Establishing the Source Code Disruption Caused by Automated Remodularisation Tools

Mathew Hall
University of Sheffield
m.hall@dcs.shef.ac.uk

Muhammad Ali Khojaye
Sopra Group
muhammadkhojaye@gmail.com

Neil Walkinshaw
University of Leicester
nw91@leicester.ac.uk

Phil McMinn
University of Sheffield
p.mcminn@shef.ac.uk

***Abstract*—Current software remodularisation tools only operate on abstractions of a software system. In this paper, we investigate the actual impact of automated remodularisation on source code using a tool that automatically applies remodularisations as refactorings. This shows us that a typical remodularisation (as computed by the Bunch tool) will require changes to thousands of lines of code, spread throughout the system (typically no code files remain untouched). In a typical multi-developer project this presents a serious integration challenge, and could contribute to the low uptake of such tools in an industrial context. We relate these findings with our ongoing research into techniques that produce iterative commit friendly code changes to address this problem.**

I. Introduction

As software evolves to meet new requirements, its design invariably deteriorates rendering it harder to maintain. 'Remodularisation' of the source code is a key software maintenance task that is concerned with stemming this problem. Source code remodularisation amounts to rearranging the files or classes into a new set of modules or packages that are (amongst other considerations) more cohesive and loosely coupled.

For any non-trivial project, remodularisation is a highly challenging, time consuming task. To address this problem, numerous automated and semi-automated remodularisation tools have been developed as aids. These tend to operate by analysing the source code in some form (e.g. its dependencies, identifiers, or version history), and derive from this a new set of modules.

Despite over two decades of research into automated software remodularisation tools [1], [2], their transfer into industrial use has been very limited [3]. Previous papers have focussed on addressing the key perceived hindrance—that the proposed changes often fail to make sense with respect to the underlying domain model of the system [3].

However, there is a further limitation to existing approaches: The impact of their solutions is often only determined in abstract, modular terms, without considering the extent of the changes at the source-code level. A minor change at the modular level can lead to disproportionately large changes at the source code level. For example, moving a single class to a different package in a Java system can require multiple changes to the import statements of its client classes. If it has any public static methods, the call sites to these methods need to be changed. If it contains any package-protected methods or variables, their permissions may need to be changed or suitable accessor methods may need to be introduced, which can amount to non-trivial changes to the interfaces of the class or its clients.

Wide-ranging changes to the source code can be notoriously problematic, especially if the software is under active collaborative development, or is used as a library by several other systems. By its nature, software remodularisation changes interfaces, which other developers of the project (and its libraries) will have become accustomed to. A study by Kim et al. [4] showed that such refactorings are correlated with sharp subsequent increases in the number of bugs (indicated by bug-fixes).

In this paper we investigate the relationship between the solutions proposed by automated refactoring tools (specifically Bunch [5]) at a modular level, and the actual changes to the source code that these entail. Besides determining how many lines of code need to be changed, we also explore whether conventional modular measures of similarity (e.g. MoJo [6]) are capable of predicting this impact.

The results indicate that, even for relatively modest systems, the number of changes required tends to run into thousands of lines of code. We also find that there is at best a very weak relationship, and at worst none at all between changes at a modular level (as measured by MoJo / MoJoFM) and changes required to the source code—the impact on source code cannot be predicted from looking at changes to the modular structure alone.

Ultimately, these findings support Bavota et al.'s argument against "Big-Bang" remodularisation [7] and the need for an iterative [8] or localised [9] approach, as well as recent work by Zanetti et al. [10] to make source code a principal consideration in remodularisation. They also raise the questions that we seek to address in our ongoing research: (1) If an iterative approach is to be used, how can we select the best refactorings that will limit disruption of the source code while maximising improvement in the global modular structure? (2) Can we use the automated facilities that are built-in to version repositories, such as pull-requests and hooks?

The core contributions of this paper are as follows:

- A novel approach to measuring the extent of code changes proposed by remodularisation tools that applies the suggestions and measures the size of the changes to the code base.
- A study showing thousands of lines of source code must change when using one of the most popular remodularisation tools (Bunch [1]).

