applied sciences

MDPI

Article

An External Client-Based Approach for the Extract Class Refactoring: A Theoretical Model and an Empirical Approach

Musaad Alzahrani * and Saad Alqithami *

Department of Computer Science, Albaha University, Albaha 65799, Saudi Arabia
* Correspondence: malzahr@bu.edu.sa (M.A.); salqithami@bu.edu.sa (S.A.)

Received: 22 July 2020; Accepted: 26 August 2020; Published: 31 August 2020

Abstract: A commonly observed ambiguity of a class is simply a reflection of multiple methods' implementation within an individual class. The process of Extract Class refactoring is, therefore, used to separate the different responsibilities of a class into different classes. A major limitation in existing approaches of the Extract Class refactoring is based on factors that are internal to the class, i.e., structural and semantic relationships between methods, in order to identify and separate the responsibilities of the class which are inadequate in many cases. Thus, we propose a novel approach that exploits the clients of the class to support the Extract Class refactoring. The importance of this approach lies in its usefulness to support existing approaches since it involves factors external to the class, i.e., the clients. Moreover, an extensive empirical evaluation is presented to support the proposed method through the utilization of real classes selected from two open source systems. The result shows the potential of our proposed approach and usefulness that leads to an improvement in the quality of the considered classes.

Keywords: software maintenance; extract class refactoring; client-based similarity; class cohesion

1. Introduction

One of the most common design issues in object-oriented systems is having a class with many responsibilities. During the maintenance and evolution activities of a system, new responsibilities may need to be added to the system. Due to time limits, software developers usually feel it is not necessary to create a separate class for a new responsibility and that responsibility can be added to one of the existing classes [1]. After several cycles of maintenance and evolution, some classes in the system will end up having many responsibilities which will increase the maintenance cost of the system because a class with many responsibilities requires more effort and time to understand and maintain. In addition, a class with many responsibilities has many reasons to change because each responsibility the class has is an axis of change [2]. Such a class needs to be refactored in order to improve its understandability and maintainability. The refactoring technique that is usually applied to overcome this issue is called Extract Class refactoring. It refers to the process of splitting a class that has many responsibilities into a set of smaller classes, each of which has one responsibility and one reason to change [1].

Manually performing the process of the Extract Class refactoring costs much time and effort. Thus, several approaches (see, e.g., in [3–5]) have been introduced in the literature to automate and support the Extract Class refactoring. These approaches conduct the Extract Class refactoring based on factors internal to the class to be refactored such as structural and semantic dependency between the methods of the class. However, this internal view of the class is inadequate in many cases to automatically determine the responsibilities of the class and to highlight potential reasons that can

cause changes to the class. For instance, consider the class `Circle` shown in Figure 1 that has four attributes and nine methods. The class has two client classes: `GeometryApp` and `GraphicsApp`. We refer to these two classes as clients of the class `Circle` because they use methods in the class (see Section 3). The class `GeometryApp` performs some computational geometry. Thus, it uses the methods that provide geometric computations (e.g., *area*()) in the class `Circle`, but never uses the methods that provide drawing services (e.g., *draw*()). The other client class (i.e., `GraphicsApp`) uses the methods that provide drawing features in the class `Circle` because it draws shapes on screen including circles.

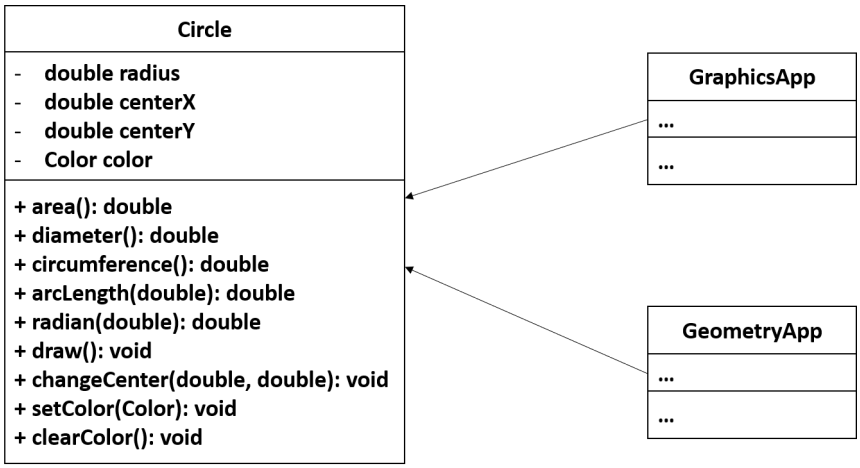

Figure 1. The class `Circle` and its clients.

It is obvious that the class `Circle` has two responsibilities: one responsibility is to perform computational geometry and the other responsibility is to perform drawings on the screen. This may lead to increasing the cost and effort of future changes of the class. Assume that the client class `Graphics` exerts a change on the method *setColor*(*Color*) in the class `Circle`. This change may affect the client class `Geometry` because it depends on the class `Circle`. Therefore, `Geometry` may need to be recompiled and retested even though it never uses the method *setColor*(*Color*). In order to address this issue, we need to perform the Extract Class refactoring by splitting these two responsibilities into two separate classes as shown in Figure 2. Existing approaches of the Extract Class refactoring may fail to suggest the refactoring solution shown in Figure 2 because they perform the Extract Class refactoring based on structural and semantic relationships between the methods of the class. The structural relationships between methods in these approaches are measured based on the shared attributes between the methods while the semantic relationships are measured based on the common vocabularies in the documents, e.g., identifiers and comments of the methods. Thus, most of the methods of the class `Circle` are—to some degree—structurally and semantically related to each other because they likely share attributes and vocabularies, e.g., radius.

To overcome the above issue, this study proposes a novel approach that performs the Extract Class refactoring based on the similarities in terms of clients between the methods of the class in question. The proposed approach can be more beneficial than the traditional refactoring techniques that consider the internal view of the class when performing the extract class refactoring. It identifies the different responsibilities of a class based on the usage of its clients. The intuition behind this is that if there are different sets of methods in the class that are used by different sets of clients than the class has different responsibilities from the perspective of its clients [6–8]. In addition, the proposed approach supports the adherence to the *Interface Segregation Principle* (ISP), which is an important object-oriented design principle that states "no client should be forced to depend on methods it does not use" [2]. Our proposed approach is not meant to replace the existing approaches of the Extract Class refactoring but to complement them because considering the structural and semantic relationships between the methods of class can be useful in many cases of the Extract Class refactoring.

