Research Note
RN/16/07

An Empirical Study of Cohesion and Coupling: Balancing Optimisation and Disruption

December 2, 2016

Matheus Paixao[1], Mark Harman[1], Yuanyuan Zhang[1], Yijun Yu[2]
Affiliation: University College London[1], Open University[2]
{matheus.paixao.14, mark.harman, yuanyuan.zhang}@ucl.ac.uk, y.yu@open.ac.uk

Abstract

Search based software engineering has been extensively applied to the problem of finding improved modular structures that maximise cohesion and minimise coupling. However, there has, hitherto, been no longitudinal study of developers' implementations, over a series of sequential releases. Moreover, results validating whether developers respect the fitness functions are scarce, and the potentially disruptive effect of search-based re-modularisation is usually overlooked. We present an empirical study of 233 sequential releases of 10 different systems; the largest empirical study reported in the literature so far, and the first longitudinal study. Our results provide evidence that developers do, indeed, respect the fitness functions used to optimise cohesion/coupling (they are statistically significantly better than arbitrary choices with $p << 0.01$), yet they also leave considerable room for further improvement (cohesion/coupling can be improved by 25% on average). However, we also report that optimising the structure is highly disruptive (on average more than 57% of the structure must change), while our results reveal that developers tend to avoid such disruption. Therefore, we introduce and evaluate a multiobjective evolutionary approach that minimises disruption while maximising cohesion/coupling improvement. This allows developers to balance reticence to disrupt existing modular structure, against their competing need to improve cohesion and coupling. The multiobjective approach is able to find modular structures that improve the cohesion of developers' implementations by 22.52%, while causing an acceptably low level of disruption (within that already tolerated by developers).

Keywords: Software Modularisation, Software Evolution, Multiobjective Search

1 Introduction

Software modularisation is almost as old as the concept of software engineering itself. The notions of cohesion and coupling were introduced in the 1970s [58]. Cohesion is the degree of relatedness enjoyed by code elements residing in the same abstract module, while coupling is the relatedness between modules. There is long-established evidence that software structure tends to degrade as the system evolves [26][27][57]. Therefore, one goal of software modularisation research, is to find ways to improve modular structure, by increasing cohesion and reducing coupling.

Search Based Software Engineering (SBSE) techniques have been widely-studied and developed as one way to automate this structural modular improvement process, guided by fitness functions that capture structural cohesion, coupling and combinations thereof. Structural cohesion/coupling is typically measured in terms of dependencies between elements. It is structural, rather than semantic, because it takes no account of the degree of semantic relations between elements, other than that which is captured through dependence measurements [9].

Many different search techniques have been proposed and developed that automate the search for improved modular structure. However, despite more than 30 publications on search based modularisation, few studies [13][11] have performed an evaluation of the disruptive effects that automated modular improvement may cause on the original modular structure of the software systems under study. A thorough study of the disruption caused by modular restructuring is needed, because there is evidence that software engineers tend to resist structural and architectural improvement in favour of similarity and familiarity [57]. Therefore, high levels of disruption might undermine the industrial uptake of techniques for software re-modularisation.

Moreover, most of the surveyed publications on search based automated re-modularisation consider only a single version of the systems under study, ignoring the systems' history of previous releases. A study involving a series of consecutive releases would be required in order to understand software engineers' decisions with respect to cohesion/coupling and the disruption that would have been caused by automated attempts to improve cohesion/coupling.

In this paper, we provide the first study of search based modularisation that considers both the opportunities for improving software structure and the consequent disruption that accrues as a result, over a series of subsequent releases of software systems. This is also the largest study in search based modularisation: we study 233 releases of 10 different open source software systems, from which we extracted the modular structure data.

We start by investigating the validity of the quality metrics that previous work on search based modularisation has used to improve software modularity. Our survey reveals that out of more than 30 papers that have previously studied this problem, many have used the Modularisation Quality (MQ) metric [31][38] to assess modularity quality. We therefore validate the use of this metric, investigating whether the existing modular structure implemented by developers respects MQ.

We complement our study of MQ by measuring the raw cohesion of each system. The raw cohesion is simply the number of dependencies that reside within a single module, and therefore do not cross any module boundary. The raw coupling, is the obverse; the number of dependencies that cut across module boundaries. Given the proposed modular structure of a system, we can thus measure raw cohesion/ coupling, simply by counting intra- and inter-dependencies between elements. Since raw coupling is the obverse of raw cohesion, we need only measure one of the two properties. Traditional search based modularisation does not use raw cohesion/coupling as a fitness function, because it would result in the algorithm moving all elements into one single module (with maximal cohesion and zero coupling). Such a 'god class' structure is undesirable [10], and various previous authors developed techniques to avoid this [38][17][47]. Although raw cohesion/coupling cannot be used to optimise the modular structure, it can be used to evaluate the quality of solutions found by search. Hereinafter, when we refer to 'cohesion', we mean this simple 'raw cohesion' metric.

In order to provide an evidence-based assessment of the degree to which developers' implementations are cohesion-respecting and MQ-respecting, we introduce an approach to validation that is grounded in frequentist inferential statistics, widely used elsewhere in software engineering, and particularly recommended for SBSE [4][15]. Using this statistical approach, we provide evidence that developers choose modular structures that are highly cohesion- and MQ-respecting. Furthermore, we show that although developers choose solutions in the local neighbourhood that have better cohesion and MQ values than at least 97.3% of the possible alternatives, in every release of every system, the developers implementations are, nevertheless, suboptimal regarding both cohesion and MQ.

This motivates the study of the degree to which search based modularisation could automatically 'improve' on the developer-implemented modular structure, according to cohesion and MQ. In order to answer this question, we empirically studied the widely-proposed hill climbing technique (Bunch) for finding improved modular structures [38]. The hill climbing approach is simple and fast, and has publicly available implementations, making it an obvious first choice for any developer seeking to use search based techniques for modular improvement. After modifications to the original approach to cope with the large scale real world systems being studied, we found that, in most releases, automated modularisation does find modular structures with statistically significantly better cohesion and MQ values, and with large effect size.

Of course, re-modularisation may not be so straightforward in practice: if there was a dramatically improved modular structure available to the developers, then it seems reasonable to ask why software engineers have not adopted it. There are two potential explanations for this:

1. The developers are unaware of any better solutions; the search space is simply too large and it defeats human-based search.
2. The developers are aware of at least one better solution, but *choose* not to implement any of the better solutions.

In all cases, we found that, even within the nearest neighbourhood to the developers' given implementation, there were always alternatives with improved cohesion/coupling. That is, improvement could be achieved simply by moving a single element from one module to another, in all of the 233 releases studied. This provides evidence that it is unlikely that developers were unaware of *any* better solution, so we turn our attention to the second possible explanation above.

If developers could easily find a better solution, even with a simple nearest neighbourhood search, why did they choose not to implement it? One possible explanation we chose to investigate, relates to the recent observation that developers are prepared to build up technical debt [22][32]; resisting the temptation to restructure systems, and tolerating degradation in structure, in order to obtain fast delivery, retain familiarity of the existing structure and/or to preserve some other property of interest. Specifically, we investigate the degree of disruption that would be caused by moving to an improved modular structure, that increases cohesion and reduces coupling. We measure disruption as the number of elements and modules that would need to be moved or merged, according to the MoJoFM metric [56]. The results were striking: while a variation of the well-known Bunch automated re-modularisation approach can improve cohesion by 25% on average, these improvements result in 57% disruption.

This provides empirical evidence that developers are reluctant to disrupt the modular structure, even when this might lead to improved cohesion/coupling. Unfortunately, most of the previous work on search-based re-modularisation has ignored this disruptive effect, leaving open many questions that we seek to answer in the present paper, such as how large the effect is and how often it occurs, whether it is correlated with the improvements achievable, and the degree to which it could be avoided, while maintaining structural improvement.

We found that, although any modular improvement inherently inflicts some degree of disruption, in general, the disruption caused by the best improvement found by standard SBSE approaches, for every release of every system, is smaller than the average disruption. Furthermore, we found no evidence that cohesion improvement is correlated with disruption increase. This is a particularly attractive finding, because it points to the possibility that a multiobjective search-based approach may be able to find balances and trade-offs between modular disruption and improvement. This more positive finding, thereby motivated our final set of experiments, in which we introduced, implemented and evaluated a novel multiobjective search based modularisation technique.

Our new approach to automated re-modularisation seeks pareto-optimal balances between disruption, as measured by MoJoFM, and improvement. On average, within the developer-determined 'acceptable' level of disruption for each system, which was calculated through longitudinal analysis between developers-implemented releases, our multiobjective approach was able to find solutions with average 22.52% and 55.75% improvements for cohesion and MQ, respectively.

The primary contributions of this paper are the findings concerning the behaviour of both developers and existing SBSE techniques for automated re-modularisation (on 233 releases of 10 different software systems), the identification of disruption as an important problem for automated re-modularisation, and the novel multiobjective approach we introduce and evaluate to tackle this problem. Our empirical study and evaluation is the largest study of search-based re-modularisation hitherto reported in the literature, and its scientific findings have an actionable conclusion for researchers and practitioners; any and all approaches to re-modularisation (search based or otherwise) need to take account of (and balance) the disruption they cause, against the improvement they offer.