- Insights arising from this study into how automated remodularisation can be made a practical prospect for routine software development.

II. Background

We first present a brief overview of automated software remodularisation techniques. This is followed by a discussion of the practical barriers that have prevented these techniques from being widely applied. We focus here on the specific barrier that is the subject of this paper: the extent to which remodularising the source code can change the contents of the source code.

A. Automated Software Remodularisation

Software systems are conventionally constructed in a modular manner; core elements (classes or other components) tend to be grouped together according to their common functionality or purpose within the system. As software evolves, elements within the system are repurposed to address changes in requirements and the core purpose of different modules can become diluted, making the system harder to understand as a whole.

Remodularisation is the challenge of reversing this process by reorganising it to improve its modular structure. The theoretical number of possible ways in which a system can be reorganised is huge [11], and developing a solution can require an intractable amount of effort and knowledge from the developer. As a consequence, numerous researchers (starting with Hutchens and Basili's work in the mid-eighties [2]) have developed automated remodularisation techniques that seek to minimise this effort. Although these tools are automated in principle, it is generally expected that the final proposed modularisation is to be refined by an expert software developer [12].

The majority of existing automated techniques are based upon established data-analysis techniques such as clustering [13], [14], [15], [16], [5], [17] or formal concept analysis [15], [18]. With both types of approach, the classes or files of the system are characterised by their key features (e.g. other classes upon which they are dependent, file names, identifiers, etc.). Data analysis techniques are subsequently applied to automatically group the files together into modules, according to shared features.

B. Practical Barriers

Despite nearly 30 years of research, automated software modularisation techniques have failed to transfer into widespread use. A key reason for this is their expense. Although they can easily propose new modularisations, these have to then be (1) manually refined into designs that are acceptable to domain experts, and (2) actually implemented as changes to the source code.

The first problem, of manually refining the proposed design, has been frequently observed (most notably by Glorie et al. in their case study that applied Bunch to a large system at Philips Medical Systems [3]). Several techniques have been developed to facilitate this step, such as Bavota et al.'s interactive genetic algorithm [8], or our SUMO supervised modularisation technique [11].

The second issue, however, has received little attention. The task of simply moving classes between packages is perceived to be a mere formality, and has only recently been fully automated by Zanetti et al. [10]. However, the issue of adopting these changes extends beyond applying them to the codebase.

Automated remodularisation tools tend to recommend wide-ranging changes that can affect most of the files. In our experience, running Bunch on a Java system tends to require changes to all of the modules to a greater or lesser extent. Thus, for a non-trivial system, the changes required by Bunch (presuming that they are validated by a developer) could run into thousands of lines of code. The changes required to adopt a new modularisation extend beyond changing package declarations and moving files. References to methods, classes and other attributes must be also updated, including those in the documentation for the class.

In a practical scenario, the software system will be under active development by multiple users. Returning to the Philips Medical Systems example of Glorie et al. [3], their example medical imaging system consists approximately 30,000 source code files, continuously being developed and maintained by over 100 developers on multiple sites (the US, Netherlands and India). In this context, applying such wide-ranging changes would be extremely disruptive, to the point that the disruption would probably undermine the purpose of the code changes—to improve maintainability. Code changes in a collaborative environment need to be gradual, with the understanding and consent of the other developers.

C. Acceptable Code Changes

In stating that certain levels of code change are "too disruptive", it is of course necessary to discuss the question of what level of change *is* acceptable. This question can in part be answered in intuitive terms; the smaller the better. Smaller changes are easier to re-test, are better understood by fellow developers, and are thus less likely to lead to faults.

To put an approximate figure on what might constitute a reasonably acceptable change, we look at existing research into the analysis of version repositories [19], [20]. Research by Alali et al. shows that typical commits are small; the median number of files changed by a commit to gcc is only two, with an average (median) of 14 lines changed [19]. Large commits are extremely rare, often encompassing sweeping changes to the whole codebase. Such large commits include non-functional changes to the code base, such as license header changes, documentation, refactoring, and remodularisation [20].

