A Decision Support System to Refactor Class Cycles

Tosin Daniel Oyetoyan1,2
1Computer and Information Science
Norwegian University of Science and Technology, Trondheim, Norway
tosindo@idi.ntnu.no

Daniela Soares Cruzes2
2Software Engineering, Safety and Security
SINTEF, Trondheim, Norway
{daniela, tosin.oyetoyan}@sintef.no

Christian Thurnmann-Nielsen3
3EVRY ASA.
Oslo, Norway
christian.thurnmann-nielsen@evry.com

Abstract—Many studies show that real-world systems are riddled with large dependency cycles among software classes. Dependency cycles are claimed to affect quality factors such as testability, extensibility, modifiability, and reusability. Recent studies reveal that most defects are concentrated in classes that are in and near cycles. In this paper, we (1) propose a new metric: IRCRSS based on the Class Reachability Set Size (CRSS) to identify the reduction ratio between the CRSS of a class and its interfaces, and (2) presents a cycle-breaking decision support system (CB-DSS) that implements existing design approaches in combination with class edge contextual data. Evaluations of multiple systems show that (1) the IRCRSS metric can be used to identify fewer classes as candidates for breaking large cycles, thus reducing refactoring effort, and (2) the CB-DSS can assist software engineers to plan restructuring of classes involved in complex dependency cycles.

Index Terms—Dependency cycle, CRSS, refactoring, software quality, decision support system.

I. INTRODUCTION

Best design practice advocates to avoid dependency cycles between software artifacts [1-4]. Dependency cycles are claimed to increase structural complexity among software artifacts such as classes or packages, and to inhibit software qualities like understandability, modifiability, testability, reusability and extensibility [1, 2, 4-6]. Testing a class in isolation is practically impossible when in a cycle with other classes [2]. A class that is tied to a large chunk of unnecessary classes cannot be reused effectively [2]. In integration testing, cycles prevent the topological ordering of classes that can be used as a test order [7-9]. Recent studies have investigated the relationship between dependency cycles and defects [10-12], and found that most of the defects are concentrated within components in and near cycles.

Application development frameworks have considered binding dependencies at runtime to better manage dependencies and provide loose coupling among modules, e.g., dependency injection frameworks (e.g. Spring framework [13]) and dynamic component models (e.g. OSGi framework [14]). Despite these advances, empirical evidence shows that dependency cycles are pervasive in modern software systems [15, 16], at different granularity levels. Time-to-market often forces developers to accumulate technical debt, e.g., by focusing more on “functional code” rather than “maintainable code” [17]. This suggests a need for approaches and tools to deal with accumulated technical debt through refactoring of large and complex cycles.

A major motivation for developing a cycle breaking decision support system is based on dialog with an industrial partner seeking to refactor class cycles, but who found no support in the C# development environment (Visual Studio). The developers do not envision an automated approach or tool where they lose control of the code structure and organization after refactoring. One respondent says: “When you have a complex part of code, it seems more like you are losing control when you just press a button and it does everything for you: which is not ideal. Especially when there is complex code and you want to know what’s going on when you are debugging”.

Against this background, we have implemented a decision support system (DSS) for refactoring class cycles. It is called DSS because it proposes architectural refactoring actions to maintenance engineers, and indicates code locations where actions can be manually implemented. The problem of breaking dependency cycles at the class granularity level is not trivial. Class cycles are large and much more complex than cycles at higher abstraction layers.

Breaking large cycles requires heuristics to suggest the minimum edges that should be treated (e.g. greedy cycle removal) [18]. Such heuristics have been applied to dependency cycle problems among software artifacts (e.g. in [19]). However, there are challenges with cycle removal heuristics when applied to software artifacts, e.g., there are edges suggested that are impractical to refactor [19]. They do not take into account the effect each edge removal/reversal has on the current structure of the system. For large and complex cycles, the minimum number of edges to break the cycles is usually large, and translates to creating a large number of new components. Lastly, breaking the suggested edges does not guarantee that the cycles would be removed. Approaches and tools are therefore needed to simulate refactoring in an adaptive and dynamic way.

In this context, we have investigated metrics to support cycle removal heuristics. This paper proposes a new metric named interface-CRSS reduction rate (IRCRSS), based on the class reachability set size (CRSS) metric [20]. The CRSS metric counts for a given class, all other classes in the system’s source code it requires for its compilation. The CRSS metric was chosen because it provides possibility to limit the number of components to be introduced during cycle breaking refactoring. This discussion is elaborated in Section II. The proposed IRCRSS metric and approach are evaluated using the cycle-breaking decision support system (CB-DSS).
Three research questions are stated to determine the performance of the new metric and the usefulness of the CB-DSS.

**RQ1** Will tuning with IRCRSS produce a refactoring fitness value that is lower than refactoring without?

**RQ2** Will every application benefit from this improvement?

**RQ3** Is the number of refactoring reduced with the IRCRSS metric?

Lastly, we performed a qualitative evaluation of the CB-DSS in an industrial setup.

This paper is structured to partly follow the design science methodology [21]. The problem identification is discussed in this section. Section II provides the background of this work. Section III presents the implementation of CB-DSS. Section IV presents the results of validating the approach. Section V provides the evaluation of the metric and the system on different cases. Section VI draws out the threats and limitations of the system. The conclusion is in Section VII.

