Delft University of Technology
Software Engineering Research Group
Technical Report Series

Using Cluster Analysis to Improve the Design of Component Interfaces

Rahmat Adnan, Bas Graaf, Arie van Deursen
and Joost Zonneveld

Report TUD-SERG-2008-026

TU Delft

SERG

TUD-SERG-2008-026

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: A short (4-page) version of this paper will appear in the Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2008, IEEE Computer Society.

Using Cluster Analysis to Improve the Design of Component Interfaces

Rahmat Adnan
Delft University of Technology
The Netherlands
radnan@gmail.com

Bas Graaf
Delft University of Technology
The Netherlands
b.s.graaf@tudelft.nl

Arie van Deursen
Delft University of Technology
The Netherlands
arie.vandeursen@tudelft.nl

Joost Zonneveld
ASML
The Netherlands
Joost.Zonneveld@asml.com

Abstract

For large software systems, interface structure has an important impact on their maintainability and build performance. For example, for complex systems written in C, recompilation due to a change in one central header file can run into hours. In this paper, we explore how automated cluster analysis can be used to refactor interfaces, in order to reduce the number of dependencies and to improve encapsulation, thus improving build performance and maintainability. We implemented our approach in a tool called "Interface Regroup Wizard", which we applied to several interfaces of a large industrial embedded system. From this, we not only learned that automated cluster analysis can indeed help to improve the design of interfaces, but also which of the final refactoring steps are best done manually by an architect.

1 Introduction

As a software system evolves over many years, the interfaces between its source modules are modified as well. Over time, interface structure is likely to deteriorate. This might result in fat interfaces with with huge numbers of definitions that are not functionally coherent, and which have many different source files ("users") depending on them. Especially in the case of large software systems this negatively affects build performance and maintainability. In the C programming language, for instance, a source file can use an interface definition by including a complete header file. As a result, for every change to a single definition in that header file, all source files that use one or more of its definitions need to be recompiled.

As an example, at ASML, a company that develops manufacturing machines for the production of microchips, over the course of years build time has increased such that it sometimes affects the development speed of its control software. Recompilation after interface changes can run into hours, leading to a loss of productivity of the development team.

The obvious solution to this problem is to refactor interfaces such that the definitions in an interface are (1) functionally coherent (for maintainability), and (2) are used by a similar set of users (for build performance). Although these criteria correlate to some extent, they are not equivalent, resulting in trade-offs [1, 8].

In this paper, we investigate the use of automatic cluster analysis to address this problem. Hierarchical clustering aims at putting together entities (interface definitions) that are similar (in terms of their users) in the same cluster (interface), while entities in different clusters are less similar. Although this type of clustering has been applied to many software engineering problems, it has not been applied to interface redesign.

We are interested in the limits and opportunities of the use of cluster analysis for interface refactoring, and the practical difficulties one encounters in doing so. Therefore, we describe in this paper the application of cluster analysis to refactor the interfaces of a complex industrial embedded software system comprising millions of lines of C code. To apply cluster analysis to this code base, we have developed an interactive tool called Interface Regroup Wizard (IRW), which supports a software architect in refactoring the interfaces in C software components.

In this paper, we explore three questions:

1. Can an automatic approach such as cluster analysis be applied to improve the design of interfaces?
2. What are the limits of automation for the refactoring of

interfaces, that is, to what extent can the this be done automatically?

3. What needs to be done in practice before interfaces can be refactored automatically using cluster analysis?

The paper is structured as follows: In the next section, we offer a summary of related work in the area of cluster analysis and interface redesign. Next, in Section 3, we describe the clustering approach that we follow. Then, in Section 4, we introduce the industrial case that motivated our research. In Section 5 we describe the interactive tool we developed to apply cluster analysis to cases like these: the results of the actual case are covered in Section 6. We conclude the paper with a discussion of our findings, a summary of our contributions, and an outlook to future work.

2 Related Work

Cluster analysis has been applied to problems in various fields, such as psychiatry, medicine, and market research [6]. Also in software engineering a wide variety of problems have been solved using clustering techniques. Many of these revolve around the problem of remodularizing legacy systems [13, 16, 3, 4]. A unifying framework discussing various subsystem classification techniques is discussed by Lakhotia [9].

van Deursen and Kuipers [15] use agglomerative hierarchical clustering to restructure legacy software for the migration to object orientation. They furthermore explore how hierarchical clustering compares to formal concept analysis, an alternative grouping technique.

Mancoridis et al. [10] and Mitchell and Mancoridis [12] present a heuristic search algorithm for the recovery of modular software structure and a tool, Bunch, implementing the algorithm. They use traditional hill climbing and genetic techniques to optimize inter- and intra-cluster connectivity.

The primary contribution of the present paper is not so much in proposing new clustering algorithms: instead, we demonstrate through a concrete case study how cluster analysis can be effectively used to address the real life problem of interface redesign that companies (such as ASML) building large software systems are facing.