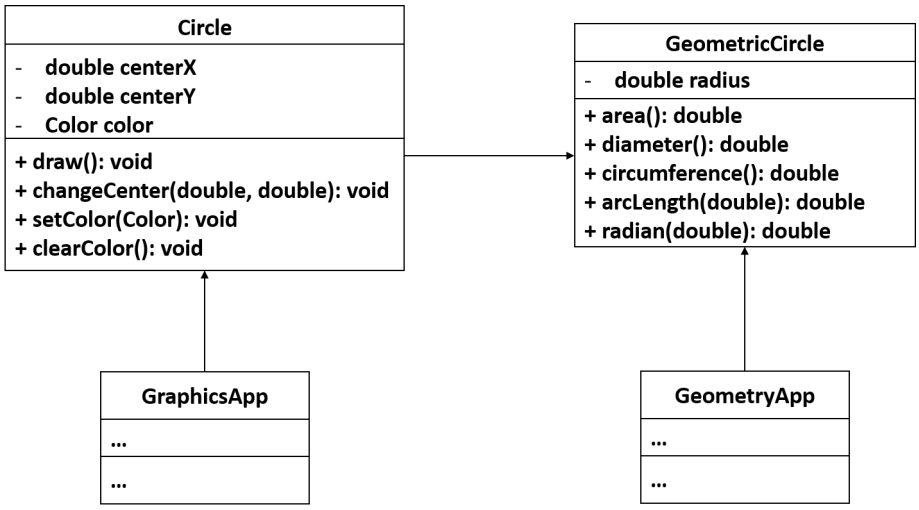

Figure 2. The Class `Circle` in Figure 1 is split into two classes: `Circle` and `GeometricCircle`.

The paper extends a previous short paper [9] that only presented an initial overview of the proposed approach without an empirical evaluation. In this paper, however, we present and extensively discuss the proposed approach for the Extract Class refactoring. In addition, we conduct an extensive empirical experimentation to evaluate the proposed approach based on real classes selected from two open source systems. The rest of the paper is organized as follows. In Section 2, we present and discuss related work. Section 3 presents the proposed approach. In Section 4, we present the empirical evaluation of the proposed approach. Section 5 gives the conclusion and future work.

2. Related Work

Martin Fowler [1] who introduced the Extract Class refactoring suggested to perform the Extract Class refactoring on a given class by first identifying each subset of data and methods that contribute to a single responsibility and next extracting them into a separate class. However, identifying the different responsibilities and identifying the data and methods of each responsibility depends on the experience of the software engineer who is performing the refactoring and it can not be done in an objective way. Therefore, many approaches have been introduced in the literature to try to support and automate the Extract Class refactoring, see, e.g., in [3–5,10–22]. In the following, we discuss and summarize the approaches that are mostly relevant to our approach.

Simon et al. [11] developed a visualization tool that supports four types of refactoring techniques including the Extract Class refactoring. The tool uses structural metrics to identify refactoring opportunities in the system in question. Marinescu [12] proposed an approach called detection strategy to support software engineers in the identification of software modules, e.g., classes, that have a particular design flaw, e.g., a class that has many responsibilities. The approach suggests to formulate metrics-based rules that can directly capture software modules that are affected by a design flaw. The process of formulating metrics-based rules for detecting a design flaw consists of four steps. First, the symptoms of the design flaw are determined, e.g., the symptoms of the design flaw God Class include high class complexity, low class cohesion, and the access of foreign data. Second, an appropriate metric is chosen to measure each symptom, e.g, $LCOM2$ [23] to measure cohesion. Third, a filtering mechanism, e.g., less than filter, is used with each metric to detect the symptom, e.g., a value of $LCOM2$ that is more than 10 is considered low class cohesion. The final step is to correlate the symptoms using AND/OR operators to detect the design flaw. Once a design flaw is detected, a refactoring opportunity exists. Similar to the work in [12], a recent work [24] proposed a metric-based detection technique for a set of bad smells (also known as design flaws or defects) including "Large Class" bad smell. The authors used size and cohesion metrics to detect large classes. They suggested four types of refactoring operations including the Extract Class refactoring to solve the bad smell Large Class.

Fokaefs et al. [3] proposed an approach for the Extract Class refactoring which employs an agglomerative clustering algorithm based on the Jaccard distance between the methods of the class to be refactored. Structural similarity between two methods of the class is used to calculate the Jaccard distance between the two methods. The higher the Jaccard distance between the two methods is, the more probably the two methods will be in the same cluster. The resulting clusters represent the Extract Class opportunities that can be identified in the class to be refactored. Similar to the work in [3], Fokaefs et al. [10] introduced a tool that can identify Extract Class refactoring opportunities in God Classes. A software engineer can select one of the suggested refactoring opportunities and the tool will automatically apply it. The tool is as an extension of the JDeodorant Eclipse plugin and it employs structural metrics for extracting a class with higher cohesion from the class to be refactored.

Bavota et al. [4] introduced an approach that splits the class to be refactored into two classes with higher cohesion than the original class based on structural and semantic similarities between the methods of class. The class is represented as a weighted graph where the nodes of the graph represent the methods of the class and the weighted edges of the graph represent the degree of structural and semantic similarities between the methods of the class. The approach uses the Max Flow-Min Cut algorithm [25] to split the weighed graph representing the original class into two weighted subgraphs representing the two classes that can be extracted from the original class. Moreover, Bavota et al. [5] introduced another approach that can automatically decompose the class to be refactored into two or more classes. The class is represented as a weighted graph in similar manner of their previous approach. Instead of using the Max Flow-Min Cut algorithm, they used a two-step clustering algorithm that removes the edges of the graph that have light weights to split the graph into a set of subgraphs representing the classes that can be extracted from the original class. Similarly, the authors of [26] introduced an automated refactoring approach based on complex network theory. The approach automatically identifies three types of refactoring opportunities: Move Method, Move Field, and Extract Class, to remove the bad smells caused by the low cohesion and high coupling in a given system.