II. BACKGROUND

A. Class Reachability Set Size (CRSS) Metric

Melton and Tempero [20] present a metric named “class reachability set size” (CRSS) to detect package partitioning problems in software systems, and propose a refactoring strategy that uses CRSS to improve the package design quality. By investigating the distribution of CRSS values for all classes in a system, it is possible to identify whether the relationships among the classes preclude them from a “good partitioning”. The notion of “good partitioning” is measured by how package design affects software quality attributes, like deployability, understandability, reusability, and testability. A good package design can be quantified by the manageable size, stand-alone, cohesion, and encapsulation principles.

Package dependencies are aggregated at the class (compilation) abstraction level. Thus, the distribution of the CRSS values of the classes in the whole system can be effectively used to understand its package formation problems. The CRSS metric is computed from the Class Dependency Graph (CDG). The shape of the CRSS distribution provides information about the underlying Package Dependency Graph (PDG). Large CRSS values of classes show a symptom of tall or cyclic PDGs and cannot be easily separated to stand-alone and of manageable sizes unless the class relationships are refactored. An example is the Azureus application in Fig. 1. It has nearly 1900 top-level class files. About 1000 of these class files transitively depend on 1300-1500 other classes, while about 900 of the classes transitively depend on 1-100 classes.

A refactoring strategy based on the dependency inversion principle [22] and a registry of singletons [23], is proposed to decouple classes and reduce CRSS values for systems with large CRSS values. This strategy is applied by extracting interfaces from 10 identified candidates. The candidates are classes widely referenced and with high CRSS values. The result after the 10th refactoring showed only 400 classes to have CRSS value of 1300+, and nearly 1300 classes transitively depended on less than 100 other classes.

B. Minimum Feedback Edge Set (mFES) and CRSS Metrics

In graph theory [24], strongly connected components (SCC) also known as a cyclic dependency graph in a directed graph \( G = (V, E) \) is a maximal set of vertices \( C \subseteq V \) such that for every pair of vertices \( u \) and \( v \) in \( C \), both are reachable from each other. An SCC can consist of several directed cyclic graphs as shown in Fig. 2 where one SCC contains two different cyclic subgraphs \((A, D, C, B)\) and \((A, D, C, F, E)\). The problem to solve is to eliminate undesirable SCCs among system classes and obtain a directed acyclic graph (DAG). Finding the smallest number of edges (minimum feedback arc/edge set) whose reversal or removal can turn a SCC into a DAG is an NP-complete problem [18]. It is therefore common to employ heuristics (e.g., greedy cycle removal) [18].

Sometimes, the minimum feedback edge set (mFES) is not ‘small’ in many software systems. To implement mFES for cycle breaking, would involve creating several new classes. For instance, to turn the SCC in Fig. 2a would require creating a new component \( J \) (Fig. 2b) to break the edge between \( D \) and \( C \). Arguably, this edge \((D \rightarrow C)\) can be reversed. In reality, however, edges between classes cannot just be reversed as they involve much more complex interactions. The mFES for Azureus 2.3.0.2 using the “greedy cycle removal” algorithm [18], gave 211 edges that should be treated (removed/reversed) to turn the SCC with 804 classes and 4275 edges to a DAG. A challenge is the need to create many new classes or interfaces to break the SCC. More challenging is the fact that not all the suggested edges in the mFES could be treated, as they represent relationships considered as strong coupling (e.g., an edge between a class and its abstract type).

In the example of Azureus 2.3.0.2 above, by utilizing the interfaces of 10 identified classes as candidates (with high CRSS and incoming dependencies), the SCC with 804 classes could be reduced to 253 (nearly 68% reduction). This motivated us to consider the CRSS metric before the mFES metric, when seeking to perform cycle breaking. By using the CRSS metric as an objective function, we do not only refactor...
classes in complex SCCs and their neighborhoods, but we also create a decoupled system that fits the discussions of manageable sizes and standalone properties of package design.

C. A New Metric based on CRSS

Following the discussion above, when an interface is introduced for decoupling, the extent that the CRSS values of the clients can be reduced, may be based on the CRSS values of the utilized interfaces of the candidates. The reason is that the interface would only depend on the types declared in the signatures of the published methods of its implementation. We establish that one candidate might be better than the other because their methods’ signatures are not tightly coupled to different concrete classes. This can be done by inspecting the CRSS values of both the extracted interface and the implementation of the candidate. If the CRSS values of the implementation and its interface are pretty much the same, we say that this may be a non-optimal refactoring point. Essentially, there may not be any reduction immediately in the transitive coupling but rather an increase in coupling because of this refactoring.

```java
public enum SystemProperties {
    INSTANCE; //refactored to enum as a singleton
    public String getUserPath() {
        LGLogger.log(...);
    }
    public String getApplicationPath() {
        ...
    }
    public boolean isJavaWebStartInstance() {
        ...
    }
    public String getEnvironmentalVariable(String _var) {
        LGLogger.log(...);
    }
}

public enum TOTorrentFactory {
    INSTANCE; //refactored to enum as a singleton
    ...
    TOTorrent deserialiseFromBEncodedFile(File file) throws TOTorrentException {
        return (new TOTorrentDeserialiseImpl(file));
    }
    ...
}
```