The rest of this paper is organised as follows: Section 2 discusses the related work that was collected during our survey, alongside some formal definitions and background of automated software modularisation. Section 3 describes how we collected the modular structure data of the systems we consider, and Section 4 presents the empirical study we performed over the 233 releases of the 10 systems we collected. In addition, Section 5 reports a qualitative analysis of the results achieved. Finally, Section 6 discusses the threats to the validity of our empirical study and Section 7 presents our conclusions and points out some future research directions.

2 Related Work and Background

We collected publications that use search based techniques to improve the modular structure, where cohesion/coupling and combinations thereof are used to assess the quality of the modularisations. We cannot guarantee that we covered every paper, but we believe this survey presents a reasonable sample of the work performed by the search based software modularisation community.

Table 1 summarizes the 35 papers we collected, and presents them sorted by year of publication. For each paper we report whether it employs a Single Objective (SO) or MultiObjective (MO) optimisation approach, and what fitness functions are used to guide the search. We also report which search algorithms are used to perform the modularisation, and how many systems and releases were considered in each evaluation.

As one can see, the work on search based software modularisation dates back to late 1990s

Table 1: Related work in Search Based Software Modularisation sorted by year of publication

Paper	Year	Optimisation Approach	Fitness Function	Search Algorithm	Number of Different Systems Used	Number of Releases Studied
Mancoridis et al. [31]	1998	SO	MQ	HC	5	5
Doval et al. [12]	1999	SO	MQ	GA	1	1
Mancoridis et al. [30]	1999	SO	MQ	HC	1	2
Mitchell et al. [33]	2001	SO	MQ	HC	7	7
Harman et al. [16]	2002	SO	Coh, Cop	HC, GA	7	7
Mitchell et al. [36]	2002	SO	MQ	HC	5	5
Mahdavi et al. [28]	2003	SO	MQ	HC	19	19
Mitchell et al. [37]	2003	SO	MQ	HC	13	13
Harman et al. [17]	2005	SO	MQ, EVM	HC	6	6
Seng et al. [51]	2005	SO	Coh, Cop, Complexity, Cycles, Bottlenecks	GGA	1	1
Shokoufandeh et al. [52]	2005	SO	MQ	HC and Spectral Algorithm	13	13
Mitchell et al. [38]	2006	SO	MQ	HC	2	2
Mitchell et al. [39]	2008	SO	MQ	HC	5	5
Abdeen et al. [1]	2009	SO	Coh, Cop, Cycles	SA	4	4
Mamaghani et al. [29]	2009	SO	MQ	Hybrid GA	5	5
Praditwong et al. [46]	2011	SO	MQ	GGA	17	17
Praditwong et al. [47]	2011	MO	MCA, ECA	Two-Archive GA	17	17
Barros et al. [5]	2012	MO	MCA, ECA	NSGA-II	13	13
Bavota et al. [7]	2012	SO and MO	MQ, MCA, ECA	GA, NSGA-II	2	2
Hall et al. [14]	2012	SO	MQ	HC	5	5
Abdeen et al. [2]	2013	MO	Coh, Cop, Modifications	NSGA-II	4	4
Kumari et al. [24]	2013	MO	MCA, ECA	Hyper-heuristics	6	6
Ouni et al. [42]	2013	MO	Fixed Defects, Effort	NSGA-II	6	6
Hall et al. [13]	2014	MO	MQ	HC	4	4
Barros et al. [6]	2015	SO	MQ, EVM	HC	1	24[1]
Jeet et al. [20]	2015	SO	MQ	BHGA	6	6
Mkaouer et al. [40]	2015	MO	Coh, Cop, MO, NCP, NP, SP, NCH, CHC	NSGA-III,IBEA, MOEA/D	5	5
Paixao et al. [44]	2015	MO	MCA, ECA	Two-Archive GA	1	1
Saeidi et al. [50]	2015	SO and MO	MQ, CQ	HC, Two-Archive GA	10	10
Candela et al. [11]	2016	MO	Structural and Contextual Coh/Cop	NSGA-II	100	100
Huang et al. [19]	2016	SO and MO	MQ, MCA, ECA	MAEA-SMCPs, GGA, GNE	17	17
Huang et al. [18]	2016	SO	MQ, MS	HC, GAs and MAEA	17	17
Jeet et al. [21]	2016	SO	MQ	HC, five GA variations	7	7
Kumari et al. [23]	2016	MO	MCA, ECA	Hyper-heuristics	12	12
Ouni et al. [43]	2016	MO	Fixed Defects, Coherence, Effort, Change History	NSGA-II	6	12
Paixao et al.	This Paper	SO and MO	MQ, Disruption	HC, Two-Archive GA	10	233

[31][30][12], with the proposal and first evaluations of the Bunch tool. The MQ metric was first proposed as Bunch's fitness function, and it is still the most used metric in search based modularisation to date. In fact, suites of quality metrics more recently used for multiobjective modularisation [47][7][5] include MQ as one of the metrics to be optimised.

2.1 Modular Structure Representation

In this paper, the modular structure of each release under study is represented as a Module Dependency Graph (MDG) [31]. An MDG is a directed graph $G(C, D)$ where the set of nodes C represents the code elements of the system and D represents the dependencies between elements. Usually, software systems organise their elements into higher level modules, which are indicated as clusters of nodes in the MDG. The allocation of nodes of the MDG to different clusters, alongside the dependencies between the nodes, is one way to represent the release's modular structure. An example of a MDG is presented in Figure 1.

Since all the systems under study are implemented in Java (see Section 3), we are going to use the Java terminology to refer to the code elements and high level modules; the elements are thus the classes and interfaces, while the modules are the packages. In this paper, both classes and interfaces will be referred to simply as "classes". Dependencies occur by method call, field access, inheritance and interface implementation.

For each release of each system, the set of classes C is represented by $C = \{c_1, c_2, \ldots, c_N\}$, where N is the number of classes in the release. A dependency $d(c_x, c_y)$ indicates that class c_x depends on class c_y to correctly deliver its functionality. The set of all dependencies is represented by $D = \{d(c_x, c_y) \mid c_x, c_y \in C\}$. The set of packages P in a release is depicted by $P = \{p_1, p_2, \ldots, p_M\}$, where M is the number of packages in the release.

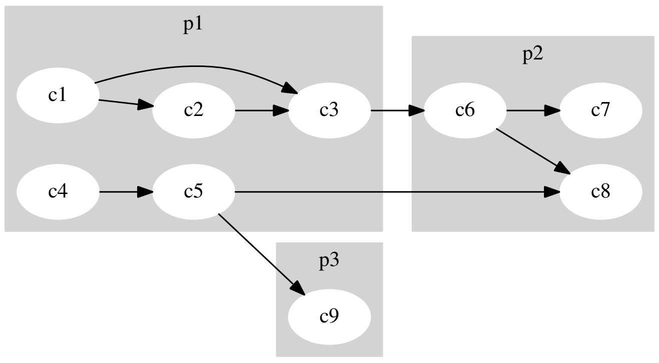

Figure 1: Example of a Module Dependency Graph that is used to represent the modular structure of the systems under study. Nodes represent code elements and edges represent dependencies between elements. Clusters of nodes (grey regions) indicate high level modules.

2.2 Modular Structure Quality Metrics

The Modularisation Quality (MQ) metric was proposed by Mancoridis et al. [31] to guide optimisation algorithms in the allocation of classes to highly cohesive and loosely coupled packages. In order to improve MQ's performance and quality assessment, the metric was re-formulated over the years [33][34], and its most recent incarnation [38] is adopted.

MQ consists of assigning scores to each package in the system, measuring the packages' individual trade-off between cohesion and coupling. The cohesion of a package p_i is represented by $coh(p_i)$, and it is computed as the number of dependencies between classes within package p_i. Accordingly, the coupling $cop(p_i)$ of package p_i is computed as the number of dependencies from classes within p_i to classes in other packages in the system. The MQ value of the overall system is computed as presented in Equations 1 and 2:

$$MQ = \sum_{i=1}^{P} MF(p_i) \tag{1}$$

$$\text{and, } MF(p_i) = \begin{cases} 0, & \text{if } coh(p_i) = 0 \\ \dfrac{coh(p_i)}{coh(p_i) + \frac{cop(p_i)}{2}}, & \text{if } coh(p_i) > 0 \end{cases} \tag{2}$$

The MQ is thus given by the sum of the Modularisation Factors (MF) of each package p_i in the system. MF(p_i) represents the trade-off between cohesion and coupling for package p_i. Since the dependencies involved in the measurement of the packages' coupling will be double counted during MQ computation, $cop(p_i)$ is divided by 2.

MQ is a function of the allocation of classes to packages; therefore, the MQ search space is composed by all possible allocations of classes to packages in the system. In this context, we define the k–neighbourhood as the subset of the MQ search space that can be achieved by performing k modifications to the original allocation of classes to packages that was implemented by the developers.

