2025 IEEE 22nd International Conference on Software Architecture (ICSA) | 979-8-3315-2090-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICSA65012.2025.00037

Deicide: Decomposing Complex Classes into Responsibility Modules

Jason Lefever
Drexel University
Philadelphia, USA
jtl86@drexel.edu

Yuanfang Cai
Drexel University
Philadelphia, USA
yfcai@cs.drexel.edu

Rick Kazman
University of Hawaii
Honolulu, USA
kazman@hawaii.edu

Ernst Pisch
Drexel University
Philadelphia, USA
epp26@drexel.edu

***Abstract*—Refactoring a large and complex class can be challenging, not only because the class aggregates many different responsibilities but also because of its potentially extensive impact on external classes. Although many extract class methods have been proposed, few support a holistic decomposition that can uncover multiple distinct responsibilities within a complex class, along with their system-wide impacts. Recent research highlights the need for such a holistic view before an organization can commit to redesigning and refactoring. To identify distinct responsibilities while minimizing internal and external impacts, we created Deicide, a new decomposition algorithm that uses an internal call graph, external usage patterns, and semantic similarity of identifiers to calculate a hierarchical set of cohesive clusters, each forming a *responsibility module.* We evaluated Deicide against three state-of-the-art extract class recommenders using 123 large, change-prone classes from 9 open-source projects. Our results show that the entities within the clusters identified by Deicide are more likely to be changed together and changed by the same group of developers, indicating de facto cohesive responsibilities. The implication is that refactoring based on Deicide's recommendations would have minimal impact on the system, and these newly extracted classes would be able to evolve independently.**

***Index Terms*—class decomposition, refactoring, extract class, code smells, clustering algorithms**

I. Introduction

Large and complex classes are among the most challenging code smells [1]—indicators of potential problems in the design or structure of code that may not immediately cause errors but can lead to maintainability, evolvability, and other quality issues over time [2]–[12]. Recent industrial studies [13], [14] have shown that decomposing these large, complex classes is challenging, despite the numerous extract class or extract method recommenders that have been proposed [10], [15]–[26]. These tools often output a sequence of refactoring suggestions, with each step extracting one or two new classes. However, practitioners have noticed that decomposing a class is "*not just a matter of applying several extract method, extract class, or move method refactorings.*" Instead of blindly following the refactoring steps recommended by a refactoring tool, there should be a planning stage, where the designers first inspect which and how many distinct responsibilities are aggregated in a complex class, how they are related, and most importantly, how they impact external classes, and make their refactoring decisions accordingly [13], [14].

While current extract class/method tools do generate internal decompositions (clusterings) to inform their recommendations, these decompositions are not provided to the user for redesign planning. And these internal decompositions have not been evaluated to assess if the decomposition can identify multiple distinct responsibilities, that is, entities that can evolve and change independently.

In this paper, we address these problems by first comparing three class decomposition algorithms from state-of-the-art extract class recommenders and evaluating their effectiveness at producing holistic decompositions. As we will show, these algorithms produce dramatically different decompositions due to their different choices of clustering criteria, using internal structural dependencies, client dependencies, or identifier semantics. Most interestingly, these algorithms often produce imbalanced (or "lopsided") decompositions because of their bottom-up approach. As a result, they tend to extract one or two new classes, rather than revealing all distinct responsibilities. A complex class with thousands of lines of code (LOC), however, often contains dozens, if not hundreds, of distinct responsibilities, and these should be presented to the designer for inspection before making refactoring decisions. This motivates our approach, *Deicide*, which uses a comprehensive set of criteria and identifies a set of distinct *responsibility modules* suitable for the planning stage of a class refactoring.

Unlike existing approaches, Deicide evaluates the three key criteria: internal structural dependencies, client dependencies, and identifier semantics to identify cohesive responsibility modules *comprehensively*, resulting in hierarchical decomposition. Using such a decomposition, a designer can flexibly choose to decompose a complex class into multiple new classes at different levels of granularity, ensuring maximum internal cohesiveness and minimal external impact. Unlike existing algorithms, Deicide employs a top-down acyclic graph partitioning approach that considers indirect transitive relationships so that entities for the same responsibility can be clustered together.

To evaluate whether Deicide's responsibility modules are truly cohesive and thus supporting independent evolution and maintenance, we compare its recommended clusterings against each project's revision history to assess how well the entities that were clustered together were actually changed together or changed by the same developers. Our rationale builds upon

2835-7043/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSA65012.2025.00037

prior research, which demonstrates that frequently co-changing entities are strong indicators of cohesion and evolutionary coupling, with the same group of authors often revising these entities to fulfill related responsibilities [27]–[39]. Consequently, we explore the following research questions:

RQ1: *To what extent are the entities clustered together by a decomposition algorithm actually changed together?* If a majority of the entities that were clustered together actually changed together, as recorded in their commit history, it means that these entities belong to a de facto cohesive responsibility group.

RQ2: *To what extent are the entities clustered together actually changed by the same developers?* Since the same responsibilities are more likely to be maintained by the same developers, if the recommended clusters include entities frequently committed by the same authors, it is more likely that these entities belong to the same responsibility modules.

Here we benchmark Deicide against three hierarchical class decomposition algorithms used by state-of-the-art extract class recommenders: Fokaefs et al.'s JDeodorant [10], [18], [19], Akash's algorithm [21], and Alzahrani's algorithm [22]–[24]. Our results demonstrate that Deicide's decompositions align significantly better with the actual commit history of the projects, achieving nearly double the score of the next best algorithm—33.32% versus 17.70% on average. This indicates that if a class was refactored using Deicide's decomposition, the resulting newly extracted classes would more likely evolve independently with minimal impact on the rest of the architecture, which is highly desirable.

II. Class Decomposition

In this section, we use a running example to illustrate how and why existing well-known and/or state-of-the-art class decomposition algorithms generate drastically different clusters and highlight the remaining challenges. For this study, we select the following three tools/algorithms:

1) **JD**: JDeodortant by Fokaefs et al. [19] uses a hierarchical agglomerative clustering (HAC) algorithm to iteratively merge entities based on the Jaccard index of their "entity sets." Each method's entity set includes all methods and fields it uses, and each field's entity set includes all methods that use it. This well-known tool is available as an Eclipse plug-in.

2) **AK**: The approach of Akash et al. [21] also uses a HAC algorithm to iteratively merge entities but measures similarity by averaging three different metrics together, including two structural metrics that, like JD, are based on the direct incoming and outgoing dependencies between methods, and one metric that uses Latent Dirichlet Allocation (LDA) to measure the semantic similarity between method pairs. This approach improves the well-known refactoring recommender work of Bavota et al. [16], [17], [20], but is the first in this line of work to produce a hierarchical decomposition.

3) **AL**: Alzahrani's approach [24] uses a greedy HAC-like algorithm that prioritizes merging with the most recently created cluster from the previous iteration. Similarity between two methods is defined as the Jaccard index of their "client sets", where the client set of a method is the set of all classes that call that method (i.e., clients).

These algorithms were selected because they are either well-known or represent the state-of-the-art in hierarchical class decomposition. JDeodorant is the most widely cited and used tool, while Akash's and Alzahrani's algorithms represent the latest advancements in hierarchical clustering-based techniques. Although each method uses a distinct combination of structural, semantic, or client information, they all share a similar bottom-up clustering approach.

In addition to hierarchical decomposition, there are many extract class recommenders that employ partitional decompositon [15]–[17], [20], [25], [26]. While larger modules derived from hierarchical decomposition can be further divided into smaller responsibility modules, the modules from partitional decomposition are mutually exclusive. In this paper, we focus on hierarchical decompositions because they offer architects better flexibility during refactoring, allowing them to examine class responsibilities at varying levels of granularity. Moreover, none of these partitional algorithms consider both internal and external relations of class entities. An in-depth comparison of these two categories is future work.