Listing 1. Opportunities in relation to the CRSS metric

```java
public interface ISystemProperties {
    public abstract String getUserPath();
    public abstract String getApplicationPath();
    public abstract boolean isJavaWebStartInstance();
    public abstract String getEnvironmentalVariable(String _var);
}

public interface ITOTorrentFactory {
    ...
    public abstract TOTorrent deserialiseFromBEncodedFile(File file)
        throws TOTorrentException;
    ...
}
```

Listing 2. Default extracted interfaces of SystemProperties and TOTorrentFactory

We demonstrate this concept with the following real world examples: `org.gudy.azureus2.core3.util.SystemProperties` and `org.gudy.azureus2.core3.torrent.TOTorrentFactory` with 15 and 16 incoming dependencies respectively in Azureus 2.3.0.2 (Listing 1): `SystemProperties` depends on `LGLogger` in the method bodies `getUserPath` and `getEnvironmentalVariable`. Similarly, `TOTorrentFactory` depends on the `TOTorrent` and `TOTorrentDeserialiseImpl` as return types and on `TOTorrentException` as an exception type. `LGLogger`, `TOTorrent`, and `TOTorrentDeserialiseImpl` have CRSS of 1376. `TOTorrentException` has a CRSS of 1. An `Extract interface` performed on `SystemProperties` and `TOTorrentFactory` produce `ISystemProperties` and `ITOTorrentFactory` (see Listing 2).

An observation of the two interfaces shows that `ITOTorrentFactory` still has dependencies on `TOTorrent` and `TOTorrentException` and thus has a transitive dependency of 1376. This value is the same as the maximum CRSS (1376) of its implementation, `TOTorrentFactory`. Conversely, the interface `ISystemProperties` contains no dependencies on any concrete implementations from `SystemProperties` and therefore has a maximum CRSS of 1 while `SystemProperties`, has a CRSS of 1376. The refactoring with `TOTorrentFactory` as candidate produced CRSS values of at least 1364 for all the 16 incoming dependencies and the SCC to 794 classes.

Using this background, we determine a new metric named interface-CRSS reduction rate that is based on the difference between the CRSS value of a class and its interface. Formally, we define the interface-CRSS reduction rate for a class X as:

\[
\text{IRCRRS}(X) = \frac{\text{CRSS}(X) - \text{CRSS}(IX)}{\text{CRSS}(X)}
\]

Where:

- IRCRSS (X) is the class reachability set size (CRSS) reduction rate for the interface of X.
- CRSS (X) is the class reachability set size of X.
- CRSS (IX) is the class reachability set size of the interface of X (IX).

The IRCRSS of X gives the likely rate at which the CRSS value of a client Y that depends on X would be reduced if it depends on the interface of X (i.e. IX). The value of IRCRSS ranges from 0 to 1. A value of zero implies no reduction in the CRSS value when the dependency of Y is changed from X to IX, while a value of 1 implies a possible 100% reduction.

D. Strategies for Edge Breaking between a Source and a Target Type

The dependency between two program classes can be represented as: `source` depends on `target` (source → target), where the `target` class is used within the `source` class. We have used these notations “source” and “target” in the following presentation. In addition, we have used standard refactoring notations [25] such as `Extract interface`, `Move method`, `Move field`, `Encapsulate field` and so on in our presentation.

1) Type Generalization: Type generalization involves declaring a variable with its abstraction (interface or abstract class). This is considered a good programming practice [25]. In general, when an interface of an implementation type is
introduced, it should be utilized by all of its clients wherever possible [26]. However, studies show that interface types are sparingly used in software development despite their potential [26, 27]. Type generalization can be used to break dependency between a source type and a target type when the relationship is an aggregation (has-a) and not a composition (part-of).

2) Registry (Service Locator): Two cases are considered. First, the target’s constructor is explicitly invoked through a “new” keyword in the source class (part-of). This type requires that the source use a new object of the target class. Service Locator/Registry of Prototypes pattern [23, 28] can be used to break this dependency. Second, a utility class (that contains only static members) may sometimes have high incoming dependencies and be a hub for big and complex SCCs. It might be needful to refactor this class into a singleton (see Listing 1) and its static methods to instance-side methods to break such complex SCCs by using the Registry of Singletons pattern [20].

The target classes for Java applications can be instantiated using the ServiceLoader1 or in the entry class of the application [20]. For C# applications, the lightweight injection container called Unity2 can be used to configure target classes. The refactoring may sometimes require some extra modifications to the target’s class. As an example, consider breaking the edge “org.yccheok.jstock.guiMainFrame→org.yccheok.jstock.engine.StockInfoDatabase” in JStock 1.0.7r where the target StockInfoDatabase uses parameters in its constructor. The ServiceLoader enforces that provider classes have zero-argument constructor so that they can be instantiated during loading. One trade-off is to modify the target as shown in Listing 3. A new empty constructor and a new public method (e.g. processParamStockInfoDatabase()) are created in the class StockInfoDatabase. The public method takes the parameters and the body of the first constructor. A reference is created from the non-empty constructor to this method. This way, the code is not broken and refactored clients can request the instance of the target and pass the parameters through the public method as shown in Listing 4.

