# Fitness Workout for Fat Interfaces: Be Slim, Clean, and Flexible

Spyros Kranas*, Apostolos V. Zarras* and Panos Vassiliadis*
*Dept. of Computer Science and Engineering, Univ. Ioannina, Greece
Email: {skranas, zarras, pvassil}@cs.uoi.gr

***Abstract*—A class that provides a *fat interface* violates the interface segregation principle, which states that the clients of the class should not be coupled with methods that they do not need. Coping with this problem involves extracting interfaces that satisfy the needs of the clients. In this paper, we envision an *interface extraction method* that serves a combination of four principles: (1) *fitness*, as the extracted interfaces have to fit the needs of the clients, (2) *clarity*, as the interfaces should not be cluttered with duplicated methods declarations due to the clients' similar needs, (3) *flexibility*, as it should be easy to maintain the extracted interfaces to cope with client changes, without affecting parts of the software that are not concerned by the changes, and (4) *practicality*, as the interface extraction should account for practical issues like the number of extracted interfaces, domain/developer specific constraints on what to include in the interfaces, etc. Our preliminary results show that it is feasible to extract interfaces by respecting the aforementioned principles. Moreover, our results reveal a number of open issues around the trading between fitness, clarity, flexibility and practicality.**

***Index Terms*—Refactoring, interface segregation, extract interface**

## I. Introduction

**Fat interfaces & interface segregation.** In the Object-Oriented paradigm, the *interface segregation principle* states that the clients of a class should not be forced to depend upon methods that they do not use [1]. Empirical studies highlighted the practical benefits of the interface segregation principle [2] and empirical evidence showed that its violation leads to time consuming, costly, and error-prone maintenance [3]. A class provides a *fat interface* if its clients use subsets of the class' instance methods [4]. Classes with fat interfaces violate the interface segregation principle.

**Problem.** Dealing with classes that provide fat interfaces amounts to employing the *Extract Interface* refactoring [5]. The rationale behind this refactoring is to define a set of interfaces that satisfy the needs of the clients' of a given class. Although this seems a simple idea, in practice interface extraction is not an easy problem because it involves a number of conflicting concerns that the developer must take into account. More specifically:

- We have to extract interfaces that *fit* the needs of the clients (i.e., they provide the methods invoked by the clients).
- As the clients' needs are typically similar, the extracted interfaces become cluttered with duplicated method declarations. Hence, we must maintain the conceptual *clarity* of the extracted interfaces.
- Software evolves over time; new clients may be added, clients may be removed, the methods invoked by the clients may change. We must extract *flexible* interfaces that can be seamlessly maintained to cope with such changes, without affecting parts of the software that are not concerned by the changes.
- We must consider further *practical issues*, like the number of interfaces that are extracted, which should be reasonable, domain or developer specific concerns that impose constraints on what to include (resp. exclude) in (resp. from) a particular interface, etc.

**State of the art:** In well known IDEs, like Eclipse[1], Netbeans[2], and IntelliJ IDEA[3], a developer extracts interfaces for a particular class by *manually* selecting the methods to be included in the extracted interfaces. Then, the IDEs provide automated means that modify the clients of the class towards using the extracted interfaces. The method proposed in [6] starts from a class that provides a fat interface and automatically extracts interfaces that fit the needs of the clients. However, the clarity of the interfaces is not considered. Specifically, if the clients use similar subsets of methods, the extracted interfaces are overlapping. In [7] and [8], the extraction of interfaces is handled as a search-based problem. These approaches start from a class that provides a fat interface and extract a set of interfaces based on the way that the fat interface is used by the clients. The extraction process tries to maximize the degree to which the interfaces fit the needs of the clients, while minimizing the number of extracted interfaces; [7] employs agglomerative clustering, while [8] relies on genetic algorithm. Hence, [7] and [8] do not guarantee that the extracted interfaces would fit the needs of the clients. The other problem with [7] and [8] is rigidity[4], in the sense that the extracted interfaces cannot be incrementally updated to cope with client additions, removals, or updates. On the contrary, to deal with such changes, the interface extraction should start all over again with different input (i.e., the changed clients) and output (i.e., the extracted interfaces). Using an entirely new set of interfaces produced by the interface extraction process, requires changing all the clients of the class.