A. How Different Methods Decompose

To illustrate the behavior of these algorithms, we present the resulting decompositions obtained by executing each one on the same class, `TsFileSequenceReaderForV2.java` of Apache IoTDB.[1] Figure 1 depicts how the 21 methods of this class are clustered by different algorithms. To highlight their differences, we selected six methods, from m_7 to m_{12} (Figure 1e), which are supposed to be clustered together as a distinct module, according to their relations and according to how they have changed in the past.

Figure 1 shows the distinct structures produced by JDeodorant (JD), Akash's algorithm (AK), Alzahrani's algorithm (AL), and our Deicide algorithm (DC), which will be detailed in the next section. The bottom layer of each subfigure represents all 21 methods of the class, labeled m_1 through m_{21}. The remaining layers contain clusters that aggregate the elements from the layers below. For convenience, we assign IDs to these clusters, following a left-to-right, top-to-bottom order. For instance, Cluster 26 in Figure 1a includes methods m_{11}, m_8, and m_{10}, while Cluster 22 encompasses all the methods in Cluster 26 along with method m_{12}.

Our first observation is that the decompositions produced by these algorithms are drastically different. Notice the placement of the six highlighted methods: In JD, they are grouped into the same top-level cluster along with eight other methods. In AK, they are split between two top-level clusters, displaying some grouping within each half (e.g., Cluster 17 and Cluster 6). In AL, these six methods are distributed more evenly across the hierarchy.

[1]This class is identifed as `iotdb-14711` in our replication package. Source code is available at https://github.com/apache/iotdb/blob/v1.1.0/tsfile/src/main/java/org/apache/iotdb/tsfile/v2/read/TsFileSequenceReaderForV2.java.

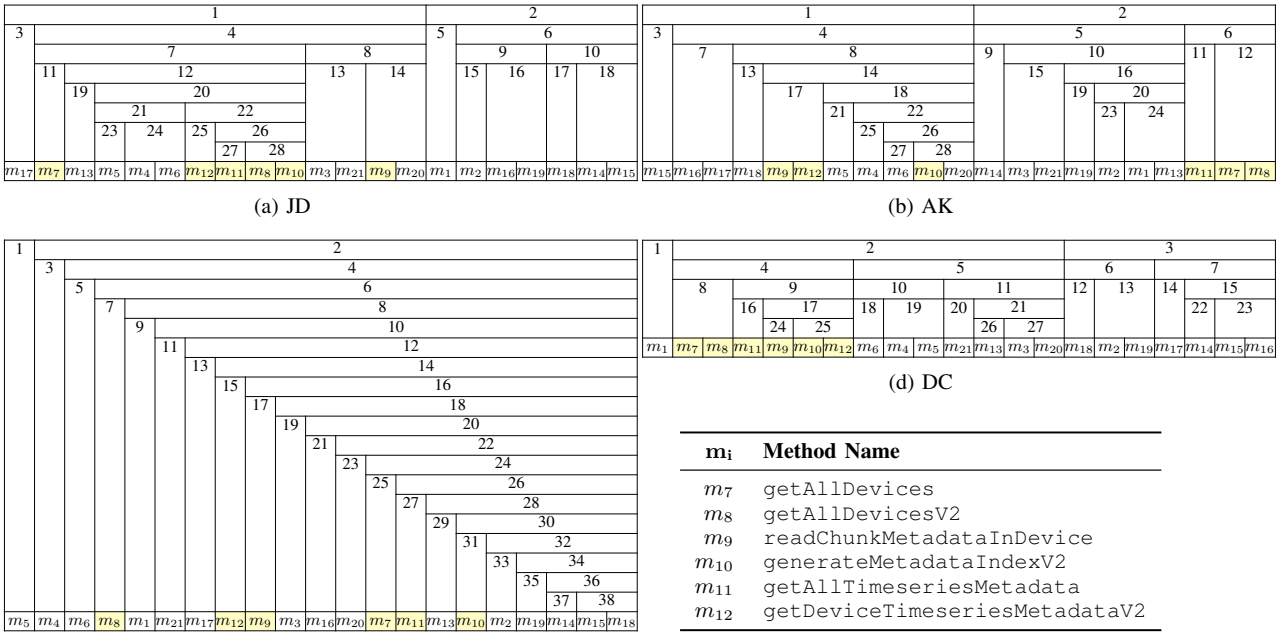

Fig. 1: Example decompositions produced by JD, AK, AL, and DC

These differences arise from the unique relationships each algorithm considers. JD relies solely on internal structure, AK incorporates both internal structure and semantics, while AL focuses exclusively on external dependencies (i.e., client relationships). For example, AK places m_7 and m_8 close together due to their similar names, while JD and AL place them much further apart. In contrast, AL places m_7 and m_9 near each other due to their shared client (not shown), but neither JD nor AK positions them similarly. Note that none of these methods aggregate these six methods together as one distinct module as they should be.

Finally, JD, AK, and AL exhibit a shared imbalance or "lopsidedness." These methods tend to identify a few central entities and assign all other entities to singleton clusters, sorted by their degree of association with these central entities. This phenomenon is particularly evident in AK (centered on Clusters 28 and 24) and AL (centered on Cluster 38). Consequently, these decompositions are not *holistic* as they focus on only one or two particular responsibilities of the class instead of capturing its full range of responsibilities.

B. Remaining Challenges

After carefully analyzing these algorithms, we have identified two primary challenges that remain to be addressed. First, these algorithms are limited by the types of connections they consider between entities. JD and AL, for instance, do not capture semantic relationships among methods, while JD and AK do not account for client dependencies. Second, the bottom-up strategy employed by these algorithms limits their ability to capture the full range of responsibilities within a class. Unlike top-down clustering algorithms, bottom-up approaches merge clusters using only information local to the clusters being merged, which prevents them from capturing broader relations [40], [41]. For example, if method m_a calls m_b, which calls m_c, which in turn calls m_d, JD and AK would fail to identify any relationship between m_a and m_d until a sufficient number of intermediate clusters had already been merged. Meanwhile, AL would overlook this relationship entirely, as it only considers client dependencies. To produce a holistic class decomposition, we must take into account relationships comprehensively, e.g., including transitive relations.

Figure 1d illustrates the outcome of Deicide, which addresses these challenges, leveraging internal structure, semantics, and client dependencies comprehensively and presenting a top-down hierarchical decomposition. Deicide is the only algorithm that aggregates all six methods together, and only these six methods, in Cluster 4 as they should be. Furthermore, Deicide's decomposition is significantly more balanced, clearly identifying the distinct responsibility modules. Unlike other algorithms, Deicide generates clear splits between modules at every level. For example, at the top layer of Figure 1d, the class is divided into three larger responsibility modules: Clusters 1, 2, and 3, and Cluster 2 and 3 can be further divided. At the second-highest layer, the class is further decomposed into five responsibility modules: Clusters 1, 4, 5, 6, and 7, each representing distinct responsibilities. Based on this decomposition, the user can choose to refactor the class into three or more new classes depending on their preferred granularity. In the following section, we introduce the Deicide algorithm in detail.

III. Algorithm

The Deicide algorithm aims to produce holistic decompositions by integrating multiple dependency types—internal call graph relationships, external client usage patterns, and semantic similarities, to maximize internal cohesiveness and minimize external impact. Deicide also captures transitive structural dependencies, and ensures that the resulting decomposition is acyclic. To demonstrate the algorithm in action, we use a running example of decomposing one of our evaluation subjects, `UrlUtils.java` of Apache Dubbo[2]. In the following subsections, we detail the steps of graph construction, condensation, and partition.

A. Graph Construction

First, we construct the internal syntactic dependency graph of the target class. Let $G_i = (V_i, E_i)$ be this graph. The vertex set V_i contains the internal entities that are to be clustered. In a typical Java class file, this would be all fields, methods, and nested classes that sit immediately inside the body of the outermost class definition. The directed edge set E_i contains the internal structural dependencies between these entities such as method calls and field accesses. Each entity $v \in V_i$ has unit weight: $w(v) = 1$, and each edge $(u, v) \in E_i$ has unit cost: $cost_i(u, v) = 1$.