On a further note, the authors of [27] conducted a systemic literature review to investigate the impact of refactoring on the design quality attributes. The author discussed and summarized results from 76 studies focusing on the impact of different refactoring scenarios including the Extract Class refactoring on quality attributes, e.g., cohesion. The presented results concluded that the Extract Class refactoring potentially improves the cohesion. To this end, it becomes clear that although the rich literature is a good addition to our view of the Extract Class refactoring, it only considers internal factors to the class to be refactored. The main limitation observed here is the applicability factor as they are not sufficient in many cases to identify ideal Extract Class refactoring opportunities within a class. In this paper, however, we introduce a new and innovative approach that automatically suggests a set of classes that can be extracted from a given class considering factors external to the class.

3. Extract Class Refactoring Based on Clients

In this section, we present basic definitions that reflect a theoretical model for an object-oriented system, and we follow with the proposed approach for the Extract Class refactoring.

3.1. A Theoretical Model

The proposed approach can be theoretically modeled based on the following set of definitions.

Definition 1 (Classes). *$S = \{c_1, c_2, \ldots, c_n\}$ is the set of classes that composes an object-oriented system, where n is a finite number of classes within the set {S}.*

For example, the set of classes of the system S shown in Figure 3 is $S = \{A, B, C, D, E\}$.

Definition 2 (Methods of a class). *Let $c \in S$. Then $M(c) = \{m_1, m_2, \ldots, m_k\}$ is the set of methods implemented in the class c, where k is a finite number of methods within the set $\{M(c)\}$.*

The set of methods of the class A shown in Figure 3 is $M(A) = \{a1, a2, a3, a4, a5, a6\}$.

Definition 3 (Clients of a method). *Let $c \in S$ and $m \in M(c)$. Then, $Client(m) = \{c' \in S - \{c\} \mid \exists m' \in M(c')$ and m' invokes or may, because of polymorphism, invoke $\{m\}$ that is the set of clients of $m\}$.*

For Example, $Client(a1) = \{B, C, D\}$ is the set of clients of the methods $a1$ in the class A shown in Figure 3. The classes B, C, and D are clients of the method $a1$ because each of them has a method that invokes the method $a1$ (i.e., $b1$, $c1$, and $d1$ for B, C, and D, respectively).

Definition 4 (Degree of client-based similarity between two methods). *Let $c \in S$ and $m_i, m_j \in M(c)$. Then, the degree of client-based similarity between the method m_i and the method m_j is defined by*

$$Sim_{clients}(m_i, m_j) = \begin{cases} \frac{|Client(m_i) \cap Client(m_j)|}{|Client(m_i) \cup Client(m_j)|} & \text{iff } |Client(m_i) \cup Client(m_j)| > 0, \\ 0 & \text{Otherwise;} \end{cases}$$

The value of $Sim_{clients}(m_i, m_j)$ ranges from $[0 - 1]$ where 0 means the methods share no common clients while 1 refer to the methods that have the same set of clients.

Consider the methods $a1$ and $a2$ in the class A shown in Figure 3. The clients of $a1$ is $Client(a1) = \{B, C, D\}$ and the clients of $a2$ is $Client(a2) = \{B, C, E\}$. Using these two sets, we compute the degree of client-based similarity between the two methods as follows,

$$Sim_{clients}(a1, a2) = \frac{|Client(a1) \cap Client(a2)|}{|Client(a1) \cup Client(a2)|} = \frac{2}{4} = 0.5.$$

```
public class A {
    ...
     public void a1()
     { ... }

     public void a2()
     { ... }

     public void a3()
     { ... }

     public void a4()
     { ... }

     public void a5()
     { ... }

     public void a6()
     { ... }

}
```

```
public class B {
    A objA;
    ...
     public void b1()
     {
         objA.a1();
         objA.a2();
         objA.a3();
         ...
     }
     ...
}
```

```
public class C {
    A objA;
    ...
     public void c1()
     {
         objA.a1();
         objA.a2();
         objA.a4();
         ...
     }
     ...
}
```

```
public class D {
    A objA;
    ...
     public void d1()
     {
          objA.a1();
          objA.a5();
          ...
     }
     ...
}
```

```
public class E {
    A objA;
    ...
     public void e1()
     {
          objA.a2();
          objA.a6();
          ...
     }
     ...
}
```

Figure 3. A hypothetical system S consisting of 5 classes. The figure shows the methods of the class A that are used by the other classes (the clients of A). The unnecessary implementation details of the classes and methods are hidden.

Definition 5 (Degree of client-based similarity between a method and a set of methods). *Let $c \in S$, $m \in M(c)$, $SM \subset M(c)$, and $m \notin SM$. Then, the degree of client-based similarity between the method m and the set of methods SM is defined by*

$$SimMethodWithSet_{clients}(m, SM) = \begin{cases} \frac{\sum_{x \in SM} Sim_{clients}(m,x)}{|SM|} & iff\ |SM| > 0, \\ 0 & Otherwise; \end{cases}$$

For example, the degree of client-based similarity between the method $m5$ in the class A shown in Figure 3 and the two methods $\{a1, a2\}$ as follows,

$$\begin{aligned} SimMethodWithSet_{clients}(m5, \{a1, a2\}) &= \frac{Sim_{clients}(a5, a1) + Sim_{clients}(a5, a2)}{|\{a1, a2\}|} \\ &= \frac{0.333 + 0}{2} \\ &= 0.167 \end{aligned}$$

Definition 6 (Cohesion Based on Client Similarity). *Let $c \in S$. Then, the Cohesion Based on Client Similarity (CBCS) for the class c is the ratio of the summation of the client-based similarity between each pair of methods in the class c to the total number of pairs of methods in c. It is formally defined as follows, when k is the number of methods in the class c.*

$$CBCS(c) = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} Sim_{clients}(m_i, m_j)$$

In order to better understand the way that the $CBCS$ is computed for a class, we compute it for the class A shown in the illustrative example presented in Figure 3.

$$\begin{aligned} CBCS(A) &= \frac{2}{6(6-1)} \sum_{i=1}^{6-1} \sum_{j=i+1}^{6} Sim_{clients}(a_i, a_j) \\ &= \frac{2(0.5 + 0.167 + 0.167 + 0.167 + 0 + 0.167 + 0.167 + 0 + 0.167 + 0 + 0 + 0 + 0 + 0 + 0)}{6(6-1)} \\ &= 0.1 \end{aligned}$$

3.2. Our Approach of the Extract Class Refactoring

The main steps of our approach are shown in Figure 4. Given a class to be refactored, we first extract the methods of the class. Then, we determine the clients of each method using Definition 3. Once we have the clients of each method in the class, we compute the degree of client-based similarity between each pair of methods in the class using Definition 4 and store the results in a $k \times k$ matrix where k is the number of methods in the class. The entry $[i][j]$ of the $k \times k$ matrix holds the degree of client-based similarity between the method m_i and the method m_j.