The raw cohesion/coupling of a system are measured by summing the cohesion/coupling of its packages. These are straightforward assessments of how many of the system's dependencies are contained in the same package and how many are cutting across the packages boundaries. Since raw cohesion/coupling are the obverse of each other, we need to measure only one of these properties, and for the rest of this paper the raw cohesion, or simply 'cohesion' of the system is computed as presented in Equation 3.

$$COH = \sum_{i=1}^{P} coh(p_i) \tag{3}$$

Apart from the selected metrics presented above, other measurements of structural cohesion and coupling have also been proposed [41] to account for different types of dependencies and different granularity levels. Recent studies reported quantitative and qualitative assessments of these metrics by investigating open source systems and interviewing developers [11][53]. As previously mentioned, MQ is the most used quality metric in search based re-modularisation (see Table 1), yet evidence that software systems respect this metric is scarce. Our empirical study performs an incremental assessment of the level of respect open source systems have to MQ; therefore, complementing previous literature and providing insights to the search based modularisation community regarding its most used fitness function.

2.3 Modular Structure Disruption

The disruption caused by an automated modularisation technique can be measured as the amount of change developers need to perform in order to adopt the solution proposed by the search algorithm, where such disruption may be assessed at both source code and modular structure levels. A previous study by Hall et al. [13] measured how many lines of code ought to be added/changed in order to apply solutions found by search based modularisation approaches. Apart from showing that developers would have to change up to 10% of their code base to adopt solution proposed by automated approaches, they also showed that the LOC to be changed strongly correlates with the modular structure disruption metric MoJoFM [56].

Mkaouer et al. [40] used the number of refactoring operations as a measurement of system disruption to be minimised in a search based many-objective approach for re-modularisation. However, operations at different granularity levels, such as *move method* and *move class*, have the same weight in the disruption computation, even though coarse grained and fine grained refactorings have a different impact in the system's modularity.

In the work by Ali et al. [42][43], the authors proposed a disruption assessment of refactoring operations based on the number of operations to be performed, where each operation is wheighted by a complexity factor. In this formulation, the possible refactoring operations also include different granularity levels, e.g. *pull up method* and *extract class*, and the different wheights are based on the authors expertise.

As argued in a recent work by Candela et al. [11], a modular level disruption metric, such as MoJoFM, better describes the "mental model" developers have of their systems. Therefore, we draw inspiration from the study performed by Candela et al., and adopt a disruption measurement that is based on the widely used [11][7][35][25] MoJoFM metric.

Given two different modularisations A and B of the same system, $MoJoFM(A, B)$ accounts for the proportional number of *Mo*ve and *Jo*in operations that are necessary to transform A in B, such as presented in Equation 4.

[1]Barros et al. [6] uses 24 releases for the system's evolution analysis, but the re-modularisation is performed in only one release.

$$MoJoFM(A, B) = (1 - \frac{mno(A, B)}{max(mno(\forall A, B))}) \times 100\% \tag{4}$$

In this paper, a *Mo*ve operation represents moving a class from its original package to another package in the system, while the *Jo*in operation represents the merge of two packages. The distance between A and B is the minimal number of operations that transform A in B, computed by $mno(A, B)$, and this value is normalised by the maximum distance between any possible modularisation partitioning of the system (denoted by $\forall A$) and B. MoJoFM is a non-trivial metric to compute, and for more technical details the reader is referred to [56].

Finally, given the original developers' implementation A and a solution B suggested by an automated modularisation technique, we propose DisMoJo in Equation 5, a disruption metric based on MoJoFM that measures how much of the original implementation developers would need to change to adopt the modular optimised solution.

$$DisMoJo(A, B) = 100\% - MoJoFM(A, B) \tag{5}$$

3 Software Systems Under Study

In this section we describe the systems we study in our empirical study of cohesion/coupling behaviour and optimisation, including the selection criteria we employed, the process for extracting the modular structure data and a short description of each system.

The primary criteria for selecting software systems to study in our empirical investigation was the availability of at least 10 subsequent releases, so that we could evaluate more than one version of the systems and not only the latest one, like in most of the related work. We conjectured that 10 releases would be sufficient for our analysis.

We avoided general libraries and APIs that provide features that are not necessarily related to each other in terms of code dependencies. We believe these kind of systems naturally have a good modular structure in terms of cohesion/coupling, and therefore, would not be valuable for our investigation. Thus, libraries such as `Commons Math`, for example, were avoided.

As a result, we selected 10 open source Java systems, which are briefly described in Table 2. The number of releases of the systems under study varies from 10 to 47, with a median of 17 releases per system. Moreover, the median number of classes varies from 150 to 895 and the median number of dependencies between classes varies from 568 to 6690, indicating that these are non-trivial medium to large real world software systems.

We employed a reverse engineering approach based on static analysis to obtain the modular structure of each release of each system. In order to do so, we used the `pf-cda` [3] tool to instrument the `jar` files of each release and subsequently extract the packages, classes and dependencies.

In order to facilitate replications of this study, we make available all 233 modularity datasets in our supporting web page[2]. In addition, the web page also contains all results from this investigation, including further details elided for brevity in this paper.

[2]`http://www0.cs.ucl.ac.uk/staff/mpaixao/cohCop/index.html` (Please note the url will be activated upon peer-reviwed publication)

Table 2: Open source systems used in the empirical evaluation of cohesion/coupling behaviour and optimisation. For each system, we report the number of releases and the median number of packages, classes and dependencies over releases. Finally, we report the median number of releases, packages, classes and dependencies for all systems.

Systems	Description	Releases	Packages	Classes	Dependencies
Ant	Tool to perform the 'build' of Java applications	30	25	576	2567
AssertJ	Library of assertions for Java	12	15	467	2095
Flume	Java logging API	10	17	255	849
Gson	Google's converter of Java objects to JSON	15	6	153	724
JUnit	Java unit testing framework	20	23	196	734
Nutch	Java web crawler	13	18	272	1007
PDFBox	Java PDF manipulation library	31	48	496	3049
Pivot	Platform for building Installable Web Applications	12	13	150	568
Procyon	Java decompiler	47	36	895	6690
Proguard	Java code obfuscator	43	18	329	3513
All	-	17	18	300	1551

4 Empirical Study

This section describes and presents the results of the empirical study we carried out in this paper to investigate cohesion/coupling behaviour and optimisation. Each of our research questions will be presented and answered, followed by a discussion of the findings.

4.1 RQ1: Is there any evidence that open source software systems respect structural measurements of cohesion and coupling?

By answering RQ1, we seek to investigate whether there is any evidence that the modular structure of existing software respects both raw cohesion and the MQ metric. We chose raw cohesion because it is a simple and intuitive measurement, and MQ because it is the most used metric in the automated software modularisation literature. Intuition suggests that developers do care about cohesion/coupling, and so we expect existing systems to exhibit *some* degree of 'respect' for these metrics.

We could survey developers with a questionnaire in order to discover a subjective self-assessment of the degree to which they care about these metrics, but such a study would be vulnerable to bias; developers may believe that they *ought* to care about these metrics, since cohesion/coupling have been recommended for many years [58][49][54]. Such feelings may lead to implicit or explicit bias that may influence developers' self-assessment of the importance that they attach to measurements of cohesion/coupling. Moreover, any such assessment would be inherently subjective.

Therefore, although such results would undoubtedly be interesting, we choose to focus on a quantitative assessment of the degree to which the existing modular structure chosen by developers respects both the raw cohesion and the MQ metric.

In order to provide such a quantitative assessment of the degree of agreement with these metrics, we propose three different techniques, each of which produces a probabilistic assessment that can be used as the basis for an inferential statistical argument, concerning the likelihood of rejecting the Null Hypothesis (that the modular structure takes no account of the modularity quality metrics).

RQ1.1: How does the solution implemented by developers compare to a purely random allocation of classes to packages? As a simple baseline, we start by considering purely random allocation of classes to packages. Therefore, we are assuming the following Null Hypothesis H_0*: The modularity measurements of the releases of the studied open source software systems follow a purely random distribution.* That is, we assume, as a Null Hypothesis, that

Table 3: Likelihood of finding a modular structure with superior measurements of structural cohesion/coupling than that produced by the systems' developers, according to 3 search strategies. PRD simply searches for random allocations of classes to packages, while the other two techniques search the neighbourhood of the solution implemented by the systems' developers. kRNS randomly searches for solutions in the k–neighbourhood of the developers' solution, by moving k classes to randomly selected packages, while SNS systematically searches the k–neighbourhood for $k = 1$, by moving each class to one of each of the other packages. Results indicate the percentage of solutions found that improve the modular structure, as assessed using raw cohesion and MQ.