1) External Usage: Now we extend this graph to include client files. Let $V = V_i \cup V_e$ where the vertex set V_e contains the external client files. These are all files that structurally depend on the target class. Furthermore, let $E_d = E_i \cup E_e$ be the set of all structural dependency edges where E_e contains the external dependencies from client files to the internal entities. Now, our graph can be expressed as $G_d = (V, E_d)$. The cost of each edge $(u, v) \in E_e$ is also $cost_e(u, v) = 1$, but we set the weight of each client $v \in V_e$ to $w(v) = 0$.

By including client files in the graph, we introduce a penalty for separating internal entities that are frequently used together by the same clients. A client file contributes an edge for every internal entity it uses. For instance, if two methods are frequently called together, separating them will cut across more edges and therefore incur a higher cost than separating two methods that are rarely used together. The client files are weighted as zero to avoid counting towards the total size of their containing clusters.

Figure 2 illustrates this graph as an adjacency matrix. The rows and columns are labeled with a subset of V that includes nine methods $(m_1, m_2, \ldots, m_{17})$ and one client file (f_{14}). Each non-empty cell indicates an edge between the item on the row to the item on the column. Figure 2 shows that the methods m_{15}, m_{16}, and m_{17} do not share any structural dependencies. If the decomposition was done using only the internal call graph, there would be no incentive to group these together. However, when we "zoom out" by including the client files into the graph, we notice that these methods are all used by the client file f_{14} as denoted by the "D" cells in the last row. This shared client implies that these three methods may fill a similar role in the wider system. After adding clients, we would expect all four items to be grouped together to avoid the penalty of cutting any of these edges.

		m_1	m_2	m_3	m_4	m_5	m_6	m_{15}	m_{16}	m_{17}	f_{14}
m_1	`parseURL`	–	S								
m_2	`parseURLs`	B	–								
m_3	`convertRegister`			–	S	S					
m_4	`convertSubscribe`			S	–		S				
m_5	`revertRegister`			S		–	S				
m_6	`revertSubscribe`				S	S	–				
m_{15}	`isConfigurator`							–			
m_{16}	`isRoute`								–		
m_{17}	`isProvider`									–	
f_{14}	`RegistryDir.java`							D	D	D	–

Fig. 2: Adjacency matrix of G, truncated
D: Structural Dependencies (E_d); S: Semantic Similarites (E_s); B: Both $(E_d \cap E_s)$

2) Identifier Semantics: Now we extend G_d to include undirected edges between internal entities that have semantically similar names. An entity name often contains a number of keywords. For instance, `isRoute` contains `is` and `route`. Their semantic similarity is calculated using a semantic vector space model formed by these keywords [42], [43].[3]

Let the edge set E_s contain the semantic edges between internal entities. Each undirected edge $\{u, v\} \in E_s$ has a cost $cost_s(u, v) \in [0, 1]$ equal to the semantic similarity score between the entities u and v. Because these scores are calculated using a vector space model, almost every pair of entities will have a nonzero score, even if their semantic similarity is negligible. We define a *semantic threshold*, thr_s. Semantic edges with a $cost_s < thr_s$ are deemed negligible and dropped. This creates a more sparse graph which improves the performance of graph processing. Finally, we define this graph with comprehensive information as $G = (V, E_d, E_s)$ to be a triple of 1) internal entities and external clients, 2) directed dependency edges, and 3) undirected semantic edges.

The inclusion of semantic edges reveals shared concepts that may not be reflected by structural dependencies. In Figure 2, notice how the methods m_3, m_4, m_5, and m_6 do not share any direct structural dependencies or clients. However, they do share many semantic edges with one another, as denoted by "S" cells. A pair of entities can have both semantic and structural edges between them. The "B" cell denotes that m_2 structurally uses m_1 while also having a semantically similar name. Their names make it clear that these methods belong to a cohesive group.

B. Condensation

To ensure a decomposition without cyclical dependencies, we start by constructing a condensation graph. If any two vertices $\{u, v\} \subseteq V$ are mutually reachable through the edges of E_d, then they are merged into the same condensed

[2]This class is identified as `dubbo-5318` in our replication package. Source code available at https://github.com/apache/dubbo/blob/dubbo-3.1.10/dubbo-common/src/main/java/org/apache/dubbo/common/utils/UrlUtils.java.

[3]We use mutual information as the term-weighting scheme and correlation as the similarity measure, following Kiela and Clark [43].

		1				2	
		3		4			
		5	6	7	8		
m_1	parseURL			×			
m_2	parseURLs			×			
m_3	convertRegister	×					
m_4	convertSubscribe	×					
m_5	revertRegister	×					
m_6	revertSubscribe	×					
m_7	revertNotify	×					
m_8	revertForbid	×					
m_9	getEmptyUrl			×			
m_{10}	isMatchCategory		×				
m_{11}	isMatch		×				
m_{12}	isMatchGlobPattern		×				
m_{13}	isServiceKeyMatch	×					
m_{14}	classifyUrls			×			
m_{15}	isConfigurator					×	
m_{16}	isRoute					×	
m_{17}	isProvider					×	
m_{18}	isRegistry						+
m_{19}	hasServiceDiscoveryRegi...				×		
m_{20}	hasServiceDiscoveryRegi...				×		
m_{21}	isServiceDiscoveryURL				×		
m_{22}	isItemMatch		×				
m_{23}	parseServiceKey						+
m_{24}	valueOf						+
m_{25}	isConsumer						+
f_1	RegistryProtocol.java	×					
f_2	InjvmProtocol.java	×					
f_3	AbstractRegistry.java		×				
f_4	MulticastRegistry.java		×				
f_5	NacosRegistry.java		×				
f_6	ZookeeperRegistry.java		×				
f_7	ConfigCenterConfig.java			×			
f_8	MetricsConfig.java			×			
f_9	ConfigValidationUtils.java			×			
f_{10}	OfflineApp.java				×		
f_{11}	OfflineInterface.java				×		
f_{12}	OnlineApp.java				×		
f_{13}	ServiceCheckUtils.java				×		
f_{14}	RegistryDirectory.java					×	

Fig. 3: File decomposed by Deicide

×: The entity on the row is aggregated into the cluster(s) on the column;

+: Single entity cluster.

		13							
		12	11						
			10	9					
				8	7				
					6	5			
						4	3		
							2	1	
m_1	parseURL			×					
m_2	parseURLs								+
m_3	convertRegister								+
m_4	convertSubscribe								+
m_5	revertRegister								+
m_6	revertSubscribe								+
m_7	revertNotify								+
m_8	revertForbid	×							
m_9	getEmptyUrl								+
m_{10}	isMatchCategory								+
m_{11}	isMatch		×						
m_{12}	isMatchGlobPattern								+
m_{13}	isServiceKeyMatch		×						
m_{14}	classifyUrls								+
m_{15}	isConfigurator							×	
m_{16}	isRoute							×	
m_{17}	isProvider						×		
m_{18}	isRegistry					×			
m_{19}	hasServiceDiscover...								+
m_{20}	hasServiceDiscover...					×			
m_{21}	isServiceDiscoveryURL								+
m_{22}	isItemMatch								+
m_{23}	parseServiceKey								+
m_{24}	valueOf								+
m_{25}	isConsumer				×				

Fig. 4: File decomposed by JDeodorant

×: The entity in the row is aggregated into the cluster(s) on the column;

+: Single entity cluster.

vertex v', so $\{u, v\} \subseteq v'$. Let $G' = (V', E'_d, E'_s)$ be the condensation graph of G. The weight $w(v')$ of each condensed vertex $v' \in V'$ is the sum of its members. The dependency cost $cost_d(u', v')$ of any condensed edge $(u', v') \in E'_d$ is the maximum $cost_d(u, v)$ obtained for any $u \in u'$ and $v \in v'$. The semantic cost $cost_s$ is calculated similarly. This condensation graph ensures that if there is a path from u' to v' in E'_d, then no path from v' to u'. After condensation, completely disconnected components become the top-level clusters of the decomposition, which represent the largest responsibility modules (such as Cluster 1, 2, and 3 in Figure 1d, or Cluster 1 and 2 in Figure 3). These top-level clusters can be further divided, as described in the next section.