3) Static Final (Read-only) Field (Copy field): High coupling between the source and a target class could occur because of static final field invocation or static field that is used as read-only (final). An example is a case in Azureus v2.3.0.2, where “BackGroundGraphic” class depends on “MainWindow” class because it uses a static Color white. The bizarre decision here is that while other color types were defined and used in the BackGroundGraphic class, the developer simply referenced the color field “white” from the MainWindow class rather than defining it in the BackGroundGraphic class. This has been refactored in the latest version by moving “Color white” to the BackGroundGraphic class. The refactoring approach here is toCopy field from the target to the source class. This makes sense because the value of such a final field would not change or become updated.

```java
public class StockInfoDatabase implements IStockInfoDatabase {
    public StockInfoDatabase(List<Stock> stocks) {
        processParamStockInfoDatabase(stocks);
    }
    ...
    public StockInfoDatabase() {
        ...
    }
    public void processParamStockInfoDatabase(List<Stock> stocks) {
        ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
        reader = readWriteLock.readLock();
        writer = readWriteLock.writeLock();
        this.init(stocks);
    }
    ...
}
```

Listing 3. Modifying target with non-empty constructors

/* Before refactoring */
public classMainFrame extends javax.swing.JFrame {
    ...
    private StockInfoDatabase loadStockInfoDatabaseFromCSV(Country country) {
        ...
        return new StockInfoDatabase(stocks);
    }
    ...
}

/* After refactoring */
public classMainFrame extends javax.swing.JFrame {
    private IStockInfoDatabase loadStockInfoDatabaseFromCSV(Country country) {
        ...
        IStockInfoDatabase sid = ServiceLoaderUtil.INSTANCE.loadService(IStockInfoDatabase.class, IStockInfoDatabase.ID);
        sid.processParamStockInfoDatabase(stocks);
        return sid;
    }
    ...
}

Listing 4. Refactoring target with parameters in its constructor

4) Encapsulate Static Field: A source class can use a target class through static field invocation. Encapsulate field [25] and Extract interface with Registry/ServiceLoader refactoring can be applied to break this dependency. The static field may be declared private in the target and assigned to an auxiliary instance field. A getter method is declared for the instance field and declared in the target’s interface. The source can then access the static field through the getter method of the instance field.

5) Inline Static Method: A source can depend on a target through the invocation of the target’s static method. To inline a static method would imply moving the method from the target to the source and creating a delegate in the target’s method to the moved method in the source [29]. Essentially, the dependency is reversed. This is similar to a situation where Move method [25] is applied to break a dependency, however, this does not involve reversing the dependency. Moving a method body can create some recursive actions and higher reachability size. We therefore propose an Extract

---

1 https://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html
2 https://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html
Interface with Registry of singletons refactoring when a target class has incoming dependencies that is more than one and static method inline when it has only one.

```java
public class A extends B implements C {
    private D d;
    private static E e = new E();
    public F meth(G g, D d) throws H {
        this.d = d;
        P.logtQ(Status, R, ID); // *Assume ID is a final variable in class R*/
        return (F) g.typeOfF();
    }
}
```

Listing 5. Example of dependency types

E. Dependency Types and Refactoring Strategy

A dependency can be formed in different ways between a source class and a target class [30]. We illustrate this with the following example code snippet in Listing 5. In this snippet, class A depends on classes B, C, D, E, F, G, H, P, Q, and R. Table 1 lists the default strategy for the dependency types.

F. Related Cycle-Breaking Studies and Tools

Graph transformation has been extensively applied in software engineering and notably in code-level refactoring activities [31, 32]. The type of graph manipulation we have employed in this study does not demand detailed graph formalism since we are only interested in removing or adding single edges in a graph. We therefore limit our discussion to other studies devoted in this manner to cycle-breaking refactoring.

Dietrich et al. [30] identified high impact edges from the program dependency graph by assigning weights to edges based on the number of anti-patterns they are involved with. Their results on the graph model demonstrated that many anti-patterns (e.g., dependency cycles at the package level) could be removed by removing such high impact edges. Shah et al. [29] implemented an automated refactoring on these edges using various refactoring techniques. Their results show that certain edges are removable, while removing certain edges would introduce errors.

Table 1: Dependency Types and Default Refactoring Strategy

<table>
<thead>
<tr>
<th>Dependency type</th>
<th>Example</th>
<th>Default Refactoring Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable declaration</td>
<td>A uses D, E</td>
<td>Extract interface</td>
</tr>
<tr>
<td>Variable declaration</td>
<td>A uses E</td>
<td>Extract interface + ServiceLoader</td>
</tr>
<tr>
<td>Method return type</td>
<td>A uses F</td>
<td>Extract interface</td>
</tr>
<tr>
<td>Method parameter type</td>
<td>A uses G, D</td>
<td>Extract interface</td>
</tr>
<tr>
<td>Method exception type</td>
<td>A uses H</td>
<td>Extract interface (abstract class)</td>
</tr>
<tr>
<td>Static method invocation</td>
<td>A uses P</td>
<td>Inline Method or Extract interface + Registry</td>
</tr>
<tr>
<td>Static field invocation</td>
<td>A uses Q</td>
<td>Encapsulate + Extract interface + Registry</td>
</tr>
<tr>
<td>Static final field</td>
<td>A uses R</td>
<td>Copy field or Move field to Interface (Extract interface)</td>
</tr>
<tr>
<td>Constructor invocation</td>
<td>A uses E</td>
<td>Extract interface + ServiceLoader</td>
</tr>
<tr>
<td>Super type</td>
<td>A uses B</td>
<td>None</td>
</tr>
<tr>
<td>Interface type</td>
<td>A uses C</td>
<td>None</td>
</tr>
<tr>
<td>Others (e.g. casting)</td>
<td>A uses F</td>
<td>Extract interface</td>
</tr>
</tbody>
</table>

Laval and Ducasse [33] implemented an enriched dependency structural matrix (eDSM) to detect dependency cycles between packages. They use contextual information, e.g. types of relationships between the coupled components and the proportion of referencing classes in the client package. The tool reports actions to be performed to remove detected dependency cycles.