Next, Algorithm 1 is applied which takes an input the set of methods in the class to be refactored, the $k \times k$ matrix that holds the degree of client-based similarity between each pair of methods, and a threshold value. The algorithm returns a clutter, i.e., a family, of subsets from the input set of methods as an output. The algorithm classifies the input set of methods into different subsets of methods based on the client-based similarities between the methods. The input threshold value is used to determine if a method to be classified is added to an existing subset of methods or added to a new subset. Therefore, if the threshold value is high and the client-based similarities between the methods of the class to be refactored are generally low, each method will probably be added into a different subset, which means

each extracted class will have only one method. To overcome this issue, the median of the non-zero client-based similarities of all pairs of methods in the class can be chosen as a threshold value.

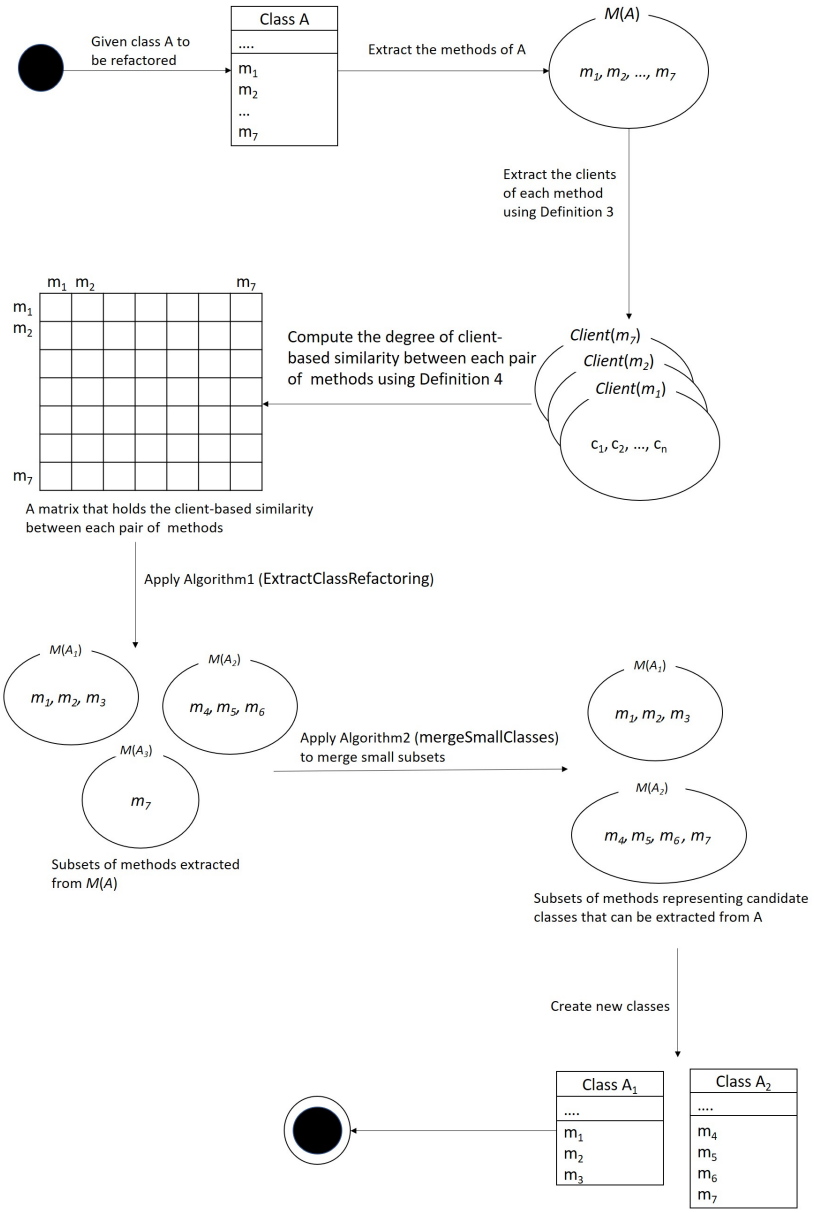

Figure 4. An outline of the main steps of our approach.

Finally, Algorithm 2 is applied to avoid the extraction of classes that have a small number of methods (e.g., one method). The algorithm takes as an input the clutter F resulting from Algorithm 1, the minimum number of methods that each extracted class can have, and the $k \times k$ matrix that holds the degree of client-based similarity between each pair of methods. The algorithm returns as an output a clutter of subsets such that each subset has a number of methods that is equal or more than the input minimum number of methods. Algorithm 2 merges any subset $X \in F$ that has less number of methods than the input minimum number of methods with the subset $Y \in F - X$ that has the highest average of the client-based similarities. Each subset in the output of Algorithm 2 represents a candidate class that can be extracted from the class to be refactored.

The attributes as well as private and protected methods of the class are not considered in this work. However, they can be automatically distributed among the extracted classes suggested by our approach based the use of the attributes as well as private and protected methods by the public methods in the

extracted classes. Each attribute can be added to the extracted class that has the highest number of methods that use the attribute; similarly, private and protected methods are distributed in the same way. Our approach automatically suggests the set of classes that can be extracted from a given class. A software engineer may evaluate the suggested classes and approve them or make some changes to them by moving the methods and attributes between suggested classes.