C. Partitioning

We continue by dividing each top-level cluster into two smaller clusters using an acyclic partitioning algorithm that we refer to as BISECT [44], [45]. This algorithm ensures: 1) minimal cut-edge costs, 2) roughly equal cluster sizes, and 3) no cycles between clusters. The process is recursively applied to generate finer clusters until a stopping threshold is reached, resulting in the final decomposition. The graph edges and their associated costs have been selected to penalize separating cohesive subsets, thereby maximizing intra-cluster cohesion. Enforcing balanced cluster sizes avoids trivial solutions, such as isolating low-degree vertices [46]. Restricting solutions to acyclic partitions ensures that the resulting clusters correspond to operable extract class opportunities.

Algorithm 1 presents the PARTITION pseudocode. The condensation graph G' is treated as constant, and the algorithm starts with all vertices active ($A \leftarrow V'$), where only active vertices are considered for division. Lines 2–4 check whether the weight of the active set is below the threshold thr_{size}; if so, further division is halted. Lines 5–10 construct maps $\dot{w}$ (active vertex weights) and $\dot{c}$ (edge costs), assigning inactive vertices a weight of zero. On line 11, BISECT creates a balanced, acyclic

Algorithm 1 PARTITION

Require: Vertex set V'
Require: Vertex weight function $w(\cdot)$
Require: Structural edge set E'_d with cost function $cost_d(\cdot,\cdot)$
Require: Semantic edge set E'_s with cost function $cost_s(\cdot,\cdot)$
Require: Weight threshold thr_{size}
1: **procedure** PARTITION(A)
2: **if** $w(A) \leq thr_{size}$ **then**
3: **return**
4: **end if**
5: **for** $v \in V'_i$ **do**
6: $\dot{w}[v] \leftarrow w(v)$ **if** $v \in A$ **else** 0
7: **end for**
8: **for** $u, v \in (E'_d \cup E'_s)$ **do**
9: $\dot{c}[u,v] \leftarrow cost_d(u,v) + cost_s(u,v)$
10: **end for**
11: $P_0, P_1 \leftarrow$ BISECT$(V', \dot{w}, \dot{c})$
12: $P_0 \leftarrow A \cap P_0$
13: $P_1 \leftarrow A \cap P_1$
14: **for** $v \in P_0$ **do**
15: RECORD$(v, \texttt{0})$
16: **end for**
17: **for** $v \in P_1$ **do**
18: RECORD$(v, \texttt{1})$
19: **end for**
20: PARTITION(P_0)
21: PARTITION(P_1)
22: **end procedure**

partition of V', guided by edges to preserve cohesion between related entities. While inactive vertices and clients do not affect the cut balance due to their zero weight, their incoming and outgoing edges influence BISECT, discouraging separation of closely related entities. Lines 12–13 remove inactive vertices, and lines 14–19 use the RECORD subroutine to log which side of the cut each active entity is assigned to by appending a binary value (`0` or `1`) to a string associated with each vertex. The algorithm then recurses on each side of the cut.

The BISECT subroutine solves the acyclic partitioning problem [44]. Although this problem is NP-Hard, the relatively small graph sizes allow for exact solutions to be computed. We use the integer linear programming (ILP) formulation proposed by Özkaya and Çatalyürek [45], solving it with Google OR-Tools [47]. In our experiments, each instance completes within two to three minutes.

D. Qualitative Illustration

In Figure 3, we demonstrate how `UrlUtils.java` is decomposed by Deicide. The rows contain the entities of the class, while the column headers contain the cluster hierarchy. Each yellow "×" cell indicates that the entity on that row belongs to the cluster(s) in the corresponding column header. For example, m_{10}, m_{11}, and m_{12} are aggregated together in Clusters 1, 3, and 6. To save space, clusters containing only a single entity are placed together in the far-right column marked using pink "+" cells.

In Section III-B, we described how a class is condensed. In this case, the condensation graph is the same as the original graph, as it is already acyclic. The disconnected components of the graph form the top-level clusters. Because they do not share any internal dependencies or external clients, notice how Cluster 2 (m_{15}, m_{16}, and m_{17}) is separated from Cluster 1 (containing the other 22 entities and further divided into three layers). Furthermore, there are four entities (m_{18}, m_{23}, m_{24}, and m_{25}) which are entirely isolated and placed in their own singleton clusters. They have no internal dependencies and no shared external dependencies with any other members.

In Section III-C, we described how each top-level cluster is further partitioned. Notice how Cluster 1 is bisected to create Cluster 3 and Cluster 4. Each method shares more internal dependencies, client usages, and semantic similarities with those in the same cluster than with methods in other clusters. The different responsibilities of these clusters are manifested by how client files use them. Notice how Cluster 3 and Cluster 4 are used by entirely different sets of clients, $\{f_1, \ldots, f_6\}$ versus $\{f_7, \ldots, f_{13}\}$. As a result, Cluster 3 and Cluster 4 form separate responsibility modules, and each can be a meaningful extract class opportunity.

Here we highlight the advantages of Deicide that is evident in our running example. Figure 4 illustrates how the same class is decomposed by JDeodorant (using the same format as Figure 3). JDeodorant works bottom-up, starting with Cluster 1 and iteratively merging similar clusters together before terminating at Cluster 13. Each level "wraps" the one below by including an additional one or two methods. Consequently, no two sibling clusters appear to be representing distinct responsibilities. By contrast, Figure 3 shows that in each level of Deicide's decomposition, there are two distinct responsibilities. For example, Deicide identifies that the method sets $\{m_{15}, m_{16}, m_{17}\}$, and $\{m_{19}, m_{20}, m_{21}\}$ address two distinctive responsibilities that can not be further divided (Clusters 2 and 8 in Figure 3), while in JDeodorant's decomposition, the methods $\{m_{19}, m_{20}, m_{21}\}$ are not clustered together at all, while methods $\{m_{15}, m_{16}, m_{17}\}$ are split across two neighboring clusters (Clusters 1 and 2 in Figure 4), and they would only be extracted together during the second round of recommendation.

Figure 5 illustrates how these entities are co-changed as recorded in the revision history. In this figure, the column headers indicate that there are 10 commits that changed these 25 methods. For example, Commit 2 changes both m_1 and m_2, and Commit 4 changes methods m_{19}, m_{20}, and m_{21} together, consistent with Cluster 8 in Deicide. Comparing Figure 5 against Figure 3 and Figure 4, it is clear that Deicide decomposition better conforms to the co-change history of this class. Next, we empirically demonstrate that Deicide decompositions are most aligned with co-change history.

IV. Evaluation

The objective of our evaluation is to assess how well Deicide can identify responsibility modules that can change and evolve independently. We first attempted to mine open-source repositories for confirmed extract class refactorings using a tool such as Refactoring Miner (RM) [48]. Unfortunately, it is rare for a large class to be completely decomposed. For example, using RM, we discovered an Apache file named `ASTGenerator.java` that has 3026 LOC before refactoring. After four new classes are extracted, 2052 LOC still remains. It is hard to tell if a mined refactoring has split out all responsibility modules that *should* be refactored out.