Several other tools have been proposed to detect cycles. For instance, JDepend4, NDepend5, JooJ [19], Dependometer6, Classycle7, STAN8, Jepsens [34], PASTA [35], Lattix9, and Structure10110. Of the aforementioned approaches and tools, only the work of Laval and Ducasse [33] has close similarity to ours in the sense that they used context data to propose refactoring actions. However, it differs in focus because we have considered refactoring cycles at the class granularity level.

III. IMPLEMENTATION

We have built a CB-DSS in Java (publicly accessible at: https://bitbucket.org/ootos/j-guiestructurer and used/extended the Jepsens-beel by Melton11 to collect dependency data. The dependency data is collected from the bytecode of Java classes using the Apache Byte Code Engineering Library12 (BCEL). We are interested in top-level classes (compilation units), since they represent maintenance units. Therefore, the dependencies of nested classes are aggregated to their top-level classes. A MSc student has integrated the CB-DSS into a Visual Studio plugin (Accessible at https://bitbucket.org/ootos/c-sharprestructurer). The simple model diagram for the CB-DSS is shown in Fig 3. There are seven major components of the model: (1) decision support table 1 - DSTable1, (2) decision support table 2 – DSTable2, (3) the dependency types – UsageType, (4) refactoring strategy - Strategy, and (5) RefactoringSimulation, (6) System Restructuring, and (7) Cycle breaking.

A. DSTable 1

This table implements the IRCRSS metric for each class in the application. IRCRSS value ranges between 0 and 1. The table is used as a look up table to decide the choice of the best class as candidate for refactoring. The selection mechanism from DSTable-1 is driven by IRCRSS, high incoming dependencies (FAN-IN), high CRSS and high SCC values. The SCC value represents the number of classes involved in the same SCC with the candidate. The list of candidates is selected using the following rules:

1. The candidate must fall within the specified topKs positions for all the three measures (FAN-IN, CRSS, SCC)
2. The IRCRSS value of the candidate must be equal or greater than the specified value by the user

4 http://clarkware.com/software/JDepend.html
5 http://www.ndepend.com
6 http://source.valtech.com/display/dpm/Dependometer
7 http://classycle.sourceforge.net
8 http://stan4j.com
9 http://lattix.com
10 http://structure101.com/products
11 https://www.cs.auckland.ac.nz/~hayden/software.htm
12 http://commons.apache.org/proper/commons-bcel/
3. The candidate must not be an interface or an abstract class (since these types cannot be instantiated).

Next, the selected topK classes are sorted by using four attributes: FAN-IN, CRSS, SCC, and STATIC. The STATIC variable implies that all the class members are static. In some applications, static members are usually widely referenced and are potential hub for large SCCs. The sorting is implemented by selecting the principal attribute. The algorithm then sorts on the principal attribute and two other attributes. The possible combination of sorting is as follows (the bold and underlined attribute denotes the principal sort attribute):

1. STATIC, FAN-IN, CRSS
2. CRSS, FAN-IN, SCC
3. FAN-IN, CRSS, SCC
4. SCC, FAN-IN, CRSS

We have decided on these sorting combinations by comparing results from several random experiments. The sorting combination orders produced the best refactoring results on different systems.

B. DSTable 2

This table stores context data and computed refactoring decisions for each edge (source → target). The DSTable-2 is computed by using the UsageType and the default refactoring Strategy as described in Table I. The DSTable-2 serves as a look up directory to select the refactoring decision for each suggested edge during refactoring.

C. System Restructuring

The approach is to begin every refactoring with System restructuring with fitness function as CRSS and SCC. This refactoring focuses on decoupling the entire software structure and it uses the DSTable-1 to determine the classes that are candidates for refactoring. A pre-selected number (N) of refactoring iteration and a combination of tuning and sorting parameters are presented to the RefactoringSimulation module. For each refactoring, the system selects the best class as candidate from DSTable-1 and then simulates the refactoring of all classes that depend on the candidate. The refactoring strategy to break each edge (class → candidate) is selected from DSTable-2. At the end of each refactoring, the fitness values are computed and a list of refactoring actions are generated. The general refactoring is performed as follows:

a. Create the interface (or abstract class) for the selected candidate (Class)
b. Create an edge between the candidate and its interface
c. Create the registry class and the respective edges from the candidate and the main class to the registry
d. Move all published dependencies of the candidate to its newly created interface using the refactoring strategy
e. Update all relevant relationships and edges
f. Compute the SCC and fitness values
g. Update decision tables

D. Cycle-Breaking Refactoring

This is used to further resolve SCCs that are not refactored during the “System restructuring”. It is driven by selecting the SCC of interest and then activating a “greedy cycle removal” algorithm [18] to determine the minimum feedback edge set (mFES). The mFES is passed to the RefactoringSimulation module. The refactoring strategies for each edge are looked up from DSTable-2. The refactoring for this edge is then simulated based on the returned refactoring strategy.