While these studies show that projects can undergo substantial changes, smaller patches are easier to integrate. In two open source systems, smaller patches have been found to have a higher chance of being accepted than larger patches [21]. Furthermore, care must be taken when making large-scale modifications, as they may introduce unforeseen consequences in the form of bugs in consumers of any APIs [4].

Tonella [18] explored the trade-off that exists between providing an architectural improvement, and the cost of implementing it, using a metric based on the distance between two partitions (similar to Tzerpos and Holt's MoJo score [6]).

Tonella argues that architectural improvement is only one side of the coin; the cost of adopting a new architecture must be considered on balance with the benefits it can provide.

Our premise is that, while remodularisation (and other types of refactoring) tools may be able to suggest helpful changes, it is impractical for those changes to be adopted. We test this premise in the following section with an analysis of the typical commits that arise when using Bunch to perform a system-wide remodularisation.

III. Measuring The Impact of Automated Remodularisations on Source Code

In this section we present a study that investigates the extent to which the solutions proposed by automated remodularisation techniques (in our case Bunch) would disrupt the source code if put into practice. In doing so, we investigate whether the MoJo metric [6] — a non source-code based metric to explore differences in modularisations — provides an adequate basis for estimating this:

RQ 1: What is the impact on the source code of a software remodularisation as proposed by an automated remodularisation tool?

RQ 2: For a given remodularisation, is there a relationship between the number of lines changed and the MoJo distance?

A. Methodology

We used Bunch [1] to perform the remodularisations because it has formed the basis for numerous other studies on modularisation [22], [23], [24], [11], and was used in Glorie et al.'s industrial case study [3]. To obtain our data, we adopted the procedure outlined in the following sections.

1. Acquire Projects: We selected four open source Java case studies (since our tool currently supports Java), shown in Table I. All have previously featured in publications on either software modularisation [10] or papers on mining software repositories [19] (given that this paper is concerned with the hypothetical commit-sizes of commits), with the exception of H2DB.

2. Extract Dependency Graph and SIL file: Bunch requires as input a graph representing the dependencies that exist between software classes. We extracted this from the JAR files for each project using Dependency Finder[1], and filtered out any references to external libraries. We also extracted the SIL file, a simple text file that represents the current modular structure of the system.

3. Clustering with Bunch: We supplied the dependency graph extracted from the previous stage to Bunch. Bunch does not merely produce a single de-facto output, but produces a hierarchy of package clusterings. In line with Mitchell [25], we chose the median layer. Bunch includes a degree of non-determinism, which we account for by running it 30 times for each system.

[1] http://depfind.sourceforge.net/

4. Measure the MoJo distance: We calculated the MoJo distance [6] between the original structure and the version produced by Bunch. We calculated both MoJo and MoJoFM. The latter is an enhanced MoJo variant that allows results to be compared between different systems [26]. For each Bunch result, we computed the number of operations required to transform the existing structure into the Bunch structure, i.e. *MoJo(Original,Bunch)*.

5. Apply the Changes: We have developed a tool that, given a target SIL file, will invoke a series of refactorings to transform the code accordingly. The tool iterates through the classes in the original modularisation and, for each class, compares the package to the corresponding package for the new modularisation. If the packages are different[2], the class is moved using the Eclipse Language Toolkit API. If the destination package does not exist, the tool creates it before moving the class. We applied this tool to modify the existing structure to the proposed remodularised version for each Bunch result.

6. Measure the Size of the Commit: Finally, we analysed the changes as a diff between the new version and the old version. In our experimental setup, we carry this out by preparing a commit to a Subversion (SVN) repository, and by using `svn diff` to compute the difference. As SVN is unable to track rename operations carried out by external tools, we impose this by temporarily moving files back to their original location and explicitly applying the move operation using SVN. We then revert the original file so it is unchanged in the repository. This ensures the diff only contains the modified lines (rather than deletion and addition of whole files). We apply correlation and linear regression to analyse our data to produce answers for our research questions.