Systems	Purely Random Distribution (PRD)		k–Random Neighbourhood Search (kRNS)		Systematic Neighbourhood Search (SNS)	
	Cohesion	MQ	Cohesion	MQ	Cohesion	MQ
Ant	0.000000	0.000000	0.000590	0.000283	0.023026	0.052496
AssertJ	0.000000	0.000000	0.000344	0.000699	0.025063	0.067731
Flume	0.000000	0.000000	0.000413	0.001359	0.025661	0.072067
Gson	0.000000	0.000000	0.002233	0.013831	0.060578	0.142030
JUnit	0.000000	0.000000	0.000560	0.001616	0.029550	0.097797
Nutch	0.000000	0.000000	0.000125	0.000695	0.019316	0.047951
PDFBox	0.000000	0.000000	0.000587	0.001112	0.028475	0.086138
Pivot	0.000000	0.000000	0.000330	0.001147	0.023383	0.065127
Procyon	0.000000	0.000000	0.000042	0.000164	0.009813	0.049230
Proguard	0.000000	0.000000	0.004786	0.001588	0.083427	0.140490
All	0.000000	0.000000	0.001001	0.000866	0.032829	0.082105

developers simply allocate classes to packages without any regard for the cohesion/coupling as captured by the chosen metrics. If this Null Hypothesis holds, then there is simply no evidence to suggest that developers care about cohesion or coupling. In such a situation, any attempt to optimise either raw cohesion or MQ, using search based or other techniques, would be unlikely to be viewed as beneficial by developers.

In order to test H_0, a random distribution of class allocations was performed for each release of each system. A random allocation for a given release is performed by randomly allocating its set of classes to packages. The probability of a class to be allocated to a certain package is uniform. One million random class allocations were performed for each release of each system, thereby forming a sample of the space of all possible allocations of classes to packages.

Since we have 233 different releases of the 10 systems under investigation, this means that the experiment conducted to answer research question RQ1.1 involves the computation of 233 million randomly constructed modularisations. One (very obvious) advantage of our approach, from an inferential statistical point of view, is the ability to work with such a large sample. This large sample size enable us to produce precise assessments of the corresponding p-values.

The first two columns of Table 3 present the results of this analysis for each system under consideration, for raw cohesion and MQ, respectively. Raw coupling is simply the obverse of raw cohesion so, for brevity, we report only the values for raw cohesion. The entries in these columns indicate the percentage of Random modularisations (over all releases) that achieve cohesion (or MQ) values that are equal to or greater than those achieved by the developers' implementation. As can be seen from these columns, not one of the 233 million randomly constructed modularisations produce a cohesion or MQ value equal to or greater than that achieved by the developers.

We can safely reject the Null Hypothesis H_0, and claim that raw cohesion and MQ values of open source software systems do not follow a random distribution. This result does not provide evidence that developers actually *care* about these metrics (it could simply be that they care about some other property that happens to correlate to significantly higher cohesion and MQ values). Nevertheless, these findings do strongly reject the claim that their allocation of classes to packages *fails to respect* modularity measurements; an obvious, yet important "sanity check" result that has not hitherto been reported upon in the literature on search based modularisation, despite the large body of previous work that use these metrics to guide modular optimisation.

Our Null Hypothesis was based on purely random allocation of classes to packages, so the rejection of such a 'weak' Null Hypothesis can provide only a 'weak sanity check' on the intuition that developers' modularisation structure respects modularity measurements. This last observation motivates the next two research questions, which seek to set a stronger baseline comparison, against which the developers' modularisation structure is compared.

RQ1.2: How does the developers' modularisation structure compare to randomly identified k–neighbour modularisations? The k–Random Neighbourhood Search (kRNS) searches a randomly selected sample of solutions in the "k–neighbourhood" of the solution implemented by the developers, as defined in Section 2.2. For this investigation, we use a value k equal to the number of classes in the systems. Therefore, kRNS proceeds by randomly selecting a subset of the classes in the system and randomly moving each of these classes to another package, to produce a single element of the sample. This process is repeated, using a freshly selected subset of classes on each occasion, to produce a sample of solutions from the k–neighbourhood. In our case, as with the previous experiment, we repeat this process 1 million times, for each release.

The third and fourth columns of Table 3 present the percentage of kRNS results that produce equal or higher cohesion and MQ values than those for the developers' modularisation. Consider `Flume`, for example. For all its releases, 0.000413% of k–neighbours found by kRNS had higher cohesion than the original solution, while 0.001359% of the k–neighbours had higher MQ. As can be seen from Table 3, over all the 10 systems, 0.004786% and 0.013831% are the highest number of cohesion-improved and MQ-improved modularisations found by the kRNS approach, respectively. Indeed, at the 0.01 α level, we would still reject the (strengthened) Null Hypothesis that "Developers simply pick an arbitrary re-allocation of classes to packages within the neighbourhood of the current solution, when producing a new version of the system".

However, it can also be observed that, for every system, there does exist a member of the k–neighbourhood that enjoys a *higher* cohesion and/or *higher* MQ than that pertaining to the modularisation implemented by the system's developers. These results for RQ1.2 therefore provide a deeper insight than was possible from the purely random search used to answer RQ1.1. They show that, while modular structure tends to respect structural cohesion and coupling, developers, nevertheless, do not produce an optimal solution; a random search within the wider neighbourhood of the developers' solutions can improve the modularity in each and all the releases. Furthermore, one can observe that in the k–neighbourhood of all systems under study, the number of cohesion-improved solutions is different from the number of MQ-improved solutions. We will return to a deeper investigation of these observed differences later.

We now turn to a more systematic investigation of the neighbourhood. Clearly, for a systematic investigation of all k–neighbours, the computational cost rises exponentially (in k), and, in the limit, as k tends to the number of classes in the system, the systematic investigation tends to an exhaustive enumeration of all possible modularisations of the systems under investigation. This is clearly infeasible [34]. Indeed, avoiding such an exponential explosion was our motivation for sampling from the overall k–neighbourhood for RQ1.2. However, it is computationally feasible to consider the nearest of all neighbourhoods; the k–neighbourhood for $k = 1$, and this allows us to answer an interesting research question:

RQ1.3: What portion of modularisation allocations within the nearest possible neighbourhood ($k = 1$) would yield an improvement in modularity? The systematic enumeration of the 1–neighbourhood is interesting because this is the set of neighbouring modularisations that can be achieved by moving only a single class to another package in the system. As such, it is the single simplest (and least disruptive) possible modification to the modularisation structure chosen by the developers. In order to answer this research question, we took each class and moved it to each of the other packages which the class was not originally assigned by the developers. This yields $(M-1) \times N$ 1–neighbours (or 'nearest neighbours'), for a system consisting of M packages and N classes, thereby systematically covering the entire 1-neighbourhood. The results of this analysis are presented in the final columns of Table 3.

As can be seen, for each system investigated, there are a nontrivial number of such single moves that can improve both the cohesion and the MQ score. Nevertheless, the solutions chosen by the developers are better than at least 96.7% and 91.7% of the whole 1–neighbourhood, for cohesion and MQ, respectively. This provides evidence for a strong developer preference for structures that respect modularity metrics. Moreover, as observed in RQ1.2, the number of cohesion-improved solutions is different from the number of MQ-improved solutions. In fact, all systems presented more MQ-improved solutions than cohesion-improved solutions in the 1–neighbourhood.

Overall, as an answer to RQ1, we conclude that there is strong evidence to suggest that the developers' allocation of classes to packages does respect structural cohesion/coupling, as assessed by the metrics of raw cohesion and MQ. Furthermore, there is equally strong evidence that the developers' allocation of classes could be improved, possibly with relatively little disruption to the system's modular structure, since there do always exist nearest neighbour modularisations that enjoy better modular structure. This is interesting and important for work on automated software modularisation, since these metrics (MQ in particular) are widely used fitness functions to guide such work on automated modularisation.

As observed in RQ1.2 and RQ1.3, when searching both the k–neighbourhood and the 1–neighbourhood of the systems under study, raw cohesion and MQ sometimes do not agree in assessing the modular structure of different solutions. We define "agreeing solutions" to be those modularisations that improve on the developers implementation according to both cohesion and MQ, while "disagreeing solutions" are those that have either higher cohesion or higher MQ, but not both.

Interestingly, and importantly for search based modularisation research, we observe that, on average, 83.04% of the neighbourhood solutions that present an improvement on the original implementation in either cohesion or MQ are "disagreeing" and only 16.96% are "agreeing". Moreover, for the disagreeing solutions, 67.61% present higher MQ than the original implementation but lower cohesion, and 32.39% present higher cohesion than the original implementation but lower MQ. These fidings replicate previous results [41]. In addition, we have presented evidence that developers' solutions more closely respect raw cohesion than MQ. Since MQ is a popular metric for search based re-modularisation, it is interesting and important for the community to understand how this metric is related to raw cohesion, which is a more basic and intuitive assessment of modular structure. This observation motivates our next research question.

4.2 RQ2: What is the relationship between raw cohesion and the MQ metric?

We performed three different analyses to investigate the relationship between MQ and raw cohesion. Each of these analyses employ a different technique to search for better modularisations.
RQ2.1: What is the relationship between raw cohesion and MQ for the solutions identified in RQ1? RQ1 performed neighbourhood search in the developers' implemented solutions to find better allocations of classes to packages. For RQ2.1, all neighbour solutions that improved on the original developers' implementation in at least one of the metrics (raw cohesion/coupling and/or MQ) were considered for analysis, which, on average, represents 0.35% of the solutions found by the kRNS and SNS in RQ1.

Table 4: Cohesion, Coupling and MQ results with standard deviation for Neighbourhood, Bunch and Package-constrained searches for improved modularisations. Cohesion, Coupling and MQ entries denote the average difference between the optimised modularisation in comparison to the original developers' implementation. In addition, we report the Kendall-τ correlation coefficients between Cohesion and MQ results for each system.