Instead, we use co-change history as an objective baseline: if a group of methods are created to accomplish the same responsibility, they will more likely be changed together and by the same authors. Since Gall et al. [49] proposed the concept of *evolutionary coupling*, co-change history has been widely used to predict future changes, bug locations, and who should fix a bug, and to identify code smells and architecture violations [27]–[39], based on the rationale that frequent co-changes and co-authorship are indications of cohesive units. Consequently, we investigate the two research questions proposed in Section I to evaluate the derived clusterings for all four algorithms.

RQ1: *Which algorithm(s) decompose entities into clusters such that the clustered entities were changed together more often, as recorded in a project's revision history?* In other words, an effective decomposition should be able to identify clustered entities that usually evolve together.

RQ2: *Which algorithm(s) decompose entities into clusters such that the grouped entities were likely to be modified by the same authors, according to the project's commit history?* That is, clusters decomposed by an effective algorithm should contain entities maintained by similar authors, an objective reflection of cohesive responsibilities.

Next, we introduce the subject selection, measurements, and statistical analysis of our evaluation process, as well as the results that answer these questions.

A. Subject Selection

Although we can apply DC, AK, and AL to any source files, the application of JD is more complicated. The JDeodorant plug-in for Eclipse requires a project to be compiled first before generating refactoring suggestions. To compare against JDeodorant, we must select projects that can be built in Eclipse. From our original pool of the 30 most popular Apache projects on GitHub with more than 2,000 Java source files, we were able to compile 18 of them within Eclipse without making extensive modifications to their build configurations.

Considering that the number of files and complexity levels can vary considerably on a project-to-project basis, we select the files that are in the top 10% of their project by lines of code (LOC), as extract class recommendations are most relevant to large classes. However, for some smaller projects, their top 10% still includes some fairly small files, e.g., around 200 LOC. Thus, we only select classes with 500 LOC or more.

TABLE I: Projects Mined for Subjects

Project	Tag	Start	End	Files
ActiveMQ	`activemq-5.18.1`	12/12/2005	4/10/2023	3.9K
Dubbo	`dubbo-3.1.10`	10/20/2011	4/17/23	3.8K
Hudi	`release-0.12.3`	12/16/2016	4/23/23	2.6K
IoTDB	`v1.1.0`	5/7/2017	3/31/23	4.7K
Kafka	`3.4.0`	8/1/2011	1/31/23	4.3K
Log4j 2	`rel/2.20.0`	5/13/2010	2/17/23	2.6K
NiFi	`rel/nifi-1.21.0`	12/8/2014	4/3/23	7.7K
Pinot	`release-0.12.1`	10/31/2014	2/27/23	3K
ShenYu	`v2.5.1`	7/11/2018	1/30/23	2.6K

Next, since we need to compare our decomposition with the revision history, we filter out files that were never revised, revised just once, or revised by only one author. Finally, because JDeodorant uses Eclipse to extract dependencies, while Deicide uses Depends [50], an open-source dependency analysis tool, they sometimes disagree on the exact entities contained within a file. We remedy this by filtering out any files where more than two entities could not be co-identified. This leaves us with 123 subjects spanning 9 projects. Table I lists the projects while Table II lists the subjects—truncated to only show those that are larger than 1000 LOC for the sake of space. Our subjects range from 503 to 3633 LOC where each one is in the 90th percentile of their project in terms of LOC. Our subjects contain dozens to hundreds of entities and revised by 4 to 50 authors.

B. Measurements

Given the decompositions produced by the four algorithms, we use the revision history of each project as the ground truth and comparison baseline. For this purpose, we consider the revision history of a file as a clustering: each commit is a "cluster" containing the entities that it has created or modified. Similarly, each author of a class can also be viewed as a "cluster" containing the entities that they created or modified. Figure 5 presents an example of how the commit history of a class can be represented as a clustering. It is rare for a field to be modified while maintaining the same identity, so we only consider methods in our evaluation.

To answer these research questions, we measure the degree to which the decompositions produced by these four algorithms conform to the revision histories, both in terms of commits and authors, of our selected subjects. We use the grand index (GRI) of Horta and Campello [51] for this task. The GRI is a score between 0 and 1 that measures the similarity between two clusterings, identifying the one most similar to a ground truth. While there are many methods available to measure the similarity between two clusterings, the GRI is one of the few metrics specifically designed to support *non-exclusive* clustering. [4] The clusterings derived from the revision history are clearly non-exclusive. Entities can (and will) be changed by multiple commits and authors. Furthermore, the

[4]The adjusted grand index (AGRI) introduced in the same paper is another option but tends to overly penalize clusterings that diverge from the ground truth cluster count. Since the number of commits is often far more than decomposed clusters, the GRI is more practical for our evaluation.

		1	2	3	4	5	6	7	8	9	10
m_1	parseURL	×	×							×	×
m_2	parseURLs	×	×								
m_3	convertRegister	×						×			
m_4	convertSubscribe	×		×							
m_5	revertRegister	×		×				×			
m_6	revertSubscribe	×		×							
m_7	revertNotify	×						×			
m_8	revertForbid	×									
m_9	getEmptyUrl	×									
m_{10}	isMatchCategory	×									
m_{11}	isMatch	×		×		×	×			×	
m_{12}	isMatchGlobPat...	×									
m_{13}	isServiceKeyMatch	×								×	
m_{14}	classifyUrls	×									
m_{15}	isConfigurator	×					×			×	
m_{16}	isRoute	×					×			×	
m_{17}	isProvider	×					×			×	
m_{18}	isRegistry	×					×			×	
m_{19}	hasServiceDisc...	×			×		×				
m_{20}	hasServiceDisc...	×			×						
m_{21}	isServiceDisco...	×			×						
m_{22}	isItemMatch	×					×				
m_{23}	parseServiceKey	×									
m_{24}	valueOf	×								×	×
m_{25}	isConsumer	×					×		×		

Fig. 5: Revision history as a non-exclusive clustering
×: The entity on the row is modified by the commit on the column.

decompositions produced by these four algorithms are also non-exclusive: entities do not belong to a single cluster but to a hierarchy of clusters. For instance, in Figure 3, `parseURL` equally belongs to Clusters 1, 4, and 7.

For the i-th subject we analyzed, let DC_i, JD_i, AK_i, and AL_i be the decomposition produced by the four algorithms, and C_i and A_i be the clusterings derived from the commits and authors respectively. We define "*commit similarity*" as $\text{CS}[x_i] = \text{GRI}[C_i, x_i]$ where x_i is a decomposition of the i-th subject. Likewise, "*author similarity*" is defined as $\text{AS}[x_i] = \text{GRI}[A_i, x_i]$. To answer RQ1 and RQ2, we analyze the two similarity scores between two decompositions. For example, to test if $\text{CS}[JD_i] > \text{CS}[DS_i]$, that is, the JD decomposition better conforms to the revision history than that of Deicide, we collect the commit similarity scores for all 123 files and statistically test this hypothesis as elaborated in Section IV-D.

C. Data Collection and Execution

We extract both historical and structural data from the 123 files identified in Section IV-A. First, we use Depends [50] to extract syntactical relationships between files, including the specific entities involved. Next, we created scripts to collect method-level commit data, leveraging libgit2 [52] to interact with the repositories, and tree-sitter [53] to quickly parse the Java source files found. For each project, we store the entities, dependencies, commits, and changes in a SQLite database.

We ran Deicide on each subject using a semantic threshold $thr_s = 0.25$ and entity size threshold $thr_{size} = 7$, the latter being the median number of entities per class across our selected projects. For the remaining three algorithms, we use the thresholds recommended by their respective authors and ran them on the same subjects. The entire process is completed in 52 minutes on a 2023 Apple MacBook Pro with an M2 Pro CPU and 16 GB of RAM.