3 Cluster Analysis for Interface Redesign

We apply agglomerative hierarchical clustering [6]. This type of clustering starts with a strong clustering (i.e., a separate cluster for every entity) and then iteratively merges the two nearest clusters (based on some distance measure) until a desired clustering is obtained or the number of clusters is reduced to one. As such, we need to select a distance metric to calculate the distance between entities, a clustering algorithm to calculate the distance between clusters of entities, and quality metrics to select one of the generated clusterings.

3.1 Distance Metric

In our case, the entities to be clustered are the symbols (which are defined in interfaces). The feature set based on which we cluster is the set of using modules (.c-files using the symbol, which we also call "users"). Thus, each symbol is an entitiy, and each using .c-file is a feature.

As a distance metric we select the Jaccard distance. Maqbool and Babri [11] argue that Jaccard is particularly suited for asymmetric binary features and gives similar results as other distance metrics for cases where the feature vector is sparse (as it is in our case). We define the distance δ between two symbols s and t to be the Jaccard distance J_δ between their sets of users U_s and U_t:

$$\delta(s,t) = J_\delta(U_s,U_t) = \frac{|U_s \cup U_t| - |U_s \cap U_t|}{|U_s \cup U_t|}$$

As such, the distance between two symbols is between zero and one, where for similar symbols (in terms of their users) it is closer to zero (if U_s and U_t are disjoint the δ equals 1, if they are the same it is 0).

3.2 Clustering Algorithm

A clustering algorithm uses the distance between individual symbols to determine the distance between clusters. Table 1 gives an overview of several measures that are commonly used to calculate the distance δ between two clusters C and D.

The build performance depends on the total number of users for the symbols in an interface: any modification to an interface requires that all the .c-files that use a symbol in that interface need to be recompiled. Therefore, we select a cluster similarity measure that results in clusters in which the symbols are relatively similar. The extent to which the symbols in a cluster are similar is referred to as compactness. Of the measures enumerated in Table 1, complete, median, average, and single linkage perform from best to worst with respect to achieving compactness.

With single linkage two clusters might be clustered because two outliers are near, but all other symbols are not. Average performs better because it uses a cluster centre measure, but still considers outliers. Median performs even better because it does not include outliers for its centre measure. Complete performs best, because with this algorithm even the symbols furthest away are relatively near (similar).

Table 1. Commonly used clustering distance measures

single	$\delta_{si}(C,D) = \min_{s\in C, t\in D} \delta(s,t)$	minimum distance between each pair of symbols such that each pair has a symbol in both clusters.
complete	$\delta_{co}(C,D) = \max_{s\in C, t\in D} \delta(s,t)$	maximum distance between each pair of symbols such that each pair has a symbol in both clusters.
average	$\delta_{av}(C,D) = \frac{\sum_{s\in C}\sum_{t\in D}\delta(s,t)}{\lvert C\rvert\cdot\lvert D\rvert}$	average distance between each pair of symbols such that each pair has a symbol in both clusters.
median	$\delta_{me}(C,D) = med\{\delta(s,t) \mid s\in C, t\in D\}$	median distance between each pair of symbols such that each pair has a symbol in both clusters [5].

$$\delta_{ad}(C,D) = \begin{cases} \delta_{co}(C,D), & \text{if } 0 \le \delta_{co}(C,D) < 1 \\ \delta_{me}(C,D), & \text{if } \delta_{co}(C,D) = 1 \wedge 0 \le \delta_{me}(C,D) < 1 \\ \delta_{av}(C,D), & \text{if } \delta_{me}(C,D) = 1 \wedge 0 \le \delta_{av}(C,D) < 1 \\ \delta_{si}(C,D), & \text{if } \delta_{av}(C,D) = 1 \wedge 0 \le \delta_{si}(C,D) < 1 \end{cases}$$

Figure 1. Adaptive linkage

Adaptive Clustering Algorithm A problem with these cluster similarity measures is that at some point all clusters are at maximum distance (i.e., $J_\delta = 1$). From then onwards, cluster decisions are taken arbitrarily (the distance between clusters cannot decrease). This is particularly relevant for our setting, since we are mainly interested in clusterings with a relatively small number of clusters (as we will see in Section 4.2, architects would typically split the symbols in an existing interface in less than seven groups). Remember that we start with a large number of clusters that is reduced for each new clustering. Hence, we are interested in clusterings generated at the end of the clustering process. Obviously, we prefer a clustering algorithm that results in few arbitrary decisions. Of the enumerated measures above, single, average, median, and complete perform from best to worst with respect to the number of arbitrary decisions (which we want to minimize).

From the previous discussion, we can conclude that for selecting a cluster similarity measure there is a trade-off between the compactness of the generated clusterings and the number of arbitrary decisions taken. Therefore, we propose a hybrid approach that combines these measures, which we call *adaptive linkage*. It uses the best measure (with respect to compactness) that does not result in arbitrary decisions. For two clusters C and D it is defined as in Figure 1.