Systems	Neighbourhood Search				Bunch				Package-constrained HC			
	Cohesion	Coupling	MQ	K-τ	Cohesion	Coupling	MQ	K-τ	Cohesion	Coupling	MQ	K-τ
Ant	0.423% ± 1.116%	-0.423% ± 1.116%	-1.907% ± 3.644%	-0.18*	-45.919% ± 2.279%	45.919% ± 2.279%	376.611% ± 0.378%	0.56*	30.063% ± 5.348%	-30.063% ± 5.348%	40.598% ± 1.078%	0.44*
AssertJ	-0.174% ± 1.066%	0.174% ± 1.066%	0.102% ± 1.130%	-0.27	-62.888% ± 1.772%	62.888% ± 1.772%	522.197% ± 0.293%	0.41*	0.714% ± 4.217%	-0.714% ± 4.217%	61.167% ± 1.816%	0.46*
Flume	-1.092% ± 2.273%	1.092% ± 2.273%	0.857% ± 0.594%	-0.22	-35.977% ± 2.100%	35.977% ± 2.100%	336.829 ± 0.594%	0.57*	19.379% ± 4.030%	-19.379% ± 4.030%	51.026% ± 1.232%	0.55
Gson	-5.269% ± 7.090%	5.269% ± 7.090%	2.934% ± 0.569%	0.08*	-55.559% ± 3.133%	55.559% ± 3.133%	584.924% ± 0.940%	0.45*	19.832% ± 5.988%	-19.832% ± 5.988%	97.978% ± 4.089%	0.64
JUnit	-0.449% ± 1.797%	0.449% ± 1.797%	0.292% ± 1.446%	-0.14*	-28.240% ± 2.900%	28.240% ± 2.900%	226.341% ± 0.998%	-0.37*	30.157% ± 4.850%	-30.157% ± 4.850%	66.530% ± 1.608%	0.48*
Nutch	-1.252% ± 1.977%	1.252% ± 1.977%	1.018% ± 0.414%	-0.27	-42.993% ± 2.015%	42.993% ± 2.015%	321.843% ± 0.420%	0.50*	17.564% ± 5.603%	-17.564% ± 5.603%	56.962% ± 0.746%	0.44*
PDFBox	-0.377% ± 1.398%	0.377% ± 1.398%	0.119% ± 1.537%	-0.18	-33.100% ± 3.595%	33.100% ± 3.595%	180.531% ± 0.350%	0.43*	31.387% ± 5.532%	-31.387% ± 5.532%	79.773% ± 0.979%	0.43*
Pivot	-1.258% ± 2.273%	1.258% ± 2.273%	0.519% ± 0.549%	-0.20	-43.776% ± 1.680%	43.776% ± 1.680%	266.257% ± 0.508%	0.63	3.927% ± 4.102%	-3.927% ± 4.102%	44.875% ± 1.155%	0.51*
Procyon	-0.064% ± 0.289%	0.064% ± 0.289%	0.102% ± 0.054%	-0.20	-71.726% ± 1.194%	71.726% ± 1.194%	408.408% ± 0.145%	0.57	-4.799% ± 4.312%	4.799% ± 4.312%	56.538% ± 0.634%	0.44*
Proguard	1.035% ± 1.435%	-1.035% ± 1.435%	-2.494% ± 3.805%	-0.10*	-42.355% ± 5.383%	42.355% ± 5.383%	295.561% ± 0.357%	0.48*	102.282% ± 10.911%	-102.282% ± 10.911%	86.169% ± 2.606%	0.46*
All	0.750% ± 2.071%	0.750% ± 2.071%	0.154% ± 1.374%	-0.19*	-46.253% ± 2.625	46.253% ± 2.625	351.950% ± 0.498%	0.49*	25.050% ± 5.489%	-25.050% ± 5.489%	64.161% ± 1.594%	0.46*

The first three columns of Table 4 present the average differences and standard deviation in cohesion, coupling and MQ for the neighbourhood search solutions, respectively. Consider the `Gson` system, for example. The neighbourhood solutions offer an average difference in cohesion, coupling and MQ of -5.269%, 5.269% and 2.934%, respectively. These values indicate that within the set of `Gson` neighbourhood solutions that improve on the original implementation in at least one of the metrics, the average differences in cohesion, coupling and MQ are -5.269%, 5.269% and 2.934%, respectively.

One should notice that, as mentioned before, cohesion and coupling differences are the obverse of each other. Since we are considering cohesion to be the number of dependencies within packages and coupling to be the number of dependencies between packages, in the case of a certain dependency being moved from between packages to inside a package, cohesion will increase and coupling will decrease. Similarly, a dependency that is moved from inside a package to between packages is going to decrease cohesion and increase coupling. Therefore, for brevity purposes, only cohesion values will be reported and discussed in the rest of the paper.

We use correlation analysis to investigate more precisely the relationship between cohesion and MQ. For each release of each system, the non-parametric Kendall-τ correlation test was applied for the cohesion and MQ values of the neighbourhood solutions. The fourth column of Table 4 presents the correlation coefficient of each system, which is computed as the median coefficient of all releases of each system. An asterisk (*) decorates the coefficient entry when not all releases exhibit a significant coefficient at the 0.01 α level. Most coefficients range from -0.2 to 0.2, which suggests little or no correlation between cohesion and MQ for the neighbourhood solutions.

However, we must be careful to not over generalize this observation, because only simple local search procedures were employed to find the solutions that were considered in the analysis, and the search space covered by the neighbourhood solutions is small. This motivated our next research question, in which we apply more sophisticated search based approaches for software re-modularisation.

RQ2.2: What is the relationship between raw cohesion and MQ for solutions found

by widely used search based cohesion/coupling optimisation approaches? For this analysis, we use the Bunch tool [38]. Bunch is a tool for search based modularisation that implements a simple hill climbing approach. There are other more sophisticated techniques for search based modularisation, that may produce superior results [47] in terms of cohesion, coupling, and the MQ metric, but at far greater computational cost. We wish to investigate whether developers can use simple and fast search-based modularisation techniques to quickly produce alternative solutions that significantly improve on the developers' given modularisations, according to MQ.

We applied the Bunch optimisation tool to all releases. Since Bunch's hill climbing algorithm is a randomized search algorithm, we performed 30 executions of Bunch for each release. The 30 resulting cohesion, coupling and MQ values found by Bunch for each release were compared with the developer's implementation, and the results are presented in the fifth, sixth and seventh columns of Table 4, alongside the respective standard deviation.

As one can see, the Bunch tool is able to find modularisations with remarkable MQ improvement (of more than 500% for some systems). However, all these MQ-optimised solutions have lower cohesion values than the developer's original implementation. Such a surprising result can be explained by the design of the MQ metric. As one can see in the MQ definition in Section 2.2, the MQ score is composed of the sum of the scores of each package in the modularisation; so, solutions with more packages tend to have higher MQ values. In fact, the solutions found by Bunch have, on average, 493.11% more packages than the developers' implementation. As a result, fewer classes are allocated to each package, thereby creating several dependencies that cut across package boundaries. We will refer to this phenomena as the MQ's 'inflation effect'.

The Kendall-τ correlation test was also applied to measure the correlation between raw cohesion and MQ of the Bunch solutions. Apart from `JUnit`, all systems have a moderate positive correlation between cohesion and MQ, which is a surprising result given that all Bunch solutions had worse cohesion than the original implementation. First of all, one needs to keep in mind that this correlation was computed using the 30 Bunch solutions of each release of each system. A positive correlation, in this case, indicates that in spite of the fact that all cohesion values of Bunch solutions are worse than the developers' implementation, the solutions with higher MQ tend to have a higher cohesion too.

As an example, the scatter plot in Figure 2 presents the cohesion and MQ differences of the 30 solutions found by Bunch for `Pivot 2.0.2` in comparison to the original implementation. As one can see, all the 30 solutions have higher MQ than the developers' implementation, yet lower cohesion; however, the solutions with higher MQ tend to have a higher cohesion too, which elucidates the positive correlations between cohesion and MQ reported in Table 4.

After an analysis of the 30 Bunch solutions, we noticed that these solutions have similar number of packages, where 16 (out of 30) solutions have 58 packages, 10 solutions have 57 packages and 4 solutions have 59 packages. This suggests that for modularisations with similar numbers of packages, higher MQ values usually denote higher cohesion. These observations motivate our next research question, where we introduce and evaluate a package-constrained approach for search based software re-modularisation as an attempt to improve the modular structure of software systems as assessed by *both* cohesion and MQ.