Table II presents the results from the largest 25 files. Subjects are sorted in descending order by LOC. The number of entities, client files, commits, and authors for each subject are listed in the Ent., Cli., Com., and Aut. columns respectively. The next eight columns show the commit and author similarity scores for the four algorithms as percentages. The cell with the best index score is highlighted. For example, the first row shows a file found in Pinot that has 3633 LOC, contains 212 entities, is depended on by 110 external clients, has been involved in 210 commits, and revised by 49 authors. Deicide decomposed this file into 43 responsibility modules, and this decomposition conforms to the revision history better than the other three, in terms of commits and authorship. The final column, Res., lists the number of responsibility modules discovered. The full table, and all other artifacts, can be found in our replication package [54].

D. Statistic Analysis and Results

To assess which statistic test should be used to evaluate the two sets of four measures: $\text{CS}[DC_i]$, $\text{CS}[JD_i]$, $\text{CS}[AK_i]$, $\text{CS}[AL_i]$, and $\text{AS}[DC_i]$, $\text{AS}[JD_i]$, $\text{AS}[AK_i]$, $\text{AS}[AL_i]$, we first use the Shapiro-Wilk test [55], [56] to determine that most of this data is not normally distributed. Accordingly, we apply the Mann-Whitney U test [56], [57]. The p-values of the pairwise tests are presented in Table III. Each cell is a p-value of the test between the data set on the column and the one on the row. For example, Table II shows that the similarity scores in the $\text{CS}[DC_i]$ column are all higher than that of $\text{CS}[AK_i]$ except for one case; and in Table III, $2.17E-08$ in row AK, column DC means that the differences are significant (the p-value is much lower than 0.05). That is, the commit similarity between DC and the co-change history are much higher than that of AK. These tables confirm the observation we conveyed in Section II: all the p-values are very small, meaning that these algorithms produce drastically different decompositions. Moreover, our DC is the winner in terms of both commit similarity and author similarity.

Table IV presents the max, median, and min of the two similarity scores of these four algorithms. For example, the commit similarities of Deicide range from 3.44% to 75.00% with a median of 33.22%. The table reveals that the median scores of DC, for both commit and author similarity, are much higher than that of AK, JD, and AL. The authorship results are similar: authorship similarities of Deicide range from 2.92% to 72.36% with an median of 25.81%, which is much higher than the median score of AK, JD, or AL. Now we are ready to answer our two research questions:

Answer to RQ1: Deicide shows a much stronger alignment with the co-change history than that of AK, JD, and AL, meaning that if these responsibility modules were extracted

TABLE II: Results: Decompositions vs. History

#	Project	LOC	Ent.	Cli.	Com.	Aut.	CS[DC_i]	CS[JD_i]	CS[AK_i]	CS[AL_i]	AS[DC_i]	AS[JD_i]	AS[AK_i]	AS[AL_i]	Res.
1	Pinot	3633	212	110	210	49	23.55%	0.50%	5.62%	8.09%	15.99%	0.33%	3.57%	5.11%	43
2	ActiveMQ	3170	409	1082	131	26	14.30%	0.88%	6.84%	1.73%	7.89%	0.45%	3.37%	0.89%	88
3	Log4j 2	2848	474	112	148	21	48.25%	1.81%	9.46%	0.16%	34.51%	0.75%	4.14%	0.06%	12
4	ActiveMQ	2548	275	225	60	14	8.03%	0.51%	2.48%	2.47%	6.09%	0.38%	1.78%	1.80%	58
5	ActiveMQ	2154	143	56	43	12	12.74%	0.94%	6.69%	2.02%	9.85%	0.75%	4.68%	1.35%	30
6	IoTDB	2148	103	96	205	50	31.08%	12.50%	52.76%	39.22%	54.85%	6.74%	34.12%	30.85%	22
7	Dubbo	1686	231	884	98	33	50.59%	4.64%	14.31%	1.08%	26.98%	2.11%	6.36%	0.48%	38
8	ActiveMQ	1620	156	29	91	18	31.04%	4.14%	10.81%	5.05%	20.89%	2.29%	6.48%	3.00%	38
9	Kafka	1603	56	2	59	18	46.14%	4.68%	20.03%	2.06%	47.04%	2.86%	12.33%	1.29%	12
10	ActiveMQ	1588	112	24	63	13	34.78%	2.05%	10.81%	2.48%	20.55%	0.93%	5.15%	1.13%	22
11	NiFi	1574	98	4	57	22	30.85%	6.78%	20.63%	12.74%	19.54%	3.77%	11.16%	6.60%	21
12	Pinot	1425	76	8	85	26	43.71%	4.66%	18.23%	11.28%	24.78%	2.00%	7.96%	5.15%	16
13	IoTDB	1423	71	2	58	15	54.27%	37.00%	28.09%	2.22%	60.21%	33.26%	22.85%	1.80%	29
14	ActiveMQ	1416	150	20	55	14	26.54%	3.79%	12.85%	2.79%	17.03%	2.05%	6.36%	1.42%	32
15	NiFi	1372	111	23	43	18	33.91%	2.24%	9.61%	2.85%	10.39%	0.66%	2.74%	0.80%	21
16	Dubbo	1333	81	39	31	18	43.01%	15.00%	14.63%	1.10%	41.13%	15.86%	14.01%	1.04%	27
17	Dubbo	1289	100	257	118	48	63.06%	18.78%	40.56%	4.05%	47.08%	11.64%	28.34%	2.40%	22
18	Hudi	1273	47	19	92	24	51.95%	33.49%	27.40%	7.38%	33.52%	52.69%	13.22%	3.58%	11
19	ActiveMQ	1272	196	768	26	9	29.67%	26.31%	17.02%	3.90%	14.46%	13.81%	7.26%	1.98%	42
20	Hudi	1228	108	25	81	36	25.11%	4.55%	10.00%	1.22%	16.50%	2.75%	6.09%	0.76%	29
21	Dubbo	1205	89	260	37	22	48.11%	10.91%	16.65%	0.95%	42.17%	12.56%	14.29%	0.82%	24
22	ActiveMQ	1129	66	19	21	10	37.78%	3.28%	11.97%	1.46%	32.52%	2.73%	9.86%	1.22%	15
23	ActiveMQ	1120	205	237	37	12	30.78%	20.97%	16.96%	11.06%	17.90%	12.80%	9.65%	5.92%	41
24	ActiveMQ	1114	43	28	11	6	48.86%	7.75%	19.34%	4.13%	26.46%	3.94%	9.43%	2.06%	10
25	Pinot	1065	33	2	13	7	48.39%	15.53%	35.27%	11.85%	34.22%	11.64%	24.82%	9.23%	7

Ent.: Entities; **Cli.**: Clients; **Com.**: Commits; **Aut.**: Authors; $\mathbf{CS}[\boldsymbol{x_i}]$: Commit similarity; $\boldsymbol{AS}[\boldsymbol{x_i}]$: Author similarity; **Res.** Responsibility modules from Deicide

TABLE III: Mann-Whitney U Test Results

	Commit Similarity (CS)				Author Similarity (AS)			
	DC	AK	JD	AL	DC	AK	JD	AL
DC	-				-			
AK	2.17E-08	-			1.38E-11	-		
JD	2.22E-16	1.04E-02	-		2.40E-14	1.04E-02	-	
AL	0.00E+00	3.89E-11	2.22E-16	-	0.00E+00	2.00E-10	9.50E-05	-

TABLE IV: Results Summary

	Commit Similarity (CS)				Author Similarity (AS)			
	DC	AK	JD	AL	DC	AK	JD	AL
Max	75.00%	66.67%	67.26%	50.49%	72.36%	66.67%	66.67%	60.12%
Median	33.32%	12.45%	17.70%	7.07%	25.81%	7.89%	10.11%	3.96%
Min	3.44%	0.26%	0.48%	0.08%	2.92%	0.20%	0.46%	0.06%

into new classes, they would be more likely to change and evolve independently.

Answer to RQ2: Deicide demonstrates a much stronger alignment with authorship history, indicating that if a responsibility module is extracted into a new class, it can be maintained by the same authors who previously worked on its entities.