3.3 Selection of a Clustering

The iterative process of hierarchical clustering creates a new clustering after each iteration that contains less clusters than the previous clustering. This type of clustering process is particularly suited when the optimal number of clusters is unknown beforehand. However, it does require a means to select a suitable clustering. In our approach we leave this decision to the software architect, the user of our approach. We aid the architect by presenting him a graph that plots each generated clustering against two validation metrics proposed by Handl and Knowles [7] to measure compactness and connectivity. A good clustering has minimal values for both measures.

Compactness As a measure for the compactness of a clustering CC we use intra-cluster variance. Intra-cluster variance is defined as the root mean square distance between entities and their cluster's centre:

$$Comp(CC) = \sqrt{\frac{1}{N}\sum_{C\in CC}\sum_{s\in C}\delta(s,\mu_C)^2},$$

where μ_C is the centre of cluster C and N is the total number of symbols.

Connectivity Connectivity is a measure for the degree to which near neighbouring symbols have been placed in the same cluster. It penalizes the situation that near neighbours are not in the same cluster; the nearer the neighbour, the higher the penalty.

$$Conn(CC) = \sum_{s\in CC}\sum_{j=1}^{L} p_{s,nn_{s(j)}},$$

where

$$p_{s,nn_{s(j)}} = \begin{cases} \frac{1}{j}, & \text{if } s\in C \wedge nn_{s(j)} \notin C \\ 0, & \text{otherwise} \end{cases}$$

Here, L is the number of near neighbours we want to take into account, and $nn_{s(j)}$ is the j^{th} nearest neighbour of symbol s.

4 Interface Redesign in Practice

In this section we discuss what needs to be done before the algorithms discussed in the previous section can be applied. This particularly relates to the question how to take into account various criteria for putting interface definitions together that are not directly related to dependencies. All this is based on our experience in applying these techniques to the interfaces of a large industrial system, which we introduce first.

4.1 Wafer Scanner

The context in which we developed our approach consists of the embedded control software of a wafer scanner developed by ASML, the leading manufacturer of lithography systems for the semiconductor industry. In a wafer scanner silicon wafers are exposed to circuit patterns in a lithographic process. The result is a wafer that holds hundreds of microchips. The process performed by a wafer scanner is a critical step in the manufacturing of microchips.

The main drivers for the development of these machines are reduction of line thickness, overlay precision, and productivity. Extreme requirements for line thickness and overlay (nanometer scale) result into complex optical, mechatronic and metrology solutions, which, in turn, result into large (approx. 20 MLOC) and complex control software. As such, many interfaces have been defined, of which some contain several thousand definitions.

During the last eight years ASML's high-end scanners have been advancing from 150 nm resolution and 80 wafers per hour to 45 nm resolution and 130 wafers per hour. This improvement has been realized by newly designed optical, mechatronic and metrology system parts. It is nearly impossible to prepare software interfaces for such changes in the system. Practice shows that some software interfaces need to be changed to support more advanced systems.

Software architecture For refactoring the interfaces of the wafer scanner control software, build performance is not the only criterion. Maintainability is another important criterion. Furthermore, the logical structure of the source code is more intricate than source files and header files; from an architectural perspective the system is decomposed into many components on different hierarchical levels. Our approach needs to take this structure into account as well.

The information we use for the cluster analysis is modelled in the architecture style depicted in Figure 2. Here, a System is composed of one or more logical components. In turn, each Component may be decomposed into subcomponents, resulting in a hierarchy of components. In our case, a 'leaf' Component consists of a number of C source code files (.c-files) and interfaces. An Interface is composed of a number of header files (.h-files) that each define a set of symbols. A Symbol might be a Macro definition, Function declaration, or Type definition. A .c-file may use any number of symbols. Such a .c-file is referred to as the user of that Symbol. An Interface can be provided by a Component, meaning that the Interface is visible to other components on the same hierarchical level. As such, we can assign a level to an Interface that corresponds to the highest hierarchical level at which it is visible. Finally, it is also possible that an Interface shares other Interfaces, that is, it includes other interfaces of which the symbols will be visible on the hierarchical level of the sharing Interface.

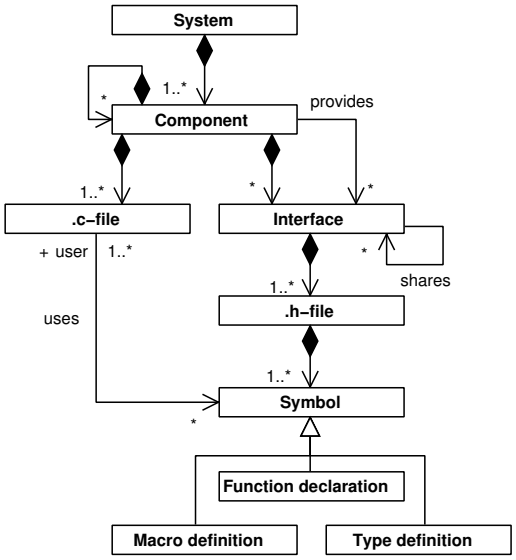

Figure 2. Architecture style

In the case of ASML, the software is structured into hundreds of components on four hierarchical levels, being, from fine grained to coarse grained: component, building block, functional cluster, and system. These components provide more than 1500 interfaces that define over half a million different symbols. Analysis of the source code revealed that in total these symbols are used 1.6 million times. As such, each symbol, on average, is used by approximately three users (i.e., .c-files). Note that we do not take into account the possibility that a symbol can be used multiple times in a single .c-file (such usage is simply counted as one).