Algorithm 1: ExtractClassRefactoring($M(c)$, *Threshold*, $k \times k$ matrix)

Input: (1) $M(c)$: the set of methods of the class c to be refactored. (2) *Threshold*: a threshold value. (3) $k \times k$ matrix: holds $Sim_{clients}$ between each pair of methods in the class where k is the number of methods in the class

Output: a clutter F of subsets of $M(c)$ such that each subset represents a class that can be extracted from c

```
begin
    F = { };
    while |M(c)| > 1 do
        find the pair of methods m_i, m_j ∈ M(c) with highest Sim_clients(m_i, m_j) such that i ≠ j ;
        if Sim_clients(m_i, m_j) < Threshold then
            break;
        else
            add m_i, m_j to a new Subset;
            remove m_i, m_j from M(c);
            while |M(c)| > 0 do
                find the method m_k ∈ M(c) with highest SimMethodWithSet_clients(m_k, Subset)
                if SimMethodWithSet_clients(m_k, Subset) > Threshold then
                    add m_k to Subset;
                    remove m_k from M(c);
                else
                    break;
                end
            end
            add Subset to F;
        end
    end
    while |M(c)| > 0 do
        add each remaining method m_r in M(c) to a new Subset;
        add Subset to F;
        remove m_r from M(c);
    end
    return F
end
```

3.3. An Application Example

In order to consider an example to show how the proposed approach can be applied, let class X be the class to be refactored. Let $M(X) = \{m_1, m_2, m_3, m_4, m_5, m_6, m_7\}$. Let the clients of each method in X be the following,

$Client(m_1) = \{A, B\}$,
$Client(m_2) = \{A, B, C\}$,
$Client(m_3) = \{B, C\}$,
$Client(m_4) = \{D, E, F\}$,

$Client(m_5) = \{F, G\}$,
$Client(m_6) = \{F, G, H\}$, and
$Client(m_7) = \{G, H\}$.

Algorithm 2: mergeSmallClasses(F, $minNumMethods$, $k \times k$ matrix)

Input: 1) F: the output of Algorithm 1 which may include small subsets of methods. 2) $minNumMethods$: the minimum number of methods that each extracted class can have 3) $k \times k$ matrix: holds $Sim_{clients}$ between each pair of methods in the original class where k is the number of methods in the class

Output: F after merging small subsets

```
begin
    for X ∈ F do
        if |X| < minNumMethods then
            find Y ∈ F − X that has the highest average of client-based similarities after the
              merger of X and Y;
            add the elements of X to Y;
            remove X from F;
        end
    end
    return F
end
```

Given the above sets, we can compute the client-based similarity between each pair of methods of class X using Definition 4. The matrix shown in Table 1 holds the client-based similarity between each pair of methods of class X. The entry $[i][j]$ of the matrix holds the value of $Sim_{clients}(m_i, m_j)$.

Algorithm 1 is applied next to perform the Extract Class refactoring on the class X. The algorithm takes three inputs: $M(X)$, a *Threshold* value, and the matrix shown in Table 1. We set the value of *Threshold* to 0.5, which is the median of the values in the matrix in Table 1 excluding the zero values and the main diagonal of the matrix. The output of Algorithm 1 is the following clutter of subsets,

$$F = \{\{m_1, m_2, m_3\}, \{m_4\}, \{m_5, m_6, m_7\}\}.$$

Each subset represents a class that can be extracted from X.

Table 1. The client-based similarity between each pair of methods in the class X.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7
m_1	1	0.67	0.33	0	0	0	0
m_2	0.67	1	0.67	0	0	0	0
m_3	0.33	0.67	1	0	0	0	0
m_4	0	0	0	1	0.25	0.20	0
m_5	0	0	0	0.25	1	0.67	0.33
m_6	0	0	0	0.20	0.67	1	0.67
m_7	0	0	0	0	0.33	0.67	1

Algorithm 2 can be applied next to avoid extracting classes with a small number of methods. The algorithm takes three inputs: the clutter F (i.e., the output of Algorithm 1); $minNumMethods$, which is a chosen value for the minimum number of methods that each extracted class can have; and the matrix shown in Table 1. In this example, $minNumMethods = 2$. The following is the output of Algorithm 2,

$$F = \{\{m_1, m_2, m_3\}, \{m_4, m_5, m_6, m_7\}\}.$$

The output of Algorithm 2 shows that the subset $\{m_4\}$ is merged with the subset $\{m_5, m_6, m_7\}$. Thus, two classes are suggested to be extracted from the original class X in our example. The first extracted class has the following set of methods $\{m_1, m_2, m_3\}$ and the second extracted class has the following set of methods $\{m_4, m_5, m_6, m_7\}$.

4. Empirical Evaluations Utilizing Open Source Systems

To evaluate the proposed approach, we considered two real systems in this study: `Tomcat` [28] version 6.0.0 and `Ant` [29] version 1.5.2. Both of the two systems are open source, object-oriented, and written in Java. `Tomcat` is a system used to implement *Java Servlet, JavaServer Pages, and JavaWebSocket technologies* and `Ant` is a Java library and command-line tool used mainly to build Java applications. Ten large classes are selected from each system based on the following two criteria. (1) The class has at least 8 public methods and (2) the cardinality of the union of the clients of the methods is at least 2. Tables 2 and 3 report the name of the `Blob`, the name of the package to which the class belongs, the number of public methods in the class (#Methods), and the cardinality of the union of the clients of each method in the class (#Clients).

Table 2. The classes selected from `Tomcat` 6.0.0.

Class Name	Package Name	#Methods	#Clients
`GenericServlet`	javax.servlet	12	21
`ServletResponseWrapper`	javax.servlet	17	36
`ServletRequestWrapper`	javax.servlet	31	50
`HttpServletRequestWrapper`	javax.servlet.http	25	35
`NoBodyResponse`	javax.servlet.http	32	38
`Request`	org.apache.catalina.connector	123	91
`ObjectReader`	org.apache.catalina.tribes.io	14	3
`XByteBuffer`	org.apache.catalina.tribes.io	40	24
`ChannelData`	org.apache.catalina.tribes.io	24	18
`MemberImpl`	org.apache.catalina.tribes.membership	39	22

Table 3. The classes selected from `Ant` 1.5.2.