V. Discussion

A. Threats to Validity

An external threat is that we only compared Deicide with three existing approaches—the most well-known, or those presented in recent publications. Since JDeodorant requires compiled binary Java code, our evaluation was limited to Java projects that can be compiled and built in Eclipse. Our results could be different for other projects, particularly those written in different programming languages.

We also note several threats to internal validity. First, the accuracy of dependency extraction largely depends on the accuracy of the tools used as a preprocessor. Deicide uses Depends [50]—an open-source dependency extraction tool, to extract dependencies, while JDeodorant uses Eclipse. We have noticed minor differences between the dependency relations extracted by both tools. To mitigate this threat, we have manually checked the results to make sure there were no significant differences, but we should evaluate the impact of such differences automatically in the future. Second, the stopping criterion for Deicide is the median number of entities of all classes, which may or may not be the best choice to find the optimal decomposition. It is also possible that thresholds for different projects can be different. We will also explore this parameter in future work. Third, Deicide leverages semantic similarity among entity names, assuming that the system follows a strict naming convention, where each entity is named based on underlying concepts. The degree to which this is true will impact the effectiveness of Deicide. Finally, our selection of subjects from the top 10% LOC of a project is another threat. A large class can have a single or a few responsibilities, and a smaller file might also be a god class. Exploring effective tools that can be used to identify complex files that need refactoring is our future work.

B. Future Work

As revealed by recent industrial experience report [13], class decomposition is a complicated problem that cannot be fully automated. Instead, the responsibility modules produced by Deicide can be used in the "planning stage" proposed by Anquetil et al. [13] and Malavolta et al. [14], and an architect should review the resulting clusters and choose the ones that should be refactored based on their domain knowledge, business goals, and estimated costs. After that, the chosen refactoring can be automated. Creating such a complete and interactive framework is our future work.

Currently, we evaluate Deicide using revision history. Another promising direction would be to apply Deicide to decompose a god class in an early version of the project and then analyze its subsequent evolution to determine whether

the proposed decomposition aligns with how the project naturally evolved. This forward-looking approach complements the current backward-looking analysis of revision history.

Another important future task is to apply Deicide to more systems written in other programming languages, and to assess the impact of using different dependency extraction tools. As a matter of fact, dependency extraction is a fundamental step for much design related research, but the accuracy and consistency of these tools has not been properly evaluated.

Finally, different clustering methods may perform better in different scenarios. For instance, if a complex class has only one or two primary responsibilities rather than multiple distinct ones, these lopsided algorithms may generate better results. To explore this further, we plan to conduct more empirical studies in the future, and interview experienced software architects to gather their feedback.

VI. Related Work

Our work is most related to extract class refactoring algorithms that have been proposed to address large or "god" classes—the most prominent code smells. It is also related to software redesign/refactoring to improve modularity and evolution in general.

Fowler introduced the term "code smell" [1] to capture problems that make code harder to change, maintain, or test in the long run. Since then, the detection [2]–[4], [6], [58] and removal [7]–[12], [18], [19] of code smells have been widely studied. In this paper, we focus on the class decompositions used by extract class recommenders to decompose large and complex classes [10], [15]–[26]. These files are often foci of maintenance difficulty, and interleave with other smells or anti-patterns [13], [59], [60]. Dallal's survey summarized the methods proposed to detect god classes [61], using metrics, clustering, graphs, etc. As Anquetil et al. pointed out [13], god classes are usually easy to spot, and developers do not need sophisticated methods to detect them. In this paper, we chose the largest active files in each project as our subjects.

Several methods have been proposed to identify extract class refactoring opportunities. De Lucia et al. [15] were among the first to leverage structural and semantic cohesion metrics to decompose large classes, employing a MaxFlow-MinCut algorithm [62] to split a weighted graph of class entities into cohesive subgraphs. Building on this, Bavota et al. [16], [17], [20] introduced an approach that integrated Latent Semantic Indexing (LSI) [63] to measure semantic similarity alongside structural dependencies. Concurrently, Fokaefs et al. [10], [18], [19] developed JDeodorant, which uses a Hierarchical Agglomerative Clustering (HAC) algorithm to discover extract class opportunities by analyzing the internal call graph.

Akash et al. [21] extended Bavota's work by replacing LSI with Latent Dirichlet Allocation (LDA) [64] for processing semantic information and merging entities based on a combination of structural and semantic metrics. Unlike Bavota's approach, which focused on binary or partitional class splits, Akash's method employs a HAC algorithm to produce a hierarchical decomposition.

Alzahrani et al. [22]–[24] introduced another notable approach, becoming the first to incorporate external client relationships into the decomposition process. Unlike Akash and Bavota, Alzahrani's method measures coupling based on shared client dependencies, resulting in decompositions that emphasize interactions between the class and external systems.

Search-based software engineering methods have addressed the problem of automated software refactoring [65]–[70]. The work of Ivers et al, Masoud et al, and Schröder et al. are the most recent advances in this area [71]–[73]. These methods typically search for a sequence of refactorings that optimize for some global quality metrics, seeking to recluster a large number of classes, rather than a single complex class as we do. Researchers have also recognized that refactoring can not and should not be fully automated [74], [75]. Anquetil et al. [13] proposed a "planning" stage before refactoring, which was done manually, supported by visualization. Deicide is designed to be used in such a planning stage, providing an automated decomposition recommendation.

VII. Conclusion

This paper presented our approach to derive a holistic decomposition of a large, complex class as a set of responsibility modules. After analyzing the decomposed structures derived from existing hierarchical decomposition algorithms, we presented Deicide, a novel algorithm that constructs a graph from three distinct types of relations: internal entity dependencies, external client-to-entity usage relations, and identifier semantic similarity. Using this graph, we transformed the class decomposition problem into an acyclic partitioning problem to generate a balanced, acyclic, and hierarchical structure where each cluster represents a responsibility module. Finally, we empirically evaluated Deicide, comparing it with the three state-of-the-art algorithms, using 123 classes identified from 9 open-source projects as subjects.

To assess how well the decomposed clusters can change and evolve independently, we compared the resulting decompositions against each project's revision history, checking to what extent the entities in the same cluster were actually changed together, and by the same authors. Our statistical analysis revealed that the responsibility modules recommended by Deicide are significantly more aligned with the project's revision history than the other three approaches. The implication is that if a class is refactored based on Deicide's recommendation, these newly created classes will be much more independent, making them easier to understand, maintain, and evolve.

VIII. Data Availability

We provide all data along with a replication package at [54].

IX. Acknowledgements

This work was supported in part by the National Science Foundation of the US under grants CCF-2232720, CCF-2232721, CCF-2213764, and TI-2236824.

REFERENCES

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Jul. 1999.

[2] L. Tahvildari and K. Kontogiannis, "A metric-based approach to enhance design quality through meta-pattern transformations," in *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, 2003, pp. 183–192.

[3] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," in *12th Working Conference on Reverse Engineering (WCRE'05)*, 2005, pp. 10 pp.–164.

[4] A. Chatzigeorgiou, S. Xanthos, and G. Stephanides, "Evaluating object-oriented designs with link analysis," in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 656–665.

[5] P. Joshi and R. K. Joshi, "Concept analysis for class cohesion," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 237–240.

[6] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*, 2009, pp. 305–314.

[7] C. Y. Chong, S. P. Lee, and T. C. Ling, "Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach," *Information and Software Technology*, vol. 55, no. 11, pp. 1994–2012, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584913001481

[8] A. Rao and K. Reddy, "Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique," *CoRR*, vol. abs/1201.1611, 01 2012.

[9] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, 2001, pp. 30–38.

[10] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique," in *2009 IEEE International Conference on Software Maintenance*. IEEE, Sep. 2009, pp. 93–101.

[11] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, 1999, pp. 109–118.

[12] K. Solanki and S. Kumari, "Comparative study of software clone detection techniques," in *2016 Management and Innovation Technology International Conference (MITicon)*, 2016, pp. MIT–152–MIT–156.