4.2 Interface Quality

To minimize build time upon recompilation, one might think that is optimal to define a seperate interface for each symbol. However, this significantly slows down the overall compilation process, since for every symol a separate (.h) file must be opened. Furtermore, overall, the optimal interface design is a trade-off between build time and maintainability. Interfaces are used to reason about the system's structure and behaviour. This requires a manageable number of them. Therefore, defining a separate interface for each symbol has a negative impact on the maintainability of the system. Hence, symbols that are related in terms of functionality are grouped in manageable interfaces. The overall criteria ASML uses to assess the quality of its interfaces are mentioned in Table 2.

The *dependency* criterion is clearly meant to reduce build time. To some extent this criterion also improves maintain-

Table 2. Interface quality criteria

Dependency	The number of .c-files that needs to be rebuilt into object files that, in turn, need to be linked, due to a modified interface has to be minimized.
Sharing	The number of interfaces that is shared by other interfaces should be minimized.
Encapsulation	We measure encapsulation as the number of symbols actually used by users on the interface's visibility level relative to the total number of symbols it defines. Encapsulation should be maximized.
Functional coherence	The symbols defined in an interface should be related to the same functionality.
Stability	Interfaces should be stable; the number of changes should be minimized.

ability as we would expect that two symbols with a similar set of users are related in terms of functionality.

Sharing also affects both maintainability and build time. Sharing makes it possible that .h-files are indirectly included by other .h-files, which reduces understandability and hence maintainability. It also affects build time. When a symbol of an interface is changed that is shared by other interfaces all users of the shared interface and the sharing interfaces (recursively) have to be rebuilt.

The level of *encapsulation* is only related to maintainability. A general design principle is to limit the visibility of elements to the scope in which they are used.

Functional coherence is important for maintainability. Although it is related to dependency, no objective measures exists for this criterion. It is, however, the criterion architects predominantly use for the design of interfaces.

The final criterion, *stability*, is related to the rate at which changes are made to interfaces. This rate should be as low as possible. This criterion is related to what symbols are defined by the interfaces, and not by which interface defines which symbol.

For these and some other criteria that can be measured, ASML specified desired values. For dependency and sharing these are specified relative to a system-wide average. For encapsulation, absolute values are specified. Based on the extent to which these measures for a particular interface deviate from the desired values, each interface can be assigned a compliancy score. All interfaces with a compliancy score above a certain threshold need to be redesigned.

Currently, ASML uses a step-by-step, manual approach to optimize interfaces with respect to these criteria. This guideline is based on two principles: 1) grouping functional coherent symbols in an interface, and 2) preventing that symbols used at different architectural levels end up in the same interface. Additionally, the number of resulting interfaces needs to be limited. In this paper, we explore how cluster analysis can be applied to automate this process.

5 Interface Regroup Wizard (IRW)

In order to be able to apply clustering techniques as described in Section 3 to the setting described in the previous section, we have developed a tool, called the *Interface Regroup Wizard (IRW)*. It aims at supporting software architects in the redesign of the interfaces of components in their software system. The IRW consists of two parts: a parser and a clustering application.

5.1 Parsing

Because parsing the source code of a system the size of our case study is expensive, and to allow efficient use of the IRW, the parsed interface data is persisted in a database. As such, for a particular software system, parsing is a one-time operation.

An existing third party source code analyzer is used to extract the necessary information from ASML's code base. The results of this analysis are used to populate a relational database, which contains tables for symbols, uses relationships, users, symbols, and so forth.

Furtermore, the database is filled with information on the architectural model containing the system decomposition in terms of components, their subcomponents, and the interfaces they provide and require. The schema of this decomposition is based on Figure 2. More details concerning the design and implementation of the tool can be found in Adnan [2].

5.2 Clustering

By storing the interface data in a database, we can subsequently efficiently use the clustering application to investigate clusterings of the interface of choice. Using the IRW, one or more interfaces can be selected. Then, for the set of symbols defined in those interfaces clusterings can be generated. The architect selects one of the generated clusterings using the variance-connectivity graph, and by inspecting clustering candidates.

The main screen of the clustering application is depicted in Figure 3. It consists of a number of toolbars and two lists that show the interfaces of a particular component, and the symbols of a particular interface respectively.