Class Name	Package Name	#Methods	#Clients
`Project`	org.apache.tools.ant	77	157
`Task`	org.apache.tools.ant	16	185
`Zip`	org.apache.tools.ant.taskdefs	19	4
`Ant`	org.apache.tools.ant.taskdefs	14	4
`Execute`	org.apache.tools.ant.taskdefs	15	31
`Extension`	org.apache.tools.ant.taskdefs.optional.extension	15	5
`VAJBuildInfo`	org.apache.tools.ant.taskdefs.optional.ide	19	4
`CommandlineJava`	org.apache.tools.ant.types	25	12
`Commandline`	org.apache.tools.ant.types	22	62
`Path`	org.apache.tools.ant.types	20	67

Our goal in the empirical evaluation is to investigate if the proposed refactoring approach can suggest to extract classes with cohesive responsibilities from a given class that has many non-cohesive responsibilities. For this purpose, we formulate the following research question.

RQ: Do the extracted classes suggested by the proposed approach have higher cohesion than the original classes?

To answer the above research question, we compare the cohesion of the original classes with the cohesion of the extracted classes. We measure class cohesion using *CBCS* given in Definition 6. *CBCS* is a variation of *CCC* which was proposed and validated in a previous works [7,8]. The original metric *CCC* measures the cohesion of a class based on the ratio of the summation of client-based class cohesion per each client of the class to the total number of clients of the class, where the client-based

class cohesion per a client is defined as the number of methods used by the client divided by the total number of methods in the class. The reason of why we consider *CBCS* instead of *CCC* is that *CBCS* computes the similarities between each pair of methods in the class in order to measure the class cohesion while *CCC* does not, and cohesion metrics that compute the similarities between the methods are better in identifying the non-cohesive (dissimilar) methods that need to be separated into different classes [30].

We developed two tools to automate the application of our proposed approach. The first tool is implemented in `Java` and it is based on `Eclipse JDT` [31]. The tool automatically extracts the public methods of the classes of the considered system and the clients of each method, and calculates the client-based similarity between each pair of methods using Definition 4. The input of the tool is the source code of the system and output are the set of public methods of each class and a $k \times k$ matrix that has the client-based similarity between each pair of public methods in the class, where k is the number of public methods in the class. The second tool we developed is a Python tool that takes the output of the Java tool and automatically applies Algorithm 1 and Algorithm 2. The output of the Python tool is the set of classes that are suggested to be extracted from the class in question. In addition, the python tool calculates the value of *CBCS* for the original class and for each class suggested to be extracted from the original class.

The research question of our empirical study investigates whether the cohesion of the extracted classes identified by our approach is higher than the cohesion of the original classes. We believe that the extracted classes will have higher cohesion values than the original classes. Thus, we set up the following null and alternative hypotheses for our research question.

- $\mathbf{H}_0$: The difference between the CBCS values of the original classes and the CBCS values of the extracted classes suggested by our approach is 0.
- $\mathbf{H}_1$: The original classes have less CBCS values than CBCS values of the extracted classes suggested by our approach.

4.1. Results And Discussions

Using our tools, we applied the proposed approach on the two considered systems. Tables 4 and 5 report the results for the ten classes selected from `Tomcat` system and the ten classes selected from `Ant` system, respectively. We set the refactoring threshold (i.e., *Threshold* used in Algorithm 1) for each considered class to the median of the values of the client-based similarities between all the pairs of the methods in the class excluding the zero values and the main diagonal entries. We set the minimum number of methods each extracted class can have to 2, i.e., *minNumMethods* used in Algorithm 2. This is due to unreasonability to extract classes that have only one method. The results in Tables 4 and 5 lead to the following observations.

- The cohesion (i.e., *CBCS* values) of the extracted classes suggested by our approach are higher than the original classes for all the selected classes.
- In most of the cases, the *CBCS* values of the extracted classes increased to more than two times that of the *CBCS* value in original class.
- In some cases, the extracted classes have relatively low *CBCS* values (e.g., classes extracted from *NoBodyResponse* class in `Tomcat`). We investigated these cases and we found the main reason behind that is the values of the client-based similarities between most of the pairs of methods in the original class are 0.
- The number of extracted classes varies from cases to cases. Two factors can influence the number of the extracted classes: the refactoring threshold and the number of methods in the original class. The higher the value of the refactoring threshold is, the more likely the number of extracted classes will increase (e.g., the cases of the classes *Extension* and *VAJBuildInfo* in `Ant` system). Similarly, the higher the number of methods in the original class, the more classes will probably be extracted (e.g., the case of the class Request in `Tomcat` system).

Table 4. Extracted classes suggested by our approach for the considered classes of the `Tomcat` system.

Class Name	Original Class		Threshold	Extracted Classes		
	#Methods	CBCS		#Methods	CBCS	Avg. CBCS
`GenericServlet`	12	0.113	0.193	8 4	0.204 0.208	0.206
`ServletResponseWrapper`	17	0.042	0.125	7 7 3	0.095 0.094 0.111	0.100
`ServletRequestWrapper`	31	0.114	0.2	25 4 2	0.157 0.183 0.273	0.204
`HttpServletRequestWrapper`	25	0.143	0.222	18 7	0.154 0.2	0.177
`NoBodyResponse`	32	0.031	0.154	7 18 3 4	0.053 0.053 0.067 0.056	0.057
`Request`	123	0.04	0.157	41 47 18 3 6 6 2	0.109 0.11 0.113 0.167 0.116 0.14 0.167	0.132
`ObjectReader`	14	0.33	0.5	4 3 3 4	0.5 1 1 0.75	0.813
`XByteBuffer`	40	0.047	0.25	6 7 7 5 8 4 3	0.24 0.214 0.214 0.267 0.262 0.278 0.278	0.250
`ChannelData`	24	0.063	0.286	10 5 4 5	0.168 0.2 0.167 0.204	0.185
`MemberImpl`	39	0.045	0.333	5 12 15 5 2	0.15 0.12 0.114 0.125 0.333	0.168

We further conducted the paired samples *t*-test using the *R* software [32] in order to test our null $\mathbf{H}_0$ and alternative $\mathbf{H}_1$ hypotheses (see Table 6). The paired samples *t*-test is a statistical procedure used to compare the means between two related sets of samples of the same size. In our case, the two sets are the *CBCS* values of the original classes and the averages of *CBCS* values of the extracted classes suggested by our approach for each original class (the reader is directed to data presented in Tables 4 and 5). In other words, a pair of samples has the *CBCS* of an original class and the average of the *CBCS* values of the classes extracted from that original class. We considered the averages of *CBCS* values of extracted classes instead of *CBCS* value of each extracted class because the total number of

the extracted classes are greater the number of the original classes. The results shown in Table 6 are statistically significant because the *p*-values are less than 0.05. Based on these results, we can reject the null hypothesis $\mathbf{H}_0$ and accept the alternative $\mathbf{H}_1$.