B. Results

1) RQ1—The Impact on Source Code: Table I summarises the case studies used in the survey and presents median metrics for the set of Bunch results for each subject. In all cases, the median number of lines changed runs into the thousands, from 1551 for JUnit to 8306 for H2DB. In the case of JUnit, this amounts to a change of almost 10% of its code base. Relating to the commit sizes for gcc of Alali et al.'s example [19], these commits would all be classified as "extra-large" and rarely occurrent. We can conclude that using a system-wide remodularisation tool will produce changes that are difficult to adopt.

Although there is a minor positive correlation between LOC and the number of line changes (Spearman $\rho = 0.53$), this is not particularly reliable and can fluctuate extensively depending on the system in question. For example, Ant is substantially larger than H2DB in LOC terms (72% larger), but both systems would require a similar number of lines changed when remodularised by Bunch.

2) RQ2—Relationship to MoJo / MoJoFM: There is a broad correlation between the MoJo measure and the number of lines changed in the source code. Figure 1 shows how

[2] Some tools such as Bunch by default assign a new package name to every module, regardless of whether it has changed or not. We undo such changes, so that only modules that have actually changed in terms of the classes they contain change their names.

TABLE I. SYSTEMS AND RESULTS (* DENOTES MEDIANS)

Project	Component	Revision	Source Files	LOC	MoJo*	MoJoFM*	Lines Changed*	%LOC*
Ant	main	f5ed8ba	852	220,215	405	50.670	8293.5	3.77
H2DB	(all)	r5717	471	159,411	249	46.255	8306.0	5.21
jEdit	(all)	r23575	533	172,607	282	45.245	4842.0	2.81
JUnit	(all)	4ac902d	174	15,753	71	57.230	1551.0	9.85

the size of the commit changes with the MoJo measure. The dashed line is fitted using ordinary least squares regression and indicates that MoJo can approximate the relative size of a commit, within each system.

However, MoJo values are specific to each subject system [26], making it a poor predictor for the number of edits made to an arbitrary system. The specific relationship between differences in MoJo value and lines of code changes can fluctuate hugely from one project to the next (indicated by the different slopes of the regression lines in Figure 1). Accordingly, the intercept term for each model varied substantially between case studies as well. This means, ultimately, that it is not possible to predict the number of lines changed, given only the MoJo value of the modularisation and the number of classes in the system. Although MoJo can give insights into the quantity of changes reflected in two modularisations for the *same* system, they cannot be generalised to estimate the commit size for any system. This lack of generality is documented in work studying comparison of different algorithms [6], [26].

We performed the same modelling using MoJoFM instead of MoJo and found it was also unable to offer generalised prediction of the commit size. The coefficients of determination r^2 for models built using MoJoFM were almost identical to those produced using MoJoFM, which is explained by the linear relationship between MoJo and MoJoFM. Using MoJoFM yields negative coefficients, due to its properties as a metric, rather than the cost function of MoJo.

For this research question, we find that, although MoJo and MoJoFM can provide a relative estimate for the refactoring cost (in terms of commit size) between two competing solutions, they cannot offer a general estimate for any system. Thus, we conclude that MoJo, and MoJoFM, are unsuitable for estimating the real cost of transforming the code base to reflect a restructuring proposed by Bunch.

C. Threats to Validity

Our experiment design necessarily introduces some threats to validity which we have addressed to the extent possible in our methodology.

We measured commits in terms of the total lines added and removed from each source file in the subject system to reflect the size of the corresponding patch. Consequently, edits to existing lines are counted twice. This limitation arises from the diff tool, which does not distinguish between a changed line and addition and deletion of unrelated lines.

Our preliminary work is based on a small survey of sample systems, all of which are implemented in Java. The small population of systems means our results may not generalise, however we ensured our choice of case studies was diverse to mitigate this threat to the extent possible. Our tool only works on Java systems, which makes us unable to analyse remodularisation when applied to systems built using different programming paradigms. This is a technical limitation which we will overcome as we develop our tooling; our approach is applicable wherever a remodularisation approach can be applied as it only relies on the source code of the system.