The top toolbar offers buttons to start the regrouping process, to view the users of a selected symbol and to refresh the two lists when changes to some configuration parameters are made. The horizontal toolbar below that consists of controls for finding a particular interface to be regrouped. On the left toolbar are controls to configure various parameters for the clustering process: the clustering method, granularity of users, local boundary, whether or not to merge

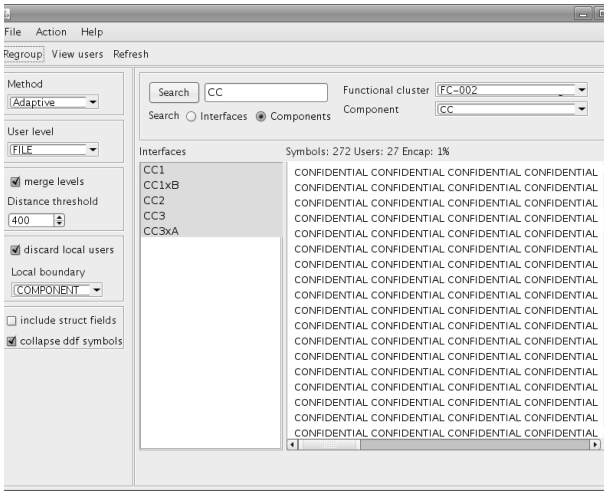

Figure 3. Main screen

clusters at different hierarchical levels, the maximum distance at which this will be considered, and whether or not to include struct members, to consider local users, or to collapse symbols generated from a single source symbol.

Once one or more interfaces are selected and configuration parameters have been set, the user can start the clustering process by pressing the regroup button. Depending on the size of the selected interface(s) this may take up to a minute.

During the regrouping process the user may be confronted with questions regarding the merging of clusters at different hierarchical levels. In principle the tool does not allow such merging. However, when the distance between two clusters at different visibility levels is below the threshold that was set, the user can decide to merge them after all. If there are too many questions the user can decide to cancel the process and lower the distance threshold or prevent clustering of such clusters all together.

5.3 Selecting a Clustering

When the clustering process is completed, one of the generated clusterings needs to be selected. To aid the architect with this, the IRW presents a graph that conveys information about the quality of the generated clusterings. Figure 4 displays how the graph is presented to the user of the IRW. The graph plots each generated clustering against its variance (vertically) and connectivity (horizontally) values.

Agglomerative hierarchical clustering starts with a separate cluster for each entity. Such a clustering has low variance and high connectivity and will be plotted in the bottom-right of the graph. Each successive clustering has fewer clusters, a lower connectivity (the clusters of neighbouring entities that gave rise to a penalty might have merged), and a higher variance. This trend can easily be observed from the graph.

The architect can use this graph to select a suitable clustering. By hovering over the plotted clusterings, a tooltip appears that indicates how many clusters that clustering contains. Assuming that the architect has an idea of the upper and lower bounds for the desired number of interfaces in the new design, an area of the graph can be selected in order to zoom in for closer inspection of the clusterings within those bounds.

Handl and Knowles [7] propose to consider the ratio $\frac{\Delta\sigma^2}{\Delta Conn}$ for two successive clusterings. The idea is to search for clusterings for which this ratio is maximal. More informally, this selection criterion identifies a clustering for which an extra agglomeration of two clusters results in a large degradation of compactness (i.e., measured by variance), while connectivity is not improved much. Visually such a clustering can be recognized by a 'knee' in the validation graph. That clustering might be a good candidate (e.g., the clustering indicated with the arrow in Figure 4).

Another indication of the quality of a clustering are the values in the list in Figure 4 that indicate the minimum distance between any two clusters for a particular clustering. Note that each line further indicates the clustering algorithm by which the most recent clustering decision was made (this is especially useful for our adaptive algorithm; C: complete, M: median, A: average, S: single). For instance, the distance between the two closest clusters in the selected clustering is 985 (i.e., 1.5% of the users of the two most distant symbols of these two clusters are identical[1]).

Candidate clusterings can be further inspected. The IRW can show the contents of the clusters for a clustering. Additionally, it displays:

- the hierarchical level at which the interface (and thus all symbols it defines) are visible;
- a measure for the encapsulation of the interface $e = \frac{|S_l|}{|S_t|} \cdot 100\%$, where S_l is the set of symbols used by users at the same hierarchical level as the interface, and s_t is the set of symbols the interface defines;
- the number of symbols used at each hierarchical level; and
- a measure for the dependency that indicates the number of users that have to be rebuilt in the case one of the symbols in the interface (cluster) changes.

The tool also displays average values for encapsulation and dependency for the complete clustering. Using these measures the architect can select a suitable clustering.

[1] Note that the IRW multiplies the Jaccard distance defined in Section 3.1 by 1000.