Table 5. Extracted classes suggested by our approach for the considered classes of the `Ant` system.

Class Name	Original Class		Threshold	Extracted Classes		
	#Methods	CBCS		#Methods	CBCS	Avg. CBCS
`Project`	77	0.046	0.167	23 8 33 11 2	0.11 0.118 0.109 0.111 0.169	0.123
`Task`	16	0.093	0.143	9 2 2 3	0.183 0.5 0.37 0.207	0.315
`Zip`	19	0.076	0.333	8 11	0.107 0.109	0.108
`Ant`	14	0.114	0.5	7 4 3	0.143 0.167 0.167	0.159
`Execute`	15	0.127	0.158	7 6 2	0.225 0.303 0.5	0.343
`Extension`	15	0.266	1	5 2 4 2 2	0.425 1 0.5 1 1	0.785
`VAJBuildInfo`	19	0.227	1	3 5 3 4 2 2	0.333 0.45 0.667 0.5 1 1	0.658
`CommandlineJava`	25	0.18	0.25	14 11	0.255 0.302	0.279
`Commandline`	22	0.053	0.114	9 9 4	0.118 0.095 0.145	0.119
`Path`	20	0.069	0.105	7 10 3	0.147 0.154 0.167	0.156

Table 6. The paired samples *t*-test for the `Tomcat` and `Ant` classes.

	t	*df*	*p*-Value	95% Confidence Interval		Mean of the Differences
				Lower	Upper	
`Tomcat`	−3.1439	9	0.005927	−∞	−0.05520043	−0.1324
`Ant`	−3.3383	9	0.004342	−∞	−0.08088784	−0.1794

4.2. Threats to Validity

Several issues may affect the validity of the results in our empirical study. The first issue is that we evaluated our approach by comparing the cohesion of extracted classes to the cohesion of the original classes and the metric used to measure cohesion (i.e., *CBCS*) is based on information that is used in our refactoring approach. *CBCS* measures the cohesion based on the client-based similarity between each pair of methods in the class and our approach uses the client-based similarity to identify pairs of methods that should belong to the same extracted class. Thus, it is most likely the extracted classes will have higher *CBCS* values than the original class. However, our main purpose was to show an empirical evidence based on real classes that extracted classes resulting from our algorithms have higher *CBCS* values than the original classes.

Another issue is that we did not evaluate the extracted classes in terms of coupling. Coupling refers to the degree of relatedness between classes. Cohesion and coupling are two quality attributes that can have an inverse relationship, meaning the improvement of one quality attribute can lead to the deterioration of the other quality attribute [13,14]. Thus, we need to keep balance between cohesion and coupling when performing the extract class refactoring [5,30]. However, the focus of this paper is to use the client-based cohesion to support the extract class refactoring, as they have not been exploited in previous approaches. Employing metrics of coupling and other aspects of cohesion, such as structural and semantic cohesion, together with the metrics of client-based similarity to support the Extract Class refactoring, is left for future research.

The last issue which may affect the validity of our results is that the extracted classes suggested by the proposed approach were not evaluated by software engineers. It is important to evaluate the solutions suggested by newly proposed refactoring approaches by a number of software engineers [5]. If the majority of the evaluators (i.e., Software Engineers) are not satisfied with solutions suggested by a refactoring approach, the applicability of the approach is questionable. However, solutions that improve important attributes such as cohesion are expected to satisfy software engineers. Thus, we evaluated our approach based on cohesion using the metric *CBCS*. Several well-known papers in the literature (see, e.g., in [15–18]) have used quality metrics to evaluate newly proposed refactoring techniques.

5. Conclusions and Future Work

Software systems that have a long lifetime (e.g., enterprise systems) usually undergo evolutionary changes in order to remain useful due to various reasons including changes to business rules, user requirements, hardware, and platforms. Unfortunately, most of the programmers who are responsible for making these changes in a system modify the source code of that system rapidly without considering the resultant effects on the design of the system. As a result, the design quality of the system deteriorates, and the system becomes very difficult to understand and change. This paper has introduced a novel approach that performs the Extract Class refactoring. The proposed approach used the clients of the class to identify classes that can be extracted from that class. An empirical study on classes selected from two open source systems was conducted to evaluate the proposed approach. The results of the empirical study highlight the potential usefulness of the proposed approach. The future work should further conduct much larger empirical studies that quantitatively analyze the relationships among multiple approaches for the Extract Class refactoring, considering the approach proposed in this paper, and real cases of Extract Class refactoring for the purpose of identifying the factors that can be used to better classify and separate different responsibilities of a class.