IV. CONCLUSIONS AND FUTURE WORK

This paper investigates the "disruption" of the code base that would be required to implement solutions proposed by remodularisation algorithms. Without any linkage to the code itself (the identifiers that must be changed and the impact on consumers of the remodularised code), structural measures can only paint a partial picture of the practicalities of a remodularisation algorithm. Our findings indicate that the solutions often lead to thousands of changes, scattered across the code base. It also indicates that the MoJo and MoJoFM measures are not suitable indicators for the extent of these changes.

Given the invariably extensive impact on source code, this clearly supports Bavota et al.'s argument that "big-bang" system reorganisations are impractical [7]. However, localised remodularisation [7], [27] cannot offer a global solution that large-scale remodularisation can. Consequently, there are several vital questions that have not yet been addressed to remedy this problem and enable global remodularisation to take place:

- How can an "idealised" target remodularisation be decomposed into smaller refactorings that are manageable, but also not too trivial to irritate the developer?
- How should these refactorings be prioritised?
- Is it possible to exploit mechanisms embedded within change-management systems to facilitate the interaction between the automated systems and developers, such as pull-requests or hooks?
- How can such an approach become adaptive, to adjust its target to accommodate ongoing software modifications that are potentially unrelated to the structural reorganisation?
- How can it encompass feedback from the developers, in terms of changes that should be avoided, or particular elements that ought to belong together?

These questions delineate our ongoing and future work. We are focussing on a source-code centric remodularisation framework, founded upon our (non source-code based) iterative SUMO algorithm [11]. Our ultimate goal is to develop a software remodularisation agent similar to Di Penta et al.'s "Renovation Framework" [28] that, in addressing the questions above, unobtrusively remodularises the system in a continuous fashion, whilst taking any specific positive or negative feedback from developers into account.

ACKNOWLEDGMENT

This work was supported by the EPSRC grants REGI (EP/F065825/1) and RECOST (EP/I010165/1).

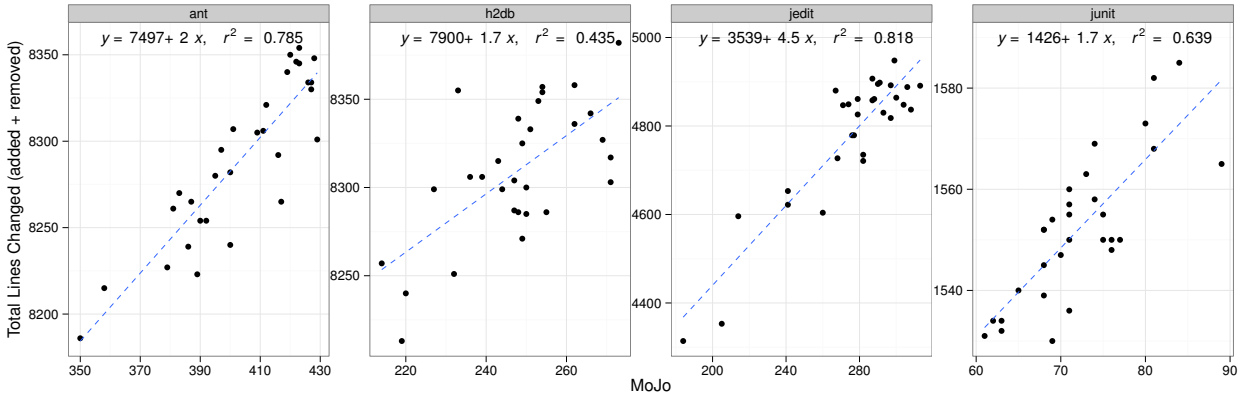

Fig. 1. MoJo values correlate with with commit size. The dashed line is the fitted using ordinary least squares regression, given by the equation.

REFERENCES

[1] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *ICSM'99: Proceedings of the IEEE International Conference on Software Maintenance.*, 1999, pp. 50–59.