[1] eclipse.org/
[2] netbeans.org
[3] jetbrains.com/idea/
[4] In general, we say that a software system is rigid if it is hard to change because every change affects too many parts of the system [1].

**Contribution.** We believe that the interface extraction problem should be handled with a fresh and broader look that considers the fitness, clarity, and flexibility of the extracted interfaces, along with, practical issues like the number of extracted interfaces, domain/developer specific constraints and so on. In this paper, we propose an interface extraction method, which starts from a class that provides a fat interface and produces a hierarchy of interfaces that has the following properties:

- *Fitness*: For each client there is an interface that provides exactly the methods the client needs.
- *Clarity*: The extracted interfaces are exempt from duplicated methods declarations due to the clients' similar needs. At the same time, the interfaces highlight the similarity between the clients' needs via generalization relations, which form the hierarchical structure of the extracted interfaces.
- *Flexibility*: The extracted hierarchy of interfaces can be incrementally updated to cope with client changes, without affecting parts that are not concerned by the changes.
- *Practicality*: The number of extracted interfaces is reasonable - typically quite lower than the number of clients.

To assess the effectiveness and the practicality of the proposed method we report the preliminary results of an empirical study, performed over six open-source projects. Our results show that it is feasible to extract fit, clean, and flexible hierarchies that consist of a resonable number of interfaces, with respect to the number of clients a class has. Moreover, our results reveal a number of open issues when trading between fitness, clarity, flexibility and practicality.

## II. Approach

In this section, we briefly sketch the main concepts of the proposed method.

**Fitness.** For a given class $c$ that provides a fat interface and the set of clients $\mathcal{C}$ that use $c$, the proposed method extracts a flexible hierarchy of interfaces $\mathcal{I}$. Initially, the method iterates over the set of the clients $\mathcal{C}$ that use $c$. For each client, it constructs an interface that declares exactly the methods used by the client.

To illustrate the interface extraction method we employ a real world example that involves the `RunNotifier` class[5] of the JUnit testing framework. `RunNotifier` is a key element of JUnit's notification mechanism that allows listeners to receive feedback about the progress of tests. `RunNotifier` has a fat interface that declares 11 methods. The class has 12 clients $c_1, c_2, \ldots c_{12}$ that use different subsets of the class' methods. Consequently, the initialization phase of the extraction method produces 12 interfaces $i_1, i_2, \ldots i_{12}$ that fit the needs of $c_1, c_2, \ldots c_{12}$, respectively. The initial interfaces are refined in the next phases of the proposed approach as discussed below. For simplicity reasons in our example, we focus on a subset of the extracted interfaces that is given in Figure 1(a).