IV. VALIDATION

We report on four case studies to evaluate the accuracy of the CB-DSS. In the first case study, we performed refactoring on Azureus 2.3.0.2 using ten candidates. Next, we refactored JStock using ten candidates. The refactoring for the above two case studies were performed by one of the authors. The third case study is VidCoder, an open source, C# application. A MSc student has performed the refactoring of six candidates for this case study. The fourth case study (commApp) is an industrial Smart Grid application developed with C#. The company’s software maintenance engineer has performed the actual refactoring of three candidates. For space reasons, the properties of the selected applications can be found here: http://www.idi.ntnu.no/~tosindo/resources/systems.pdf

We summarized the results of the validations in Table II. As shown in the Table, the fitness values, mean (CRSS), and max (SCC) for manual refactoring are close to the fitness values of the CB-DSS. For Azureus, the results of the CB-DSS and the actual refactoring are nearly the same. For JStock, VidCoder and commApp, the results of manual refactoring are modest and are reasonably comparable to the result of the CB-DSS. In the case of VidCoder, the differences could be due to the fact that 5 edges out of the 9 proposed were not refactored. The reason is that the developer used lightweight injection container (Unity) in Visual Studio instead of defining a custom registry class. This is similar to JStock, where ServiceLoader fits better than a customized registry. This is positive as the developers have control during the refactoring activities. For commApp, some of the changes made by the maintenance engineer involved additional refactoring such as splitting a class into two and thereby increasing both the number of classes and relationships. These would affect the fitness values.
Overall, the results show that it is possible to use the CB-DSS as a decision support tool for planning refactoring activities.

V. EVALUATION AND DISCUSSION

We have used 15 software applications to answer the research questions. They are: commApp, Azureus (Vuze), Jstock, VidCoder, Hibernate, Openproj, Jxplorer, Megamek, Weka, SomToolBox, GanttProject, Squirrel, Jstock, VidCoder, Hibernate, Openproj, Jxplorer, Megamek, and Logism. Apart from commApp and VidCoder, the remaining applications are Java applications selected from SourceForge based on their popularity (four to five stars rating) and at least 500 downloads per week. All the applications are driven through a user interface, since candidate classes that are singletons need to be instantiated from the start of the applications. This is easier to figure out for GUI applications. For properties of selected applications see the link in Section IV.

A. RQ1: Will Tuning with the IRCRSS Metric Produce Refactoring Fitness Value that is Lower than Refactoring without?

1) Case Study-Azureus 2.3.0.2 [20]: Our goal is to find out whether using the CB-DSS with the IRCRSS metric would improve the result when compared to the manually selected candidates used for refactoring by Melton & Tempero [20]. We found that Azureus 2.3.0.2 fits the version they analyzed because it has approximately the same value of CRSS. Both the versions before and after have a wide CRSS range gap to the reported value.

2) Approach: We have simulated the refactoring with the manually selected candidates by Melton & Tempero [20]. In this simulation, we turned off the selection algorithm and allow the CB-DSS to iterate through the selected candidates as presented by the authors. Next, we allow the CB-DSS to automatically select candidates for refactoring using a combination of tuning and sorting parameters. We perform different simulations by varying the percentiles (topKs) of the sorting parameters (SSC, CRSS and FAN-IN) and turning on/off the IRCRSS metric.

3) Results and Discussion: Table III lists the candidates selected by our approach vs. the ones reported in Melton & Tempero [20]. The last candidate (TorrentImpl) is a non-singleton class. Fig. 4 shows that at the 7th refactoring, the selection by the CB-DSS has better results than the 10th refactoring with the manual mode. Furthermore, it indicates that using a CB-DSS with IRCRSS metric can significantly improve the refactoring results. The result from Table IV shows there is more reduction in the CRSS and SCC values when IRCRSS is used and in combination with optimal values for CRSS, FANIN, and SCC. As listed in Table IV, the max SCC after refactoring drops from 804 to 253 while for their selection it drops to 333. Furthermore, Their reported number for frequency of classes with CRSS of 1000+ is 400, which is modestly comparable to the simulated number of 427. Using automatic selection produced a better result of 348 classes. In all cases, the results from using the selection parameters from the CB-DSS produced better results but notably, with the IRCRSS metric. We performed a statistical test using Wilcoxon rank sum test [36] to determine whether the fitness values (CRSS) by using the IRCRSS metric is statistically and significantly lower than the fitness values without (i.e. \( H_0 = \text{The fitness of refactoring with IRCRSS is significantly higher than the fitness without IRCRSS}. \) We performed 15 refactoring (see column 6 of Table IV). The mean CRSS from both groups are recorded separately (group1 = 360.7 and group2 = 315.11) and are tested for significant difference. The