[2] D. H. Hutchens and V. R. Basili, "System structure analysis: Clustering with data bindings," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 8, pp. 749–757, 1985.

[3] M. Glorie, A. Zaidman, A. van Deursen, and L. Hofland, "Splitting a large software repository for easing future software evolution—an industrial experience report," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 113–141, Mar. 2009.

[4] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of API-level refactorings during software evolution," in *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*, May 2011, pp. 151–160.

[5] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[6] V. Tzerpos and R. C. Holt, "MoJo: a distance metric for software clusterings," *WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering*, pp. 187–193, 1999.

[7] G. Bavota, A. Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, Oct. 2012.

[8] G. Bavota, F. Carnevale, A. D. Lucia, M. D. Penta, and R. Oliveto, "Putting the Developer in-the-Loop: An Interactive GA for Software Remodularization," in *SSBSE 2012: Proceedings of the 4th International Symposium on Search Based Software Engineering*, ser. Lecture Notes in Computer Science, vol. 7515, 2012, pp. 75–89.

[9] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Transactions on Software Engineering Methodology*, vol. 23, no. 1, p. 4, Feb. 2014.

[10] M. S. Zanetti, C. J. Tessone, I. Scholtes, and F. Schweitzer, "Automated software remodularization based on move refactoring: a complex systems approach," in *MODULARITY '14: Proceedings of the 13th international conference on Modularity*, Apr. 2014, pp. 73–84.

[11] M. Hall, N. Walkinshaw, and P. McMinn, "Supervised software modularisation," in *ICSM 2012: Proceedings of the 28th IEEE International Conference on Software Maintenance*, 2012, pp. 472–481.

[12] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse-engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, Dec. 1993.

[13] N. Anquetil and T. C. Lethbridge, "Recovering software architecture from the names of source files," *Journal of Software Maintenance*, vol. 11, no. 3, pp. 201–221, 1999.

[14] ——, "Experiments with clustering as a software remodularization method," in *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, 1999, pp. 235–255.

[15] A. van Deursen and T. Kuipers, "Identifying objects using cluster and concept analysis," in *ICSE '99: Proceedings of the 1999 International Conference on Software Engineering*, 1999, pp. 246–255.

[16] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 150–165, 2005.

[17] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *ASE 2011: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 552–555.

[18] P. Tonella, "Concept analysis for module restructuring," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 351–363, Apr. 2001.

[19] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension*, 2008, pp. 182–191.

[20] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits," in *MSR '08: Proceedings of the 2008 International Working Conference on Mining software Repositories*, May 2008, pp. 99–109.

[21] P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!" in *MSR '08: Proceedings of the 2008 International Working Conference on Mining software Repositories*, May 2008, pp. 67–76.

[22] K. Mahdavi, M. Harman, and R. Hierons, "A multiple hill climbing approach to software module clustering," in *ICSM 2003: Proceedings of the 2003 International Conference on Software Maintenance*, 2003, pp. 315–324.

[23] K. Praditwong, M. Harman, and X. Yao, "Software Module Clustering as a Multi-Objective Search Problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2011.

[24] M. Shtern and V. Tzerpos, "Evaluating software clustering using multiple simulated authoritative decompositions," in *ICSM 2011: Proceedings of the 27th IEEE International Conference on Software Maintenance*, 2011, pp. 353–361.

[25] B. S. Mitchell, "A Heuristic Search Approach to Solving the Software Clustering Problem," Ph.D. dissertation, Drexel University, 2002.

[26] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, 2004, pp. 194–203.

[27] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse, "Towards automatically improving package structure while respecting original design decisions," in *WCRE 2013: Proceedings of the 20th Working Conference on Reverse Engineering*, 2013, pp. 212–221.

[28] M. D. Penta, M. Neteler, G. Antoniol, and E. Merlo, "A language-independent software renovation framework," *Journal of Systems and Software*, vol. 77, no. 3, pp. 225–240, 2005.