
<td>181.17</td>
<td>822.17</td>
<td>151</td>
<td>35.17</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>75.17</td>
<td>75.17</td>
<td>1366.33</td>
<td>1366.33</td>
<td>1366.33</td>
<td>1366.33</td>
<td>1366.33</td>
</tr>
<tr>
<td>Lucene</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
</tr>
<tr>
<td>commApp</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
</tr>
<tr>
<td>openPDC</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
</tr>
<tr>
<td>Eclipse</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
<td>79.17</td>
</tr>
</tbody>
</table>

### Table A.2 – Average of #defective components and their proportions for both Cycle and Not-In-Cycle groups

<table>
<thead>
<tr>
<th>System</th>
<th>KLOC_{inC}</th>
<th>KLOC_{C}</th>
<th>KLOC_{NC}</th>
<th>D_{def}</th>
<th>D_{C}</th>
<th>D_{NC}</th>
<th>D</th>
<th>DD_{def}</th>
<th>DD_{C}</th>
<th>DD_{NC}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camel</td>
<td>60.71</td>
<td>66.30</td>
<td>10.1</td>
<td>22.30</td>
<td>20.83</td>
<td>2.33</td>
<td>22.33</td>
<td>0.34</td>
<td>0.32</td>
<td>0.23</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>66.63</td>
<td>77.82</td>
<td>44.05</td>
<td>37.17</td>
<td>39.83</td>
<td>8.67</td>
<td>44.00</td>
<td>0.57</td>
<td>0.52</td>
<td>0.19</td>
</tr>
<tr>
<td>Lucene</td>
<td>38.29</td>
<td>46.08</td>
<td>20.48</td>
<td>4.33</td>
<td>5.50</td>
<td>3.00</td>
<td>6.33</td>
<td>0.13</td>
<td>0.13</td>
<td>0.14</td>
</tr>
<tr>
<td>commApp</td>
<td>38.29</td>
<td>46.08</td>
<td>20.48</td>
<td>4.33</td>
<td>5.50</td>
<td>3.00</td>
<td>6.33</td>
<td>0.13</td>
<td>0.13</td>
<td>0.14</td>
</tr>
<tr>
<td>openPDC</td>
<td>38.29</td>
<td>46.08</td>
<td>20.48</td>
<td>4.33</td>
<td>5.50</td>
<td>3.00</td>
<td>6.33</td>
<td>0.13</td>
<td>0.13</td>
<td>0.14</td>
</tr>
<tr>
<td>Eclipse</td>
<td>38.29</td>
<td>46.08</td>
<td>20.48</td>
<td>4.33</td>
<td>5.50</td>
<td>3.00</td>
<td>6.33</td>
<td>0.13</td>
<td>0.13</td>
<td>0.14</td>
</tr>
</tbody>
</table>

### Table A.3 – Average of LOC, Actual defect and defect density for both Cycle and Not-In-Cycle groups

---

12 inC = in-cycle; DC = depend-on-cycle; C = (inC ∪ DC); NC = non-cycle; N = Number of Components; X_d = Defective (X), where X can represent inC, C, DC, or NC, D = Total defect in the system; D_{X} = Total defect for X group; DD_{X} = D_{X}/KLOC_{X}, Defect density of X group; P(X_{d}) = X_{d}/X, proportion of defective X group
(a) Camel - r2.10.2

(b) ActiveMQ - r5.7.0

(c) Lucene - r4.0.0

(d) commApp - r4.2.4

(e) openPDC - r1.5
Figure A.1 - Scatter plots of $<V, E_{out}, E_{in}>$ and $<V, E>$ for cyclic dependency graphs for the last release of (a) Camel (b) ActiveMQ (c) Lucene (d) commApp (e) openPDC (f) Eclipse