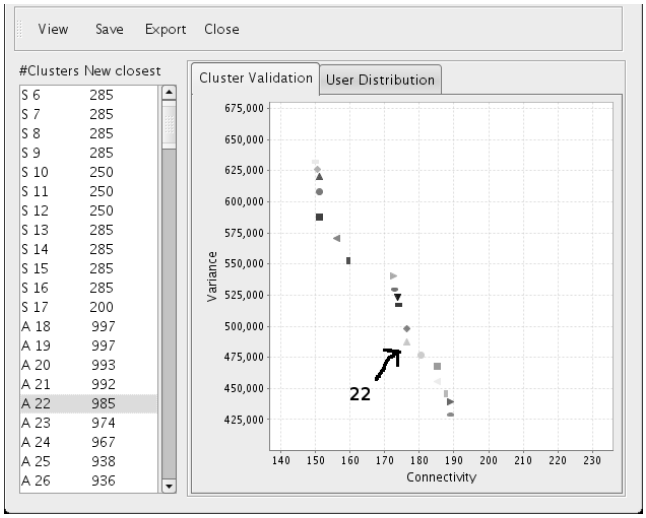

Figure 4. Selecting a clustering

5.4 Managing Interface Quality Criteria

The cluster similarity measures discussed above result in clusterings that are optimized with respect to compactness and connectivity. Considering the criteria that ASML uses to evaluate interfaces discussed in Section 4.2, this only relates to dependency. The IRW takes into account the other criteria (except stability) as well.

To take encapsulation into account, the IRW in principle does not merge clusters of which the symbols are used at different architectural levels. Only when the distance between two clusters is smaller than a configurable threshold (i.e., the number of users shared by the two clusters is above a certain threshold) the architect is notified with the percentage of shared users and given the opportunity to merge them. Although this negatively impacts encapsulation (by not allowing such merging, encapsulation will always be 100%), in some cases it might be justifiable to sacrifice encapsulation for some other criterion.

Sharing is addressed similarly by a simple extension of the architectural levels. For each defined architectural level (that is, in the case of ASML, 'component', 'building block', 'functional cluster'), we add an extra 'shared' level. To clusters that contain symbols that are used by .h-files (i.e., are shared) we assign these 'shared' levels. For instance, a cluster with shared symbols visible at the component level, is assigned the level 'component-shared', instead of 'component'. Because, as explained above, the IRW prevents as much as possible the merging of clusters with different levels, the number of symbols in a shared cluster (interface) is reduced.

Functional coherence is taken into account to the extent that an architect can decide whether or not to merge clusters, for which domain knowledge might be used. Similarly, the architect needs to select a suitable clustering from all generated clusterings.

6 Case Study

To assess the usefulness of the IRW and to further improve it, we applied IRW to a series of actual ASML interfaces. We asked ASML's high-level architects to propose interfaces in need of a redesign or ones that have recently been redesigned manually. In total we applied the IRW to twenty interfaces of which four had already been redesigned. As such, for the latter, we could compare the results with manually redesigned interfaces. To this end, we considered to MoJo distance [14], that is, the number of symbol moves and cluster joins required to transform IRW's proposal into the manual redesign.

For the remaining interfaces we considered the measurable criteria for interface quality defined by ASML (Section 4.2).

On a 1.73 GHz, 1024 MB notebook, parsing of the two input files and population of the database took several hours. With the database filled, the wizard was used interactively to refactor the interfaces suggested by ASML's architects.

Since we do not have room to discuss all these refactorings we only discuss one refactoring that we evaluated with respect to ASML's measurable criteria for interface quality, and one refactoring for an interface for which a manual refactoring was already made. The results for the other 18 interfaces we refactored are similar to those discussed below.

Table 3. Generated clustering 1

id	#sym.	enc.	#dep.
1	43	100%	156
2	30	100%	32
3	20	95%	49
4	16	100%	3
5	9	100%	11
6	8	100%	8
7	7	100%	385
8	5	100%	32

6.1 Assessment Based on Interface Quality Criteria

The interface of the first refactoring we discuss defines 545 symbols, which is significantly larger than the average size of an interface at ASML. After collapsing generated symbols, and enumerations, 165 symbols remain for clustering.

The dependency measure for this interface is 483, that is, 483 files need to be rebuilt for each single change to this interface. The encapsulation measure is 10%, that is, 90% of the symbols this interface defines are visible on higher hierarchical levels than they are used. This value for the encapsulation measure is much lower than desired (80%). Finally, this interface also shares more other interfaces than desired. As such, this interface is one of the interfaces that needs to be redesigned according to ASML's compliancy criteria.

We used the IRW to find a clustering of the symbols in this interface, such that all resulting clusters comply to ASML's interface criteria. The generation of clusterings for the interface we selected takes approximately half a minute of computation time on the same notebook as before. Using the heuristic explained earlier, we select the clustering consisting of 22 clusters from the validation graph in Figure 4. The eight largest clusters are depicted in Table 3. These eight clusters contain 138 of the 165 interface symbols.

Each of the generated clusters complies to ASML's interface criteria. Only one clusters has an encapsulation value of less than 100%. One cluster remains with a relatively high dependency value, but all other clusters have much lower dependency values.

Figure 5 displays a pie chart that shows which part of the users (of all symbols in the clustering), uses symbols in a certain number of clusters. This chart can be used to illustrate the quality of a clustering. Investigation of this chart (by hovering the mouse pointer over the different slices), reveals, for instance, that the maximum number of clusters used by any user is eight and that only 7% of the users use symbols in more than four clusters.