**TABLE II. RESULTS OF VALIDATION**

<table>
<thead>
<tr>
<th>Fitness</th>
<th>Azureus (N=10)</th>
<th>JStock (N=10)</th>
<th>VidCoder (N=6)</th>
<th>commApp (N=3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR</td>
<td>AR (Actual)</td>
<td>BR (Actual)</td>
<td>BR (Actual)</td>
<td>BR (Actual)</td>
</tr>
<tr>
<td></td>
<td>(RD -24hrs)</td>
<td>(RD -4hrs)</td>
<td>(RD -10hrs)</td>
<td>(RD -NR)</td>
</tr>
<tr>
<td>Mean (CRSS)</td>
<td>703.67</td>
<td>154.6</td>
<td>17.22</td>
<td>115.06</td>
</tr>
<tr>
<td>StdDev (CRSS)</td>
<td>295.46</td>
<td>107.27</td>
<td>10.87</td>
<td>115.88</td>
</tr>
<tr>
<td>Mean (SCC)</td>
<td>684.88</td>
<td>37.63</td>
<td>6.4</td>
<td>9.57</td>
</tr>
<tr>
<td>Max (SCC)</td>
<td>22.82</td>
<td>121.03</td>
<td>24.91</td>
<td>9.57</td>
</tr>
</tbody>
</table>

\( N = \text{number of selected classes}; BR = \text{Before Refactoring}; AR = \text{After Refactoring}; RD = \text{Refactoring Duration}; NR = \text{Not Reported} \)

**TABLE III. CB-DSS VS MANUAL CANDIDATES SELECTION**

<table>
<thead>
<tr>
<th>Order</th>
<th>Candidates by Melton &amp; Tempero [20]</th>
<th>Candidates (CB-DSS) - IRCRSS= True</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>org.gudy.azureus2.core3.logging.Log4j.Logger</td>
<td>org.gudy.azureus2.core3.util.Debug</td>
</tr>
<tr>
<td>2</td>
<td>org.gudy.azureus2.core3.config.COCConfigurationManager</td>
<td>org.gudy.azureus2.core3.util.Debug</td>
</tr>
<tr>
<td>3</td>
<td>org.gudy.azureus2.core3.config.COCConfigurationManager</td>
<td>org.gudy.azureus2.core3.util.Debug</td>
</tr>
<tr>
<td>4</td>
<td>org.gudy.azureus2.core3.util.FileUtil</td>
<td>org.gudy.azureus2.ui.swt.Messages</td>
</tr>
<tr>
<td>5</td>
<td>org.gudy.azureus2.platform.PlatformManager</td>
<td>org.gudy.azureus2.core3.util.FileUtil</td>
</tr>
<tr>
<td>7</td>
<td>org.gudy.azureus2.core3.util.TorrentUtils</td>
<td>org.gudy.azureus2.core3.util.TorrentUtils</td>
</tr>
<tr>
<td>8</td>
<td>org.gudy.azureus2.core3.util.LocalCacheUtil</td>
<td>org.gudy.azureus2.ui.swt.components.shell.ShellFactory</td>
</tr>
<tr>
<td>9</td>
<td>org.gudy.azureus2.core3.util.DisplayFormatters</td>
<td>org.gudy.azureus2.ui.swt.mainwindow.Colors</td>
</tr>
<tr>
<td>10</td>
<td>org.gudy.azureus2.core3.util.DirectByteBufferPool</td>
<td>org.gudy.azureus2.pluginsimpl.local.torrent.TorrentImpl</td>
</tr>
</tbody>
</table>
result of the test is statistically significant at alpha = 0.05 (with p-value=0.011). The pooled standard deviation [37] from both groups (sp) is calculated as 68.2 giving an effect size of (360.7 - 315.11)/68.2 = 0.66, which is a moderate (0.378 – 1.000) effect [37]. We thus reject the null hypothesis and conclude that applying the IRCRSS metric gives a significantly lower (better) result for this application.

4) Manual Refactoring: The CB-DSS proposed nine singleton classes and one non-singleton class as candidates. We then manually refactored the code by using the actions reported from the system. There are instances where an interface already exists for the candidate. E.g. TorrentImpl class that implements Torrent (an interface). To refactor the proposed edges, some new methods must be declared in the old interface or in the proposed interface. A maintenance practice is to introduce a new interface that extends Torrent and add those methods in the new interface. This is a kind of interface upgrade. By doing so, we maintain a downward compatibility of the old interface (Torrent) and do not break the code. Otherwise, declaring new methods in the interface, would force other children of Torrent to implement them.

The new Java SDK version 8 however makes it possible to declare such new methods with empty or default implementations in Torrent. This produces the same results, as the old classes are not forced to implement the new methods. This is significant because it simplifies maintenance and refactoring activities. Rather than defining a new interface because of additional functions, it is now possible to define new contracts as default methods and without the burden of forceful implementation of the new methods by all children. Arguably, this feature can also be a shortfall. First, it would be hard for children to be aware of declared methods in the interface because it is not required anymore. Second, in terms of maintenance and upgrade, it would be hard to keep track of changes (extensions, etc.) that have been made as the system evolves.