(a) Eclipse - r3.0
(b) Camel - r2.10.2
(c) ActiveMQ - r5.7.0
(d) Lucene - r4.0

13 $E_{out}$: Outgoing edge from a component and $E_{in}$:Incoming edge from a component. Each dot in the chart represents a single component (class file) and shows the number of $E_{out}$ and the number of $E_{in}$ at the same time

14 Each dot in the $<V, E>$ graph represent a cyclic dependency graph with a number of nodes and edges
Figure A.2 – Diameter\textsuperscript{15} and Radius\textsuperscript{16} vs. number of Cycles for the last release of (a) Camel (b) ActiveMQ (c) Lucene (d) commApp (e) openPDC (f) Eclipse

\textsuperscript{15} Diameter is the maximum eccentricities of the nodes in the graph
\textsuperscript{16} Radius is the minimum eccentricities of the nodes in the graph
Appendix B – R Code

# Author: Tosin Daniel Oyetoyan

# Normality shapiro.test
# T.Test
# Non-Parametric wilcox.test

#x1 -> inCycle
#x2 -> Not-In-Cycle

generic <- function(x){
    data <- read.table("/Users/odtosin/Documents/activemq/ndata/activemq-
parent-5.7.0_FilestatCycle.csv", header=TRUE, sep=";", na.strings="NA", dec=".", strip.white=TRUE);
    # Number of defective components
    x1 <- c(data[2,2], data[9,3], data[9,4], data[9,5], data[9,6], data[9,7]);
    x2 <- c(data[7,2], data[7,3], data[7,4], data[7,5], data[7,6], data[7,7]);
    p <- x1-x2;
    numS <- shapiro.test(p);
    if(num$voice > 0.05){
        numT <- wilcox.test(x1,x2,paired=TRUE, conf.level=.95, alternative="greater")
    } else {
        numT <- wilcox.test(x1,x2,paired=TRUE, alternative="greater")
    }
    # Proportion of defective components
    x1 <- c(data[12,2], data[12,3], data[12,4], data[12,5], data[12,6], data[12,7]);
    x2 <- c(data[13,2], data[13,3], data[13,4], data[13,5], data[13,6], data[13,7]);
    p <- x1-x2;
    propS <- shapiro.test(p);
    if(prop$voice > 0.05){
        propT <- t.test(x1,x2,paired=TRUE, conf.level=.95, alternative="greater")
    } else {
        propT <- wilcox.test(x1,x2,paired=TRUE, alternative="greater")
    }
    # Actual number of defect
    x1 <- c(data[22,2], data[22,3], data[22,4], data[22,5], data[22,6], data[22,7]);
    x2 <- c(data[23,2], data[23,3], data[23,4], data[23,5], data[23,6], data[23,7]);
    p <- x1-x2;
    defS <- shapiro.test(p);
    if(def$voice > 0.05){
        defT <- t.test(x1,x2,paired=TRUE, conf.level=.95, alternative="greater")
    } else {
        defT <- wilcox.test(x1,x2,paired=TRUE, alternative="greater")
    }
    # Defect density
    x1 <- c(data[26,2], data[26,3], data[26,4], data[26,5], data[26,6], data[26,7]);
    x2 <- c(data[27,2], data[27,3], data[27,4], data[27,5], data[27,6], data[27,7]);
    p <- x1-x2;
    ddS <- shapiro.test(p);
    if(dd$voice > 0.05){
        ddT <- t.test(x1,x2,paired=TRUE, conf.level=.95, alternative="greater")
    } else {
        ddT <- wilcox.test(x1,x2,paired=TRUE, alternative="greater")
    }
    head <- c("#DefSh", "#DefT", "propS", "propT", "defS", "defT", "ddS", "ddT");
    res <- c(num$voice,numT$voice,propS$voice,propT$voice,defS$voice,defT$voice,ddS$voice,ddT$voice);
    cres <- cbind(head,res);
    write.csv(cres,file="/Users/odtosin/Documents/activemq/ndata/TTtests-Cycle.csv", quote = FALSE);}