RQ2.3: What is the relationship between raw cohesion and MQ for solutions found by a package-constrained search for improved cohesion/coupling? The MQ metric was originally designed to optimise the cohesion/coupling of software systems from scratch, without any previous information on the modular structure other than the dependencies between elements. However, when Bunch is applied to large-scale real world software systems, the 'inflation effect' induced by MQ may be undesirable. Because of this effect, new packages are created and existing classes are moved to these new packages, causing a (large) decrease in the system's

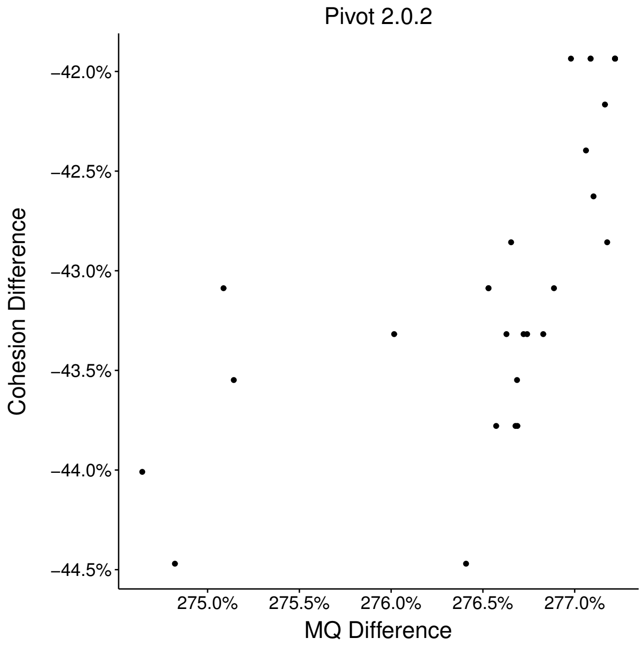

Figure 2: Cohesion and MQ differences for 30 modularisations found by Bunch for `Pivot 2.0.2` when compared to the original developers' implementation

cohesion.

We performed a longitudinal analysis of the allocation of classes to packages throughout releases as implemented by the developers themselves. We found no release (out of 233) where a new package was created and only existing classes were moved to the new package. Therefore, apart from decreasing the cohesion of the system, a Bunch re-modularisation might also be unrealistic because developers rarely create new packages to accommodate existing classes. This observation adds evidence to recent claims [13][11] against "Big Banlowere-modularisation approaches (i.e., a complete re-allocation of the system's classes in packages), where recent studies have used the original modular structure implemented by developers as a guide to find more suitable packages for certain classes [1][8].

Thus, in this research question, we introduce a package-constrained version of search based re-modularisation that maximises MQ and constrains the search algorithm to search only for modularisations with the same number of packages of the original developers' implementation. This way we avoid the creation of new packages, so that classes are only moved to packages that developers are already familiar with. Moreover, as suggested in RQ2.2, higher MQ values may lead to higher cohesion values for the same number of packages. Therefore, by maintaining the same number of packages as the original implementation, we might be able to optimise MQ and improve the overall cohesion of the system.

Hence, we re-implemented the hill climbing search approach of the Bunch tool including the number of packages as a constraint to the search. We executed the approach 30 times for each release of each system, and the average cohesion, coupling, MQ and standard deviations achieved by the package-constrained search are reported in the ninth, tenth and eleventh columns of Table 4, respectively.

Apart from the `Procyon` system, all package-constrained solutions yield improvements in *both* cohesion and MQ. On average, the cohesion of the systems under study was improved by 25.05%, and the biggest cohesion improvement was in `Proguard` with 102.28%. However, similar results were not achieved in some of the systems, such as `AssertJ` and `Pivot` that had small cohesion improvements, and `Procyon` that had an worse average cohesion than the original implementation. The results for these three systems indicate that in some cases, even in a package-constrained setting, MQ optimisation do not lead to better modularity, as assessed by raw cohesion. It might be possible that these systems already have a good cohesion, and cannot be further optimised. Nevertheless, the modularity of these systems need to be further investigated, so that accurate conclusions can be drawed.

The Kendall-τ correlation coefficients between cohesion and MQ for the package-constrained search are reported in the last column of Table 4. The moderate positive coefficients that can be seen for the package-constrained search resemble the coefficients computed for the Bunch solutions. These results reinforce the observation that MQ can indeed guide the search towards solutions with better cohesion when the search is package-constrained. This is an important finding for the search based re-modularisation community.

As an answer to RQ2, we showed that raw cohesion and MQ do not commonly agree in assessing the modularity of software systems. We noticed that this is mainly due to the 'inflation effect' of MQ, where Bunch creates an average of 493.11% new packages in the system, which decreases the cohesion when compared to the original developers' implementation. However, we observed that in solutions with similar number of packages, MQ and cohesion have a moderate positive correlation, which mainly led us to introduce a new package-constrained search as an attempt to mitigate MQ's 'inflation effect'. In general, package-constrained automated modularisation was able to improve the cohesion of the systems under study by 25.05% without creating new packages.

Considering the results presented in RQ1 and RQ2, we showed that developers have some degree of respect for structural measurements of cohesion and coupling as the original solutions are better than the ones found by random and neighbourhood search; however, optimal values of cohesion and coupling might not be pursued since developers' solutions are worse than the ones found by the hill climbing search. This observation endorses a recent study [11] that compared developers' modularisations of open source systems with alternatives found by multiobjective search for cohesion/coupling improvement. Even though the empirical studies presented in this and in the related paper [11] use different quality metrics, different search procedures and different software systems, they complement each other by presenting evidence that developers do respect structural measurements of cohesion and coupling, but optimisation of these properties is not sought.

4.3 RQ3: What is the disruption caused by search based approaches for optimising software modularisation?

The previous section showed that search-based algorithms can be used to optimise the trade-off between cohesion and coupling in open source software systems. In fact, candidate solutions in the package-constrained search usually present improved modular structure, as measured by both cohesion and MQ metrics. This raises the obvious question: if systems can be optimised for modularity, and there is evidence that systems respect structural measurements, then why do developers implement solutions with a sub-optimal modular structure?

One answer to this question lies in the potential size and complexity of the search space; humans have been shown, repeatedly, to be sub optimal, in their ability to find solutions to SBSE problems such as this [45][55]. However, it is also important to explore another possibility: perhaps the improvement in modular structure achieved using SBSE comes with a price of

Table 5: Disruption to the modular structure caused by Bunch and Package-constrained search based approaches for modularisation improvement. We report the mean disruption caused by the 30 executions of each search approach. In addition, we report the disruption caused by the best solutions (out of the 30), as assessed by Cohesion and MQ. Each entry in the table is an average over all releases of the system.

Systems	Bunch			Package-constrained		
	Mean	Best - Cohesion	Best - MQ	Mean	Best - Cohesion	Best - MQ
Ant	82.70%	80.63%	81.70%	57.86%	53.50%	55.73%
AssertJ	90.11%	89.66%	90.11%	59.02%	55.51%	56.54%
Flume	79.90%	79.09%	79.19%	62.20%	57.57%	58.16%
Gson	88.49%	85.22%	87.78%	56.04%	48.72%	51.16%
JUnit	69.87%	68.24%	69.23%	52.11%	49.53%	50.12%
Nutch	77.22%	75.92%	76.45%	61.91%	60.30%	60.15%
PDFBox	66.78%	64.18%	65.96%	53.12%	51.49%	51.74%
Pivot	79.26%	78.39%	78.32%	60.47%	56.36%	56.36%
Procyon	85.98%	84.39%	85.23%	56.94%	53.94%	54.88%
Proguard	83.66%	82.02%	82.74%	58.59%	56.44%	58.24%
All	80.39%	78.77%	79.67%	57.82%	54.33%	55.30%

significant disruption to the existing modularity. There is evidence in the literature [57] that developers are reluctant to change the structure of systems, choosing instead, to retain the familiar structure rather than move to an improved version. Therefore, we turn our attention to assessing the degree of disruption that would result from an improvement performed by the SBSE approaches to automated software re-modularisation presented in RQ2. For this analysis, we use the DisMoJo metric, which is formally defined in Section 2.3.

RQ3.1: What is the disruption caused by widely used search based tools for automated software modularisation? In this research question we want to assess how much disruption developers would have to endure when using a widely used tool for modularity optimisation. For this analysis, the solutions found by the Bunch tool in RQ2.2 will be considered. Each of the 30 solutions found by Bunch for each release of each system are compared to the original developers' implementation. The average disruption caused by Bunch, as assessed by DisMoJo, for each system under study is presented in the first column of Table 5.

Considering all systems, the average disruption that developers would need to endure in order to optimise the modular structure using Bunch is 80.39%. This observation provides evidence that even though existing SBSE techiniques can improve modular structure, the high disruption caused to the original system might inhibit wider industrial uptake of search based re-modularisation.

After an inspection of all 30 Bunch solutions of each release of each system, we collected the solutions with higher cohesion and MQ, and reported the average disruption caused by these best solutions over all releases. The disruption caused by the best cohesion and best MQ solutions found by Bunch are presented on the second and third columns of Table 5, respectively.

Since the modularity optimisation process consists in moving classes around packages, one might expect that the most optimised solutions will also be the most disruptive ones. However, as can be seen from Table 5, the disruption caused by the best cohesion and MQ solutions are actually *smaller* than the average disruption. This counterintuitive observation suggests that, in the scenario where developers are willing to endure considerable disruption to optimise their system modular structure, it is possible to find solutions with less modifications than expected.

RQ3.2: What is the disruption caused by the package-constrained search based approach for automated software modularisation? The package-constrained search approach for software modularisation was introduced in RQ2.3 as an alternative to mitigate the 'inflation effect' of the Bunch tool. The average disruption caused by the package-constrained search is presented in the fourth column of Table 5.