The architects at ASML want to split an interface in only two to seven pieces. The IRW, however, proposes many more clusters than that. Looking closer at Table 3 it can be seen that the clustering consists of only a few large clusters and many small clusters. These should be merged manually with the target clusters by the architect using domain knowledge.

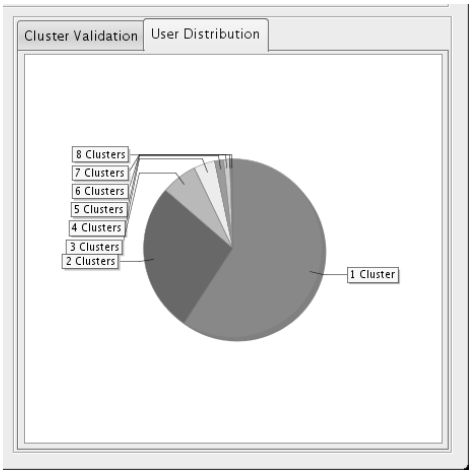

Figure 5. User distribution

Table 4. CC interfaces

interface	#sym.
CC1	326
CC1xB	12
CC2	4
CC3	90
CC3xA	17

6.2 Comparison with Manual Refactoring

The interface of the component named CC was already refactored manually. To compare the proposals of the IRW with those of the architects, we analyse the differences in terms of the operations (symbol moves, clusters merges) required to transform IRW's proposal into that of the architect.

The original CC interfaces define 449 symbols that are used by 128 different .c-files. After collapsing generated symbols and enums, 272 symbols remain. When viewed as a single interface the encapsulation value is 1%. The ASML architects decided to split this interface into five interfaces. The number of symbols in each of these interfaces is depicted in Table 4. This allows us to use this manual refactoring to evalute the result of the IRW.

Using the validation graph and information on the distance of between the two closest cluster in a clustering, we select one of the generated clusterings. This clustering contains 28 clusters of which the clusters with two or more symbols are listed in Table 5. The table shows for each cluster the number of symbols it contains, the encapsulation value, the number of users for all its symbols, which of the

Table 5. Generated clustering 2

id	*#sym.*	*enc.*	*#dep.*	*interface*	#errors
1	25	92%	71	CC1	2
2	23	65%	39	CC1	0
3	22	100%	5	CC1	0
4	20	100%	23	CC3	0
5	17	100%	8	CC1	0
6	16	31%	4	CC3xA	0
7	15	100%	34	CC1	1
8	15	100%	22	CC1	2
9	15	100%	3	CC3	0
10	13	100%	2	CC1	0
11	12	100%	2	CC1	0
12	9	100%	3	CC1	0
13	8	100%	8	CC1	1
14	8	100%	1	CC1	1
15	8	100%	16	CC3	0
16	8	100%	4	CC3	0
17	7	100%	2	CC1	2
18	6	100%	3	CC1	0
19	6	100%	9	CC1xB	0
20	5	100%	100	CC1	0
21	3	100%	1	CC1	0
22	3	100%	1	CC1xB	0
23	2	100%	18	CC1	0
24	2	100%	1	CC2	0

manually created interfaces its symbols correspond to, and the number of misplaced symbols. The latter is the count of the symbols that had been placed in a different interface by the architect than the majority of the cluster's symbols. In total 9 out of 272 symbols were misplaced.

The difference between IRW's refactoring proposal for the CC interfaces and the refactoring done manually is exactly 23 cluster merges and 9 symbol moves. As such, the number of decissions required to refine the automatically generated refactoring is significantly less than is required for the complete manual refactoring of an interface consisting of 272 symbols.

7 Discussion

Overall, the architects at ASML concluded that the interface refactoring proposals generated by the IRW are a valuable starting point for manual refinement and that effort is saved by application of the tool; the required refinements require less effort than a complete manual refactoring.

When we evaluate the IRW-proposals with respect to the number of symbols that end up in a wrong cluster, we conclude that this number increases considerably for clusterings with only a few clusters. The reason for this is that the decisions to merge two clusters become less obvious during the clustering process. For the interfaces of the CC component discussed in Section 6, for instance, the percentage of identical users of the two clusters closest together dropped below 15% as soon as the remaining number of clusters was 17. Our other experiments showed similar results.

As such, we envision the IRW to be used to obtain a refactoring proposal consisting of a set of clusters that is larger than desired. The architects at ASML typically aim at refactoring an interface into 2-7 pieces It is up to the architect to finish the restructuring using domain knowledge. By taking a clustering consisting of more than the desired number of clusters, the final work mainly consists of merging instead of moving symbols around, which requires more effort and is error-prone. To summarize, from our experiments we can conclude that automating the restructuring makes sense as long as the decisions made by the tool are obvious, in the sense that there is a considerable amount of overlap between the users of the two clusters to be merged.