B. RQ2: Will every Application Benefit from the Improvement?

To answer this question, we simulate refactoring on the fifteen applications. The results in Table V demonstrate that it is possible to benefit from the IRCRSS metric in several applications. Cases such as Logisim, JXplorer, Azureus, OpenRocket, and Hibernate, have relatively high SCC reductions when the measurement from IRCRSS metric is applied. However, in a few applications (e.g. Megamek, VidCoder and Squirrel-sql), there is no difference in the results with the IRCRSS metric. In total, there are improvements in the fitness values of 12 out of the 15 applications. We can conclude that the IRCRSS metric can improve the code structure in the majority of the cases. The metric (IRCSS) provides useful information to clients/services that are being coupled. The IRCRSS value is zero or nearly zero when the published types of a class are tightly coupled with other classes (in most cases, concrete classes and not interfaces). This has implications for maintenance and testing. A class/service that is heavily reused and is tightly coupled in its published members would be difficult to maintain and test.

C. RQ3: Is the Number of Refactoring Reduced with the IRCRSS Metric?

To answer this question, we have categorized the restructuring effort into two. 1). The rate of reduction in the number of edges (source → target) that the CB-DSS proposes for refactoring with and without IRCRSS metric (i.e. %ReduceProposedEdges = 100 * #ProposedEdgesIRCSS=\text{false} - #ProposedEdgesIRCSS=\text{true} / #ProposedEdgesIRCSS=\text{true}). 2). The rate of reduction in the class edges created in the applications after refactoring is simulated with and without IRCRSS (i.e. %ReduceClassEdges = 100 * #EdgeIRCSS=\text{false} - #EdgeIRCSS=\text{true} / #EdgeIRCSS=\text{true}). As shown (Table V), when IRCRSS metric is used, the CB-DSS is able to reduce the refactoring efforts. In six cases, there are significant reductions in the number of edges proposed for refactoring. E.g in Logisim and Hibernate; the proposed edges are reduced by 63.2% and 66% respectively when IRCRSS metric is used. This is noteworthy because refactoring fewer edges would translate to a reduction in refactoring efforts. Also, the total number of edges created after refactoring using IRCRSS reduced reasonably in some applications (e.g. Logisim, OpenRocket, Hibernate and JStock). In other words, fewer relationships are created in the code when IRCRSS is applied. When the signatures of published methods of a class are tightly coupled to other concrete classes, it could result in more relationships being created during restructuring. It is positive to have fewer classes and relationships during restructuring. The fewer the number of edges that exist in an application, the better it would be to reason about the coupling situation in the application.

---

13http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html
D. Qualitative Evaluation

We have carried out an interview with the software maintenance engineer of our industrial partner to determine the usefulness and usability of the CB-DSS. We drafted questions that covered four areas namely; user experience, compatibility, impact and functionalities. For user experience, the questions focused on ease of use, ease to learn quickly, clarity and understanding of system’s functionalities and future use. Under compatibility, whether the CB-DSS fits well with the work practices. In terms of impact, whether the approach is useful for refactoring complex structural part of the code, whether the approach is able to identify good candidates for refactoring and whether the actual code structure improved after refactoring.

E. Summary of Respondent’s Views

1) User Experience: The respondent views are that the tool is easy to use and can be learned quickly and individually with a proper help file. It does not take time to learn how to use it. The functionalities are clear and understandable and can motivate maintenance practices in the company. The only challenging part is how to choose the parameters for the algorithm.

2) Compatibility: The tool will fit maintenance work practices and using the tool regularly can help developers to have a picture of the code’s structure and keep an eye on maintainability.

3) Impact: Respondent states that at present, large parts of their code are not that maintainable, looking at the code you can spot some areas that should be changed (e.g. excessively large and coupled classes), some bad coding practices and so on. The tool will stimulate actions to correct some of these problems. In addition, code reuse would be easier.

4) Functionalities: The approach is able to identify good targets for refactoring and the code structure improved after refactoring. The respondent prefers a simulation tool rather than an automated tool for this large scale restructuring. This agrees with the feedback from three other developers in the same company during our presentation sessions.

VI. Threats and Limitations

The CB-DSS is implemented on top of the Jepend tool that uses BCEL to collect class dependency data. Java’s specification uses type erasure, therefore, information about type parameters of generic types are not available in the Java bytecode. In addition, we cannot identify dependencies created by the use of reflection. This is a common limitation of static analysis and can reduce the accuracy of the CB-DSS.

The refactoring result by CB-DSS is sometimes an approximation due to the use of a new and generic interface. A candidate may have an existing interface that only needs to be upgraded during refactoring. The CB-DSS does not take this into consideration during its computation and simulation. Additionally, we can not claim to have covered all dependency possibilities. New cases not treated in the case studies could have edges that cannot be refactored or bring variations in the simulated vs. actual refactoring results. Lastly, the qualitative validation is based on a single user view and it is thus limited.

VII. Conclusion

We have implemented a new metric, IRCRSS and a cycle breaking decision support system (CB-DSS) to resolve class dependency cycles and improve the overall code structure. The evaluation of the CB-DSS proved that it is useful and implementable in many cases in real life systems. Our contributions in this work are therefore as follows:

1. Significant improvement on the strategy employed in Melton and Tempero [20] by introducing a new metric IRCRSS, to identify CRSS reduction between an interface and its implementation. In this way, it is possible to improve the structural quality of the code and reduce the refactoring efforts

2. A cycle breaking system that proposes refactoring actions. These actions are fine-tuned for each proposed edge (source → target), with details such as the strategy and action to break the edge, and the actual code location (method or field) where the strategy should be applied in the source class

3. We demonstrate the validity of the CB-DSS by the manual refactoring on industrial and open source systems.
REFERENCES