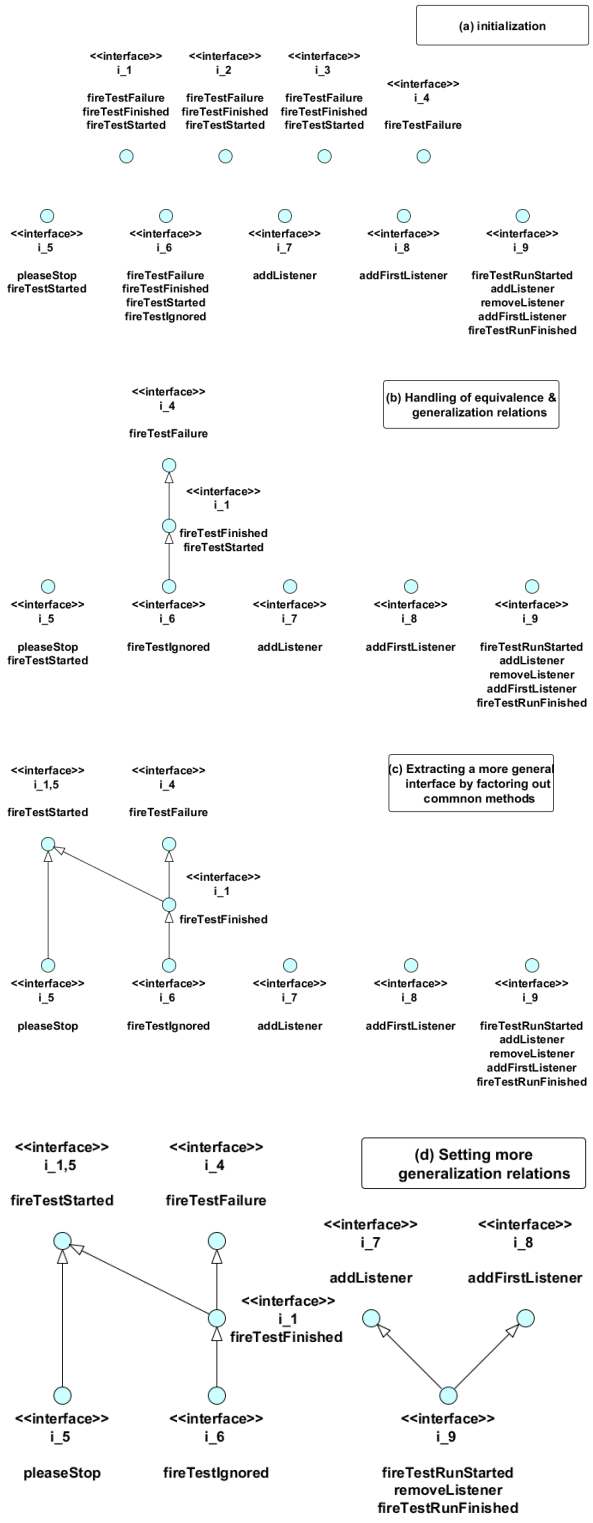


Fig. 1. Illustrating example: Hierarchy of interfaces for the `RunNotifier` class.

**Clarity.** The initial distribution of methods to interfaces

[5] junit.sourceforge.net/javadoc/org/junit/runner/notification/RunNotifier.html

ensures that the extracted interfaces satisfy the needs of the clients. Nevertheless, if clients have similar needs, the extracted interfaces may be overlapping, or even equivalent. The proposed method proceeds with an iterative process that improves the clarity of the interfaces. Each iteration, looks for the most similar pair of interfaces, $i_1^{max}$, $i_2^{max}$. To measure interface similarity, we employ the Jaccard similarity coefficient [9], i.e., the similarity between $i_1^{max}$, $i_2^{max}$, is the cardinality of the intersection of the methods declared in the interfaces, divided by the cardinality of the union of the methods declared in the interfaces. Then, clarity is improved as follows:

- *Dealing with equivalence between $i_1^{max}$, $i_2^{max}$:* If $i_1^{max}$ and $i_2^{max}$ provide the same methods[6], only one of the two interfaces is retained.
- *Setting a generalization relation between $i_1^{max}$, $i_2^{max}$:* If the set of methods provided[7] by $i_1^{max}$ is a subset of the methods provided by $i_2^{max}$, the method declarations of $i_1^{max}$ are removed from $i_2^{max}$ and $i_2^{max}$ becomes an extension of $i_1^{max}$. The case where $i_2^{max}$ is more general than $i_1^{max}$ is treated in the same way.
- *Extracting of a more general interface for $i_1^{max}$, $i_2^{max}$:* If there is no equivalence or generalization relation between $i_1^{max}$ and $i_2^{max}$, a more general interface, $i_{1,2}^{max}$, is extracted. In particular, $i_{1,2}^{max}$ declares the methods that are common in $i_1^{max}$ and $i_2^{max}$. Following, generalization relations are set between $i_{1,2}^{max}$, $i_1^{max}$ and $i_{1,2}^{max}$, $i_2^{max}$. Specifically, the common method declarations are removed from $i_1^{max}$ and $i_2^{max}$; $i_1^{max}$ and $i_2^{max}$ become extensions of $i_{1,2}^{max}$.