To aid the architect with the selection of a generated clustering that mainly requires cluster merges instead of symbol moves to finish, the distance between the two closest clusters in the current clustering is particularly useful information (see Figure 4). It gives an indication of the meaningfulness of the next step in the clustering process. And as the decisions with the type of clustering implemented by the IRW will never be reversed later on (i.e., this value only increases), it gives a clue on what clustering to select.

Important to note here is that for our experiments the user of the IRW had very little domain knowledge. This user had to make a number of manual decisions during the refactoring process. For instance, the tool requires a confirmation for merging two clusters with different visibility levels (see Section 5.2). Also the selection of an appropriate clustering requires manual input. In our experiments these decisions were primarily based on the measures and information the tool provides. As such, we expect that even better results are obtained when the tool is used by domain experts, which will generally be the case.

A final issue is that sometimes the IRW places definitions that should be together from a functional perspective in different clusters. A possible solution would be to make to tool more interactive to allow architects to indicate that such symbols should be kept together. Although we leave this is future work, a potential way to solve this is to replace such pairs (or sets) with a single symbol during clustering, like we do with generated symbols. In the final proposal such symbols would have to be expanded again.

8 Conclusion

In this paper we investigated the applicability of automated techniques for the redesign of software interfaces. To this end we developed a tool, the Interface Regroup Wizard (IRW), which we applied to several interfaces of the embedded control software of a complex manufacturing system.

We consider the following as our key contributions:

- A demonstration that automated cluster analysis can be successfully to the redesign of C interfaces, a realistic industrial task actually conducted by ASML.

- An analysis concerning the level of automation that can be achieved, and guidelines indicating when the automation should stop, and when the software architect should take over the clustering process.
- An operationalization of quality criteria guiding interface redesign.
- An interactive tool, IRW, which supports the software architect in the redesign of a software system's interfaces.

From our experiments we conclude that our tool provides a useful starting point for the refactoring of software interfaces as described in this paper. However, one of the lessons learned is that this type of clustering can only partly automate the solution to this problem. In this case, this means that the tool gives a starting point (typically containing too many clusters) that architects can use to further refine (i.e. reduce the number of clusters based on different criteria than common users).

Naturally, the architect is likely to use domain knowledge for selecting one of generated clusterings generated by the IRW for use as a starting point for refactoring an interface. In addition to that, means to objectively assess the quality of the generated clusterings exist. We provide such means by displaying different metrics for compactness and connectivity, as well as an indication of how obvious the next clustering decision is (by showing the distance between the current closest clusters).

The IRW helps to directly improve three of the five interface quality criteria defined by ASML: dependency, sharing, encapsulation. Furthermore, when used by a domain expert, the interactive nature of our tool (to a lesser extent) also addresses functional coherence.

As part of our future work, ASML has expressed interest in conducting further experiments with the tool, which will certainly lead to further improvements and refinements to the method and the tool. In addition to that, we are interested in applying the approach to other (non-ASML) systems, for example from the open source domain.

Acknowledgements Part of the research described in this paper was sponsored by NWO via the Jacquard Reconstructor project. We thank the functional cluster architects at ASML for their kind cooperation, as well as Remco van Engelen for his support in initiating this research direction.

References

[1] R. Adams, W. Tichy, and A. Weinert. The cost of selective recompilation and environment processing. *ACM Trans. Softw. Eng. Methodol.*, 3(1):3–28, 1994.

[2] R. Adnan. Interface Regroup Wizard. Master's thesis, Software Engineering Research Group, Delft University of Technology, October 2007.

[3] B. Andreopoulos, A. An, V. Tzerpos, and X. Wang. Clustering large software systems at multiple layers. *Inf. Softw. Technol.*, 49(3):244–254, 2007.

[4] A. Christl, R. Koschke, and M.-A. Storey. Automated clustering to support the reflexion method. *Inf. Softw. Technol.*, 49(3):255–274, 2007.

[5] Roy G. D'Andrade. U-statistic hierarchical clustering. *Psychometrika*, 43(1):59–67, March 1978.

[6] Brian S. Everitt. *Cluster Analysis*. Edward Arnold, 3rd edition, 1993.

[7] J. Handl and J. Knowles. Exploiting the trade-off – the benefits of multiple objectives in data clustering. In *Proc. 3rd Int. Conf. on Evolutionary Multi-Criterion Optimization (EMO'05)*, volume 3410 of *LNCS*, pages 547–560. Springer-Verlag, 2005.

[8] M. de Jonge. Build-level components. *IEEE Trans. Software Eng.*, 31(7):588–600, 2005.

[9] Arun Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, 36(3):211–231, 1997.

[10] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings ICPC*, pages 45–54. IEEE Computer Society, 1998.

[11] O. Maqbool and H. A. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, November 2007.

[12] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[13] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 83–92. IEEE, 1991.

[14] V. Tzerpos and R. C. Holt. MoJo: A distance metric for software clusterings. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 187. IEEE Computer Society, 1999.

[15] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 246–255. IEEE Computer Society, 1999.

[16] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 33. IEEE Computer Society, 1997.

TUD-SERG-2008-026
ISSN 1872-5392