[13] N. Anquetil, A. Etien, G. Andreo, and S. Ducasse, "Decomposing god classes at siemens," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 169–180.

[14] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Transactions on Software Engineering*, vol. 39, 12 2012.

[15] A. De Lucia, R. Oliveto, and L. Vorraro, "Using structural and semantic metrics to improve class cohesion," in *2008 IEEE International Conference on Software Maintenance*. IEEE, Sep. 2008, pp. 27–36.

[16] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "A two-step technique for extract class refactoring," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE10, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, Sep. 2010, pp. 151–154.

[17] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, no. 3, pp. 397–414, Mar. 2011.

[18] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE11, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, May 2011, pp. 1037–1039.

[19] ——, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, Oct. 2012.

[20] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, May 2013.

[21] P. Akash, A. Sadiq, and A. Kabir, "An approach of extracting god class exploiting both structural and semantic similarity," in *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, E. Damiani, G. Spanoudakis, and L. A. Maciaszek, Eds. SCITEPRESS - Science and Technology Publications, 2019, pp. 427–433.

[22] M. Alzahrani and S. Alqithami, "An external client-based approach for the extract class refactoring: A theoretical model and an empirical approach," *Applied Sciences*, vol. 10, no. 17, p. 6038, Aug. 2020.

[23] M. Alzahrani, *Using Clients to Support Extract Class Refactoring*. Springer International Publishing, 2021, pp. 695–704.

[24] ——, "Extract class refactoring based on cohesion and coupling: A greedy approach," *Computers*, vol. 11, no. 8, p. 123, Aug. 2022.

[25] T. Chen, Y. Jiang, F. Fan, B. Liu, and H. Liu, "A position-aware approach to decomposing god classes," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: ACM, Oct. 2024, pp. 129–140.

[26] D. Cui, Q. Wang, Y. Zhao, J. Wang, M. Wei, J. Hu, L. Wang, and Q. Li, "One-to-one or one-to-many? suggesting extract class refactoring opportunities with intra-class dependency hypergraph neural network," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '24, M. Christakis and M. Pradel, Eds. ACM, Sep. 2024, pp. 1529–1540.

[27] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proc. 26th*, May 2004, pp. 563–572.

[28] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *ICSE11: International Conference on Software Engineering*. New York, NY, USA: ACM, 5 2011.

[29] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Proc. 4th*, May 2007.

[30] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," vol. 18, pp. 207–236, 2006.

[31] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.

[32] D. Rapu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* IEEE, 2004, pp. 223–232.

[33] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 268–278.

[34] D. Sas, P. Avgeriou, R. Kruizinga, and R. Scheedler, "Exploring the relation between co-changes and architectural smells," *SN Computer Science*, vol. 2, pp. 1–15, 2021.

[35] A. Saydemir, M. E. Simitcioglu, and H. Sozer, "On the use of evolutionary coupling for software architecture recovery," in *2021 15th Turkish National Software Engineering Symposium (UYMS)*, 2021, pp. 1–6.

[36] H. Yapici and H. Sözer, "Coevolution index: A metric for tracking evolutionary coupling," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2023, pp. 1582–1587.

[37] R. Mo and Z. Yin, "Exploring software bug-proneness based on evolutionary clique modeling and analysis," *Information and Software Technology*, vol. 128, p. 106380, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920301476

[38] A. Imran, "Design smell detection and analysis for open source java software," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 644–648.

[39] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 361–370. [Online]. Available: https://doi.org/10.1145/1134285.1134336

[40] M. Ackerman and S. Ben-David, "A characterization of linkage-based hierarchical clustering," *J. Mach. Learn. Res.*, vol. 17, pp. 232:1–232:17, 2016. [Online]. Available: https://jmlr.org/papers/v17/11-198.html

[41] M. Roux, "A comparative study of divisive and agglomerative hierarchical clustering algorithms," *Journal of Classification*, vol. 35, no. 2, pp. 345–366, Jul. 2018.

[42] P. D. Turney and P. Pantel, "From frequency to meaning: Vector space models of semantics," *J. Artif. Int. Res.*, vol. 37, no. 1, p. 141–188, jan 2010.

[43] D. Kiela and S. Clark, "A systematic study of semantic vector space model parameters," in *Proceedings of the 2nd Workshop on Continuous*

Vector Space Models and their Compositionality (CVSC), 2014, pp. 21–30.

[44] J. Nossack and E. Pesch, "Mathematical formulations for the acyclic partitioning problem," in *Operations Research Proceedings 2013*, D. Huisman, I. Louwerse, and A. P. Wagelmans, Eds. Cham: Springer International Publishing, 2014, pp. 333–339.

[45] M. Y. Özkaya and Ü. V. Çatalyürek, "A simple and elegant mathematical formulation for the acyclic dag partitioning problem," *arXiv preprint arXiv:2207.13638*, 2022.

[46] U. von Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, vol. 17, no. 4, pp. 395–416, Aug. 2007. [Online]. Available: https://doi.org/10.1007/s11222-007-9033-z

[47] "Google or-tools." [Online]. Available: https://developers.google.com/optimization

[48] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[49] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. 14th*, Nov. 1998, pp. 190–197.

[50] Z. Gang, "Depends." [Online]. Available: https://github.com/multilang-depends/depends

[51] D. Horta and R. J. G. B. Campello, "Comparing hard and overlapping clusterings," *J. Mach. Learn. Res.*, vol. 16, no. 1, p. 2949–2997, jan 2015.

[52] "libgit2." [Online]. Available: https://libgit2.org/

[53] "Tree-sitter." [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[54] Anonymous, "Deicide: Decomposing God Files Into Responsibility Modules," 2024. [Online]. Available: https://doi.org/10.5281/zenodo.14183961

[55] S. S. SHAPIRO and M. B. WILK, "An analysis of variance test for normality (complete samples)†," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, 12 1965. [Online]. Available: https://doi.org/10.1093/biomet/52.3-4.591

[56] A. P. Field, *Discovering statistics using IBM SPSS statistics*. Sage Publications, 2013.

[57] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50 – 60, 1947. [Online]. Available: https://doi.org/10.1214/aoms/1177730491

[58] P. Joshi and R. K. Joshi, "Concept analysis for class cohesion," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 237–240.

[59] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture Anti-Patterns: Automatically Detectable Violations of Design Principles," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 1008–1028, 5 2021.

[60] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *ICSE '16: 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 5 2016.

[61] J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, vol. 58, pp. 231–249, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584914001918

[62] R. L. R. Thomas H Cormen, Charles E Leiserson and C. Stein, *Introduction to Algorithms*. MIT Press, 2022.

[63] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[64] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[65] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *the 9th annual conference*. New York, New York, USA: ACM Press, 1 2007.

[66] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 3 2013.

[67] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *ASE '14: ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 9 2014.

[68] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-Objective Software Remodularization Using NSGA-III," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 3, pp. 1–45, 5 2015.

[69] V. Alizadeh, M. A. Ouali, M. Kessentini, and M. Chater, "RefBot: Intelligent Software Refactoring Bot," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 11 2019.

[70] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *FSE'16: 24nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 11 2016.

[71] J. Ivers, C. Seifried, and I. Ozkaya, "Untangling the knot: Enabling architecture evolution with search-based refactoring," in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, 2022, pp. 101–111.

[72] M. Aghdasifam, H. Izadkhah, and A. Isazadeh, "A new metaheuristic-based hierarchical clustering algorithm for software modularization," *Complexity*, vol. 2020, p. 25, 09 2020.

[73] C. Schröder, A. van der Feltz, A. Panichella, and M. Aniche, "Search-based software re-modularization: a case study at adyen," in *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '21. IEEE Press, 2021, p. 81–90. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP52600.2021.00017

[74] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 211–221.

[75] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.