Author Contributions: Conceptualization, M.A. and S.A.; methodology, M.A.; software, M.A. and S.A.; validation, M.A. and S.A.; formal analysis, M.A. and S.A.; investigation, M.A.; resources, M.A.; data curation, M.A. and S.A.; writing—original draft preparation, M.A.; writing—review and editing, M.A and S.A.; visualization, S.A.; supervision, M.A.; project administration, M.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Professional: Boston, MA, USA, 1999.
2. Martin, R.C.; Martin, M. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*; Prentice Hall PTR: Upper Saddle River, NJ, USA, 2006.
3. Fokaefs, M.; Tsantalis, N.; Chatzigeorgiou, A.; Sander, J. Decomposing object-oriented class modules using an agglomerative clustering technique. In Proceedings of the 2009 IEEE International Conference on Software Maintenance, Edmonton, AB, Canada, 20–26 September 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 93–101.
4. Bavota, G.; De Lucia, A.; Oliveto, R. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *J. Syst. Softw.* **2011**, *84*, 397–414. [CrossRef]
5. Bavota, G.; De Lucia, A.; Marcus, A.; Oliveto, R. Automating extract class refactoring: an improved method and its evaluation. *Empir. Softw. Eng.* **2014**, *19*, 1617–1664. [CrossRef]
6. Mäkelä, S.; Leppänen, V. Client-based cohesion metrics for Java programs. *Sci. Comput. Program.* **2009**, *74*, 355–378. [CrossRef]
7. Alzahrani, M.; Melton, A. Defining and validating a client-based cohesion metric for object-oriented classes. In Proceedings of the 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Turin, Italy, 4–8 July 2017; IEEE: Piscataway, NJ, USA, 2017; Volume 1, pp. 91–96.
8. Alzahrani, M.; Alqithami, S.; Melton, A. Using Client-Based Class Cohesion Metrics to Predict Class Maintainability. In Proceedings of the 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA, 15–19 July 2019; IEEE: Piscataway, NJ, USA, 2019; Volume 1, pp. 72–80.
9. Alzahrani, M. Using Clients to Support Extract Class Refactoring. In Proceedings of the 18th International Conference on Software Engineering Research and Practice, Las Vegas, NV, USA, 27–30 July 2020; in press.
10. Fokaefs, M.; Tsantalis, N.; Stroulia, E.; Chatzigeorgiou, A. JDeodorant: Identification and application of extract class refactorings. In Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 21–28 May 2011; IEEE: Piscataway, NJ, USA, 2011; pp. 1037–1039.
11. Simon, F.; Steinbruckner, F.; Lewerentz, C. Metrics based refactoring. In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, 14–16 March 2001; IEEE: Piscataway, NJ, USA, 2001; pp. 30–38.
12. Marinescu, R. Detection strategies: Metrics-based rules for detecting design flaws. In Proceedings of the 20th IEEE International Conference on Software Maintenance, Chicago, IL, USA, 11–14 September 2004; IEEE: Piscataway, NJ, USA, 2004; pp. 350–359.
13. Stewart, K.J.; Darcy, D.P.; Daniel, S.L. Opportunities and challenges applying functional data analysis to the study of open source software evolution. *Stat. Sci.* **2006**, *21*, 167–178. [CrossRef]
14. Du Bois, B.; Demeyer, S.; Verelst, J. Refactoring-improving coupling and cohesion of existing code. In Proceedings of the 11th working conference on reverse engineering, Delft, The Netherlands, 8–12 November 2004; IEEE: Piscataway, NJ, USA, 2004; pp. 144–151.
15. Praditwong, K.; Harman, M.; Yao, X. Software module clustering as a multi-objective search problem. *IEEE Trans. Softw. Eng.* **2010**, *37*, 264–282. [CrossRef]
16. Seng, O.; Stammel, J.; Burkhart, D. Search-based determination of refactorings for improving the class structure of object-oriented systems. In Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, Seattle, DC, USA, 8–12 July 2006; pp. 1909–1916.
17. Abdeen, H.; Ducasse, S.; Sahraoui, H.; Alloui, I. Automatic package coupling and cycle minimization. In Proceedings of the 2009 16th Working Conference on Reverse Engineering, Lille, France, 13–16 October 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 103–112.
18. O'Keeffe, M.; Cinnéide, M.O. Search-based software maintenance. In Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'06), Bari, Italy, 22–24 March 2006; IEEE: Piscataway, NJ, USA, 2006; p. 10.

19. Bavota, G.; De Lucia, A.; Marcus, A.; Oliveto, R. A two-step technique for extract class refactoring. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, 20–24 September 2010; pp. 151–154.
20. Pappalardo, G.; Tramontana, E. Suggesting extract class refactoring opportunities by measuring strength of method interactions. In Proceedings of the 2013 20th Asia-Pacific Software Engineering Conference (APSEC), Bangkok, Thailand, 2–5 December 2013; IEEE: Piscataway, NJ, USA, 2013; Volume 2, pp. 105–110.
21. Bavota, G.; De Lucia, A.; Marcus, A.; Oliveto, R.; Palomba, F. Supporting extract class refactoring in eclipse: The aries project. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 1419–1422.
22. Bavota, G.; Oliveto, R.; De Lucia, A.; Antoniol, G.; Gueheneuc, Y.G. Playing with refactoring: Identifying extract class opportunities through game theory. In Proceedings of the 2010 IEEE International Conference on Software Maintenance, Timi oara, Romania, 12–18 September 2010; IEEE: Piscataway, NJ, USA, 2010; pp. 1–5.
23. Chidamber, S.R.; Kemerer, C.F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* **1994**, *20*, 476–493. [CrossRef]
24. Bafandeh Mayvan, B.; Rasoolzadegan, A.; Javan Jafari, A. Bad smell detection using quality metrics and refactoring opportunities. *J. Softw. Evol. Process.* **2020**, e2255.10.1002/smr.2255. [CrossRef]
25. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introd. Algorithms*; MIT Press: Cambridge, MA, USA, 2009.
26. Wang, Y.; Yu, H.; Zhu, Z.; Zhang, W.; Zhao, Y. Automatic software refactoring via weighted clustering in method-level networks. *IEEE Trans. Softw. Eng.* **2017**, *44*, 202–236. [CrossRef]
27. Al Dallal, J.; Abdin, A. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Trans. Softw. Eng.* **2017**, *44*, 44–69. [CrossRef]
28. Apache Tomcat. Available online: https://tomcat.apache.org/ (accessed on 22 March 2020).
29. Apache Ant. Available online: https://ant.apache.org/ (accessed on 22 March 2020).
30. Al Dallal, J.; Briand, L.C. A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2012**, *21*, 1–34. [CrossRef]
31. Eclipse Java Development Tools (JDT). Available online: https://www.eclipse.org/jdt/ (accessed on 22 March 2020).
32. R Foundation. The R Project for Statistical Computing. 2018. Available online: http://www.r-project.org (accessed on 22 March 2020).