As expected, the disruption caused by the package-constrained search is smaller (57.82%), but it still denotes a high number of modifications to the original system in order to optimise the modular structure.

The average disruption caused by the best cohesion and MQ solutions for the package-constrained search are presented in the last two columns of Table 5, respectively. Similarly to the Bunch results, the disruption of the solutions with best modular structure is smaller than the average disruption caused by the 30 executions for each release.

In general, the disruption caused by the package-constrained optimisation approach is smaller than the disruption caused by the Bunch tool. An unexpected observation from these analyses was that solutions with the best modular structure, as assessed by both cohesion and MQ, presented smaller disruption than the average.

As an answer to RQ3, the disruption caused by search based approaches to automated re-modularisation is high. The results found in this paper complement a recent disruption analysis performed by Candela et al. [11], where despite using different optimisation algorithms, different cohesion/coupling metrics and different software systems, both studies showed that search based modularisation is highly disruptive. We conjecture that such disruption inhibit industrial uptake of these techniques.

4.4 RQ4: Can multiobjective search find allocations of classes to packages with a good trade-off between modularity improvement and disruption of the original modular structure?

Summarizing the findings of RQ1-3: open source software systems respect structural measurements of cohesion and coupling (RQ1), but although search based techniques can substantially improve the systems' modular structure (RQ2), these techniques tend to dramatically disrupt the original developers' implementations (RQ3).

Motivated by these findings, we introduce a multiobjective evolutionary search based approach to find candidate modularisations with a good trade-off between modular improvement and disruption. Our intuition is that since the systems under study exhibit considerable respect for structural measurements of cohesion and coupling, developers might be willing to improve their systems' modular structure when the changes required for improvement lie within an acceptable range.

In order to carry out this analysis, we propose two different multiobjective experiments, each of which uses different search strategies; therefore, providing different insights on how multiobjective search can be used to improve software modularity, while taking disruption into account.

For all multiobjective experiments we use the Two-Archive Genetic Algorithm [48], which was demonstrated to perform well in a previous multiobjective investigation of automated software modularisation [47]. The Two-Archive GA settings are mostly based on this earlier work [47], and are the same for all experiments: The population size is set to N, where N is the number of classes in the system. Single point crossover is employed with a 0.8 probability when $N < 100$, and 1.0 probability otherwise. Swap mutation is performed with a probability of $0.004 \log_2^N$. Parents are selected by tournament, with a tournament size of 2. In addition, the probability of selecting parents from the convergence archive is 0.5, and the size of the archives is limited to 100 individuals. Finally, the number of generations is set to $50N$.

RQ4.1: What is the trade-off between modularity improvement and disruption for the package-free search? The first multiobjective experiment is concerned with the widely used [38][47] optimisation approach to improve software modularity where the search algorithm has no constraints on the number of packages it can create. We call this search strategy "package-free". In order to identify the trade-off between modularity improvement and disruption, the search algorithm attempts to maximise MQ and minimise DisMoJo. In addition, we measure the raw cohesion of the solutions found by the multiobjective search.

In RQ2.2 we used the Bunch tool to find MQ-optimised solutions for each release of each system under study; therefore, the solutions found by Bunch can be used as starting points (seeds) for the multiobjective algorithm in its search for solutions with high MQ value. Similarly, the original developers' implementation of each release is also used to seed the Two-Archive GA.

Figure 3 presents some of the pareto fronts found for the package-free multiobjective execution. We selected one release as a representative of each system to be discussed in this paper. However, we make all results available on the paper's complementary web page[3]. As one can see, the results for the different systems are considerably similar, where all releases present a clear and almost constant trade-off between MQ improvement and DisMoJo, which is an expected behavior because MQ improvement is achieved by adding new packages; therefore, leading to large scale disruption.

RQ2 showed that an improvement in MQ does not necessarily indicate an improvement in the raw cohesion of the system; therefore, we also measured the raw cohesion of all different modularisations found by the multiobjective search that targets MQ improvement. When considering all the package-free MQ-optimised modularisations in the pareto fronts, most of them have a cohesion value that is *worse* than the original developers' implementation. These results add evidence to the observation in RQ2, that MQ-optimised solutions may decrease the cohesion of the original system. In fact, when considering the pareto fronts computed for `Nutch`, `PDFBox` and `Proguard`, for example, *all* the modularisations are worse than the original system in terms of raw cohesion.

RQ4.2: What is the trade-off between modularity improvement and disruption for the package-constrained search? This second multiobjective experiment is concerned with the automated software re-modularisation approach proposed in RQ2.3, where the search algorithm is package-constrained. Similarly to RQ4.1, the multiobjective search tries to maximise MQ and minimise DisMoJo; however, the search algorithm is constrained to the same number of packages as those in the original developers' implementation.

For this research question, the Two-Archive GA is seeded with the original system (as in RQ4.1) and the MQ-optimised solutions found by our package-constrained implementation of the hill climbing algorithm used by the Bunch tool (see RQ2.3). Figure 4 presents the pareto fronts found by the package-constrained multiobjective search.

Similarly to RQ4.1, there is a clear trade-off between MQ and DisMoJo; however, the pareto front structure is different: we observe a larger number of gaps and 'knee points' in the package-constrained pareto fronts than in the package-free ones.

The cohesion improvements achieved by all modularisations in the the package-constrained pareto fronts were also measured. In most of the systems, the number of MQ-optimised modularisations with better cohesion than the original implementation is noticeably bigger than in RQ4.1. Moreover, for almost all the systems, it is possible to find modularisations with a considerable improvement in cohesion and yet a relatively small disruption. This is a very positive outcome; although re-modularisation approaches may be too disruptive, multiobjective search migth be able to find solutions with useful compromises between modular improvement and disruption.

As one can notice in the package-constrained pareto fronts in Figure 4, sometimes the modularisation found by the hill climbing package-constrained search is not part of the pareto front. Considering Ant, Flume and JUnit, for example, the hill climbing modularisation has higher dis-

[3]`http://www0.cs.ucl.ac.uk/staff/mpaixao/cohCop/index.html` (Please note the url will be activated upon peer-reviwed publication)

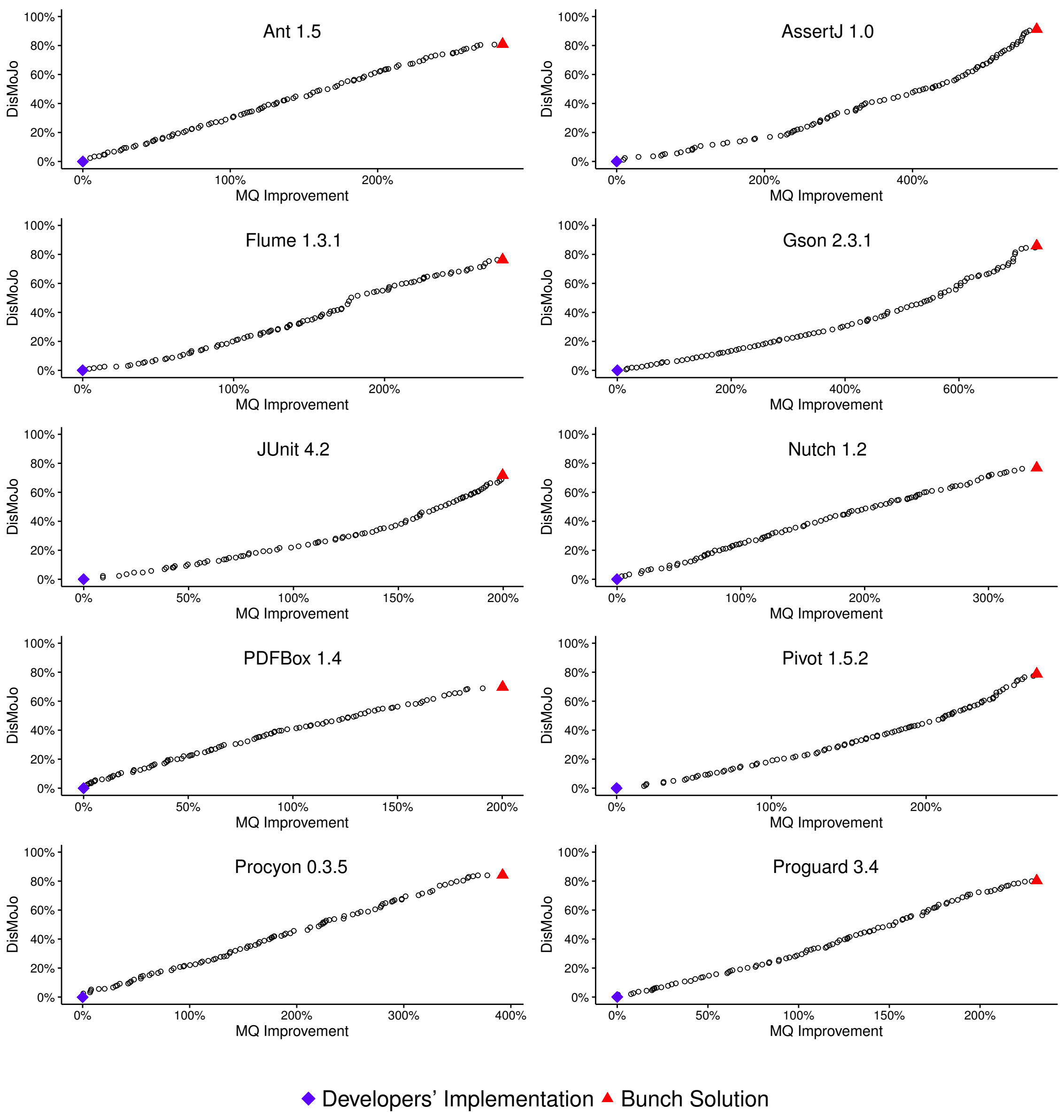

Figure 3: Pareto fronts reporting the trade-off between MQ and DisMoJo for the package-free multiobjective search