In our example, the first two iterations (Figure 1(b)) discard $i_2$ and $i_3$ because they are equivalent with $i_1$. In the third iteration the most similar pair of interfaces is $i_1$, $i_6$. The methods provided by $i_1$ are a subset of the methods provided by $i_6$. Hence, a generalization relation is detected and $i_6$ becomes an extension of $i_1$. Similarly, in the fourth iteration, a generalization relation is set between $i_1$ and $i_4$. Figure 1(c) gives the results of the fifth iteration. The most similar pair of interfaces is $i_1$, $i_5$. None of these interfaces is a generalization of the other. Therefore, a more general interface $i_{1,5}$, is extracted. The interface contains the `fireTestStarted` method that is common in $i_1$ and $i_5$; $i_1$ and $i_5$, become extensions of $i_{1,5}$. Finally, Figure 1(d) depicts two more generalization relations, set in the last two iterations; the first one between $i_9$ and $i_7$, and the second one between $i_9$ and $i_8$.

**Flexibility.** Fitness and clarity, enable flexibility, i.e., dealing with changes to clients without affecting parts of the software that are not concerned by the changes.

More specifically, given the hierarchy of interfaces $\mathcal{I}$ that is extracted for a class $c$, there are 3 main evolution scenarios concerning the clients of $c$: addition of new clients, deletion of clients, and update of the clients' needs. Let $\mathcal{A}$, $\mathcal{D}$ and $\mathcal{U}$ denote the sets of clients that are added, deleted, and updated, respectively. Then, the incremental update of $\mathcal{I}$, with respect to $\mathcal{A}$, $\mathcal{D}$ and $\mathcal{U}$ takes place as follows:

- Based on the set of deleted clients $\mathcal{D}$, update $\mathcal{I}$ by removing interfaces that are no longer useful, i.e., interfaces that do not serve any client.
- Client updates are handled as client deletions followed by additions of new clients. Hence, $\mathcal{I}$ is further modified by removing the interfaces that had been extracted for the updated clients $\mathcal{U}$.
- Extract interfaces for $\mathcal{A} \cup \mathcal{U}$: (1) add to $\mathcal{I}$, interfaces that fit the needs of the added and the updated clients; (2) improve the clarity of $\mathcal{I}$.

In our example scenario, suppose that the needs of client $c_6$ change. To deal with this situation we remove $i_6$ from the hierarchy that is given in Figure 1(d). Then, an interface $i_6^{new}$ that satisfies the new needs of client $c_6$ is added in the hierarchy (Figure 2(a)). Finally, two generalization relations are established, one between $i_6^{new}$ and $i_{1,5}$, and another between $i_6^{new}$ and $i_7$ (Figure 2(b)).

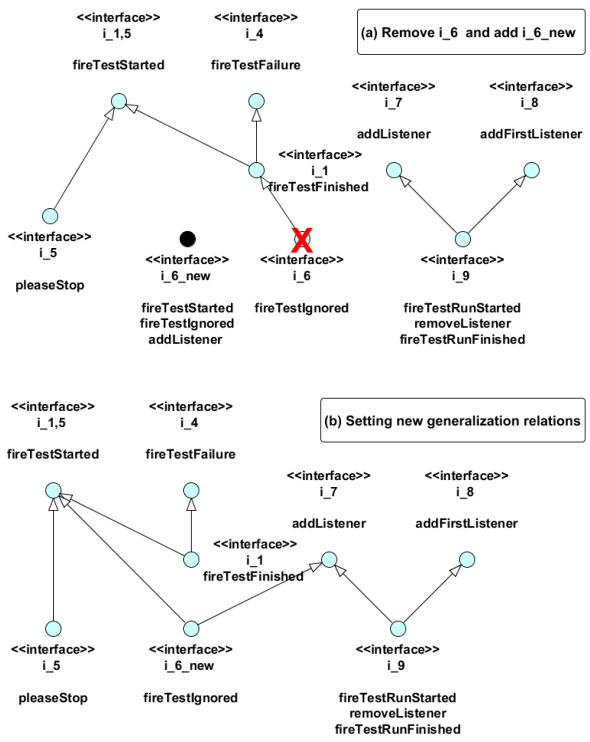


Fig. 2. Illustrating example: Incremental update of the hierarchy.

## III. Status & Emerging Results

As a proof of concept for the proposed method we developed an Eclipse plugin. The plugin uses the Eclipse AST facility to gather information (method declarations, method invocations, etc.) about the classes of a given Java project. Based on this information, it extracts interfaces by applying the proposed method to classes with fat interfaces. At this

[6]We assume that two methods are equal if the method prototypes are the same, by definition in Java interfaces.

[7]We use the term *provided* to refer to the union of (a) the methods declared in an interface and (b) the methods inherited from other interfaces.

TABLE I
CASE STUDIES.

| Case Study | # of Classes with fat interfaces | URL |
|---|---|---|
| JUnit | 44 | junit.org |
| Concordion | 15 | www.concordion.org |
| JavaML | 23 | java-ml.sourceforge.net |
| JAGA | 28 | www.jaga.org/ |
| JHotDraw | 45 | sourceforge.net/projects/jhotdraw |
| BlueJ | 151 | www.bluej.org |

stage, the prototype does not directly refactor the code of the classes. However, to facilitate the work of the developer, it generates UML diagrams that specify the extracted interfaces and their relations.

We performed a study over six open-source projects (Table I) : we used two testing frameworks, JUnit v4.6 and Concordion v1.4.2; JavaML v0.1.7 that offers a collection of machine learning algorithms; JAGA v1.0 that provides a suite of genetic algorithms; JHotDraw v7.0.6, an extensible drawing tool that is based on design patterns; BlueJ v3.1.4, a popular Java development environment designed for beginners. In the case studies the number of classes with fat interfaces ranged from 15 to 151. The average number of method declarations for the classes, varied from 7.88 to 18.55, while the average number of clients, ranged from 6.25 to 11.88.

**Fitness, clarity, and flexibility.** Fitness and clarity are the key enablers of flexibility. Hence, the first issue of our study was to assess the proposed method with respect to these aspects. To this end, we measured the *decoupling degree* (**DD**) that is achieved for the clients $\mathcal{C}$ of a class $c$, when using the interfaces that are extracted for $c$. To measure the decoupling degree for a client $c_j \in \mathcal{C}$, when using an interface $i_{c_j}$ extracted for $c$, instead of using $c$, we employ the Actual Context Distance ($ACD$) introduced by Steimann [6]: $ACD(i_{c_j}, c) = \frac{number\ of\ c\ methods\ -\ number\ of\ i_{c_j}\ methods}{number\ of\ c\ methods}$. The rationale behind $ACD$ is that the class provides methods that the client does not need, while the interface provides exactly the methods that the client needs. Hence, the difference between the number of the class' methods and the number of the interface's methods reflects the amount of decoupling that is achieved. Then, the *decoupling degree* that is achieved for the clients of $c$ is: $\mathbf{DD} = \mathbf{avg}_{c_j \in \mathcal{C}} ACD(i_{c_j}, c)$. Moreover, we measured *the number of duplicated method declarations* that have been eliminated from the interfaces that we extracted for the classes of the case studies.