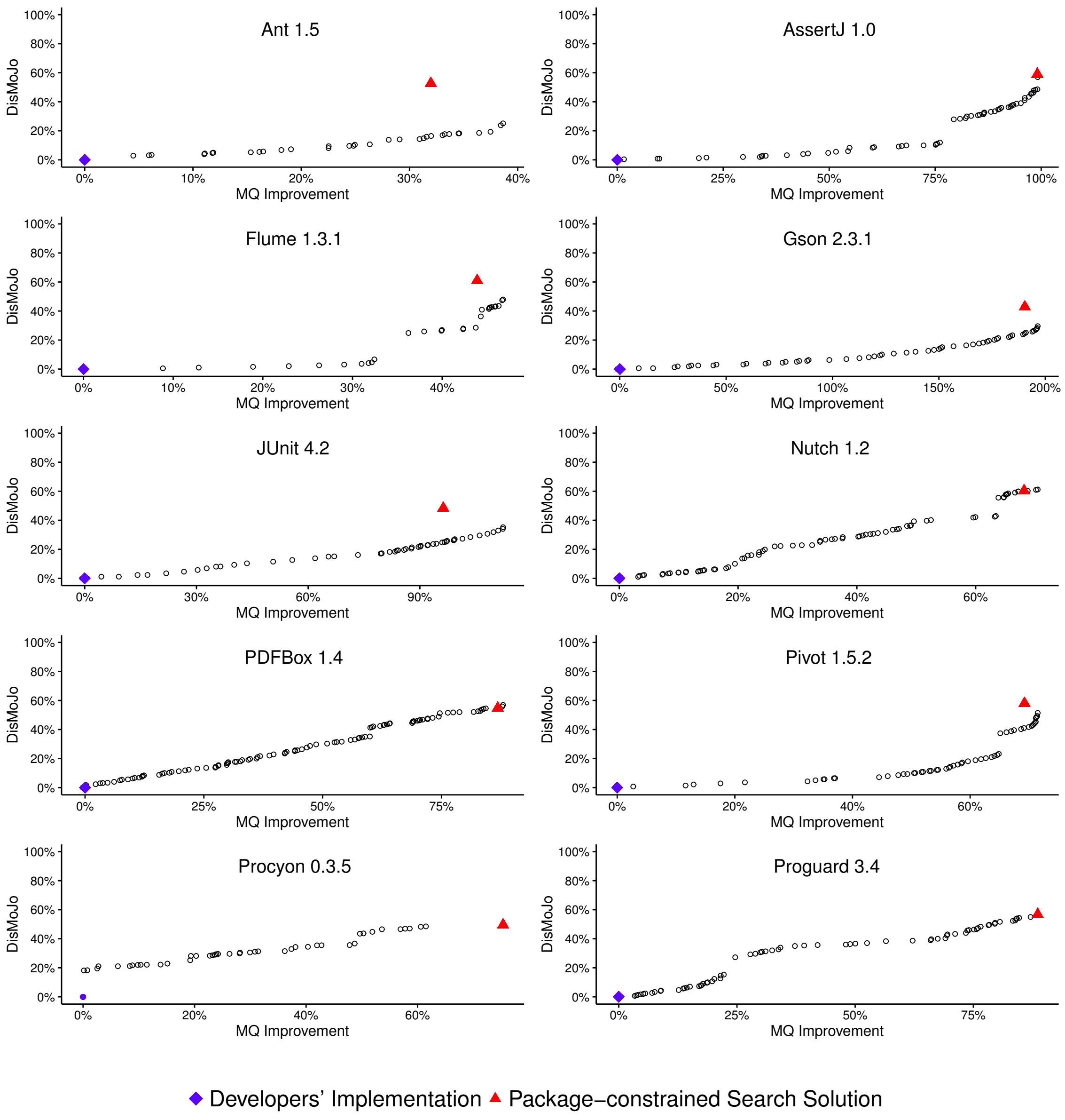

Figure 4: Pareto fronts reporting the trade-off between MQ and DisMoJo for the package-constrained multiobjective search

ruption and lower MQ than other solutions in the pareto front. Differently, in the package-free pareto fronts in Figure 3, the Bunch solution is always the one with highest MQ in all pareto fronts.

This might be possible because even though we followed what was described in the PhD thesis [34] of one of Bunch's creators, the Bunch tool has continued to be improved over the years [38][39], so that our implementation might only be able to find local optima modularisations. According to Table 4, the standard deviation of our implementation of the hill climbing search is higher than Bunch's, which may be an indicator of the conjecture above. However, it might also be the case that the MQ search space of the package-constrained envinronment is different than the package-free one, where solutions with bigger MQ improvement can be found on the neighbourhood of solutions with small disruption. Nevertheless, this is an interesting finding that needs further investigation.

The main goal of RQ4 (and Figures 3 and 4) is to illustrate the trade-off between improvement in modular structure and disruption to the original implementation that can be achieved with multiobjective search. The state of the art techniques for automated software modularisation, both single [38] and multi [47] objective, are mostly concerned with modularity improvement, which we know usually causes a large disruption to the original implementation (see RQ3). Previously, developers who would like to optimise the modular structure of their systems using search based approaches would have two choices: improve the system as much as possible and thereby considerably change the original structure, or keep the original implementation and do not perform any improvement. With the multiobjective approach proposed to answer RQ4.1 and RQ4.2, developers would have a wider range of options.

The analyses performed in RQ4 took into consideration all solutions in the computed pareto fronts, providing general insights on the shape of the fronts and on the quality of the solutions within the fronts. In RQ5 we show how developers can pick a particular modularisation from the pareto fronts according to their needs and constraints.

4.5 RQ5: What is the modularity improvement provided by the multiobjective search for acceptable disruption levels?

In RQ4 we showed that the proposed multiobjective search can find solutions that improve the modularity of the original developers' implementation, as assessed by MQ and cohesion, especially for package-constrained search. However, we did not discuss how developers can use the proposed multiobjective approach. We believe that the multiobjective optimisation of modularity and disruption can be used by developers at different moments during the software lifecycle, depending on how much disruption they are willing to endure in order to achieve modularity improvement.

As an example, consider the scenario where developers are planning amajor release of the software system. Since it is a major release, the system will possibly undergo large changes to accommodate the new features. In this case, developers can take advantage of the fact the system is going to undergo substantial change, and perform large refactorings to improve the modular structure. On the other hand, in minor or bug-fixing releases, developers may be less willing to change modular structure, therefore, favouring smaller changes. However, this 'acceptable disruption' level is not obvious.

Therefore, in this research question we introduce three different methods to estimate the 'acceptable' level of modularity disruption that can be sustained by developers in order to obtain modularity improvement. All methods are based on a longitudinal analysis of the developers' implementations of each release of each system under study. Later, we show how these different 'acceptable' levels of disruption can be used to select solutions from the pareto fronts found by the multiobjective search approach.

RQ5.1: What is the longitudinal modular disruption introduced by developers? As a software system evolves, new features are added, changed or removed; as such, the modular structure of the system needs to change in order to cope with the new requirements and demands. Therefore, the modular structure of a software system is constantly disrupted by its developers during the system's lifetime, in which we call the 'natural disruption' of the system. Although the 'acceptable disruption' level that developers are willing to endure to improve the modularity is difficult to measure, we argue that the 'natural disruption' level that developers introduced during the system evolution is a good proxy. Thus, we introduce three different methods to assess the 'natural disruption' of the systems under study, each of which are used as an estimation of the 'acceptable' level of disruption.

The first two methods use the DisMoJo metric in a different way than used in RQ3 and RQ4. *DisMoJo*(A, B), as defined in Section 2.3, is used to measure the disrution between A and B when both modularisations are composed by the same set of classes. Therefore, since classes can be added or removed between two different releases of the same system, DisMoJo cannot be used to measure the disruption between releases of the same system. In order to provide a lower and an upper bound of the disruption between releases, we introduce Intersection DisMoJo and Union DisMoJo, respectively.

Consider two subsequent releases A and B of the same system. Intersection DisMoJo is computed by considering only the subset of classes that belongs to both A and B. We say this is a lower bound disruption between releases because it considers the minimum number of classes that can be moved between releases. Accordingly, Union DisMoJo is computed by aggregating all classes that belong to both A and B, where classes that belong