Figure 3 gives the distribution of the decoupling degree achieved for the case studies. The average decoupling degree that we obtained was quite high, ranging from 69.97% (in the case of JavaML) to 80.96% (in the case of JHotDraw). In all case studies, we further observed *a positive correlation between the number of methods, declared by a class, and the decoupling degree that is obtained for the clients of the class*. To statistically validate this observation, we calculated the Spearman correlation coefficient, $\rho$, for these variables, along with the corresponding p-values (Table II). All the $\rho$ values are positive, indicating that there is indeed a positive correlation between the variables. Moreover, the values are close to 1, indicating a strong correlation.

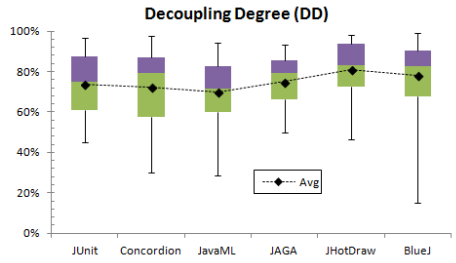


Fig. 3. Distribution of the decoupling degree achieved for the case studies.

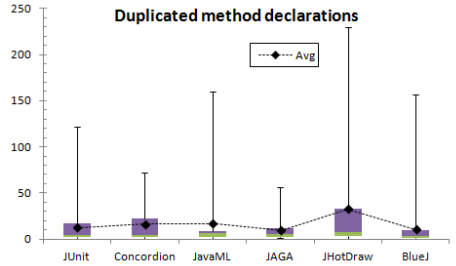


Fig. 4. Distribution of duplicated methods for the case studies.

Figure 4, gives the distribution of the number of duplicated methods that have been eliminated. Overall, the average number of duplicated method declarations that have been eliminated varied quite a lot, from 9.32 (in the case of JAGA) to 32.56 (in the case of JHotDraw). In all case studies, we also observed a positive correlation between the number of duplicated methods that have been eliminated and the number of clients a class has. Due to lack of space we omit these findings.

**Practicality.** The second issue of the study was *to assess the price that we have to pay for fitness, clarity, and flexibility*. To address this issue, we measured the number of interfaces that are extracted for the examined classes. The detailed results that we obtained are given in Figure 5. For each case study, a scatter plot relates the number of extracted interfaces for each class, with the number of the class' clients. Each scatter plot further depicts a solid line that corresponds to the *identity*

TABLE II
SPEARMAN CORRELATION BETWEEN METHODS & DECOUPLING DEGREE.

| Case Study | $\rho$ | Sample size | p-value |
|---|---|---|---|
| JUnit | 0.86 | 44 | 3.26E-14 |
| Concordion | 0.80 | 15 | 1.13E-04 |
| JavaML | 0.83 | 23 | 1.92E-07 |
| JAGA | 0.91 | 28 | 1.65E-12 |
| JHotDraw | 0.83 | 45 | 5.95E-13 |
| BlueJ | 0.86 | 151 | 5.17E-47 |

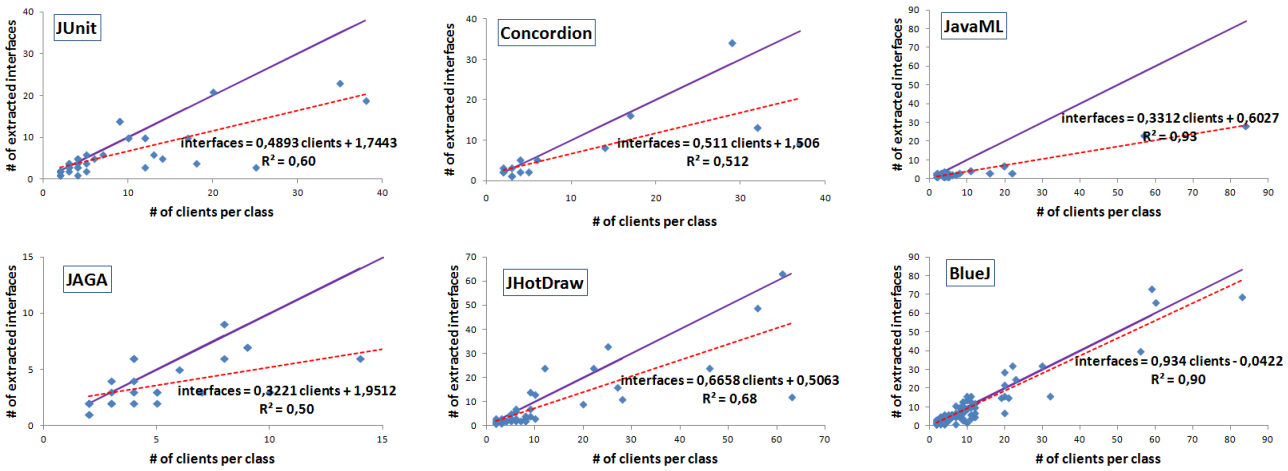


Fig. 5. Extracted interfaces, with respect to the number of clients.

*function* (i.e., *number of interfaces = number of clients*). All points that lie below the solid line correspond to classes for which the number of extracted interfaces is less than the number of clients. With the exception of BlueJ, the main observation that comes out from the results is that *typically the number of interfaces that are extracted for a class is less than the number of clients that depend on the class*. Even in the case of BlueJ, for most classes the number of extracted interfaces is less than, or equal, to the number of clients. To get more insight concerning the relation between the number of extracted interfaces and the number of clients, we further performed a linear regression analysis, which indicated *a linear relation between the number of clients that use a class and the number of interfaces that are extracted for the class*. For most cases, the regression coefficients that we obtained are medium-low (e.g., 0.48 in JUnit, 0.32 in JAGA, 0.33 in JavaML), which again shows that the number of extracted interfaces is usually lower than the number of clients. For BlueJ, the regression coefficient is close to, but still less than 1.

## IV. Next Steps

To sum up, our preliminary results ascertain that *it is feasible to extract fit, clean, and flexible hierarchies of interfaces, which consist of a reasonable number of interfaces, with respect to the number of clients a class has*.

Nevertheless, especially from the perspective of trading between fitness, clarity, flexibility, and practicality, there is still room for further research. In particular, extracting interfaces for every class that provides a fat interface may still clutter a particular software with a large number of interfaces. At the same time, there may be classes with fat interfaces, for which the extraction of interfaces that fit exactly the needs of the clients does not make much sense, or interfaces for which a degree of unclarity may be bearable, reasonable, or even desirable due to certain domain or developer specific concerns. An automated interface extraction approach must account for such issues that can emerge in real-world situations. Hence, a first objective of our future research agenda is to enhance the proposed approach with means that would allow the developers to focus the interface extraction method to the right classes by analyzing potential tradeoffs between the interface extraction benefits and costs. A second objective of our future research agenda is to enhance the proposed approach with means that would allow to customize the interface extraction process with domain and developer specific constraints (e.g., preferences/limits for the desired number of interfaces, methods per interface).

## Acknowledgments

This work received funding from the CHOReOS EU FP7 project no 257178.

## References

[1] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[2] H. Abdeen, H. A. Sahraoui, and O. Shata, "How We Design Interfaces, and How to Assess It," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013, pp. 80–89.

[3] A. F. Yamashita and L. Moonen, "Exploring the Impact of Inter-Smell Relations on Software Maintainability: an Empirical Study," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 682–691.

[4] D. Romano and M. Pinzger, "Using Source Code Metrics to Predict Change-Prone Java Interfaces," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 303–312.

[5] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[6] F. Steimann, "The Infer Type Refactoring and its Use for Interface-Based Programming," *Journal of Object Technology*, vol. 6, no. 2, pp. 99–120, 2007.

[7] R. Adnan, B. Graaf, A. A. van Deursen, and J. Zonneveld, "Using Cluster Analysis to Improve the Design of Component Interfaces," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008, pp. 383–386.

[8] D. Romano, S. Raemaekers, and M. Pinzger, "Refactoring Fat Interfaces Using a Genetic Algorithm," in *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 351–360.

[9] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Addison Wesley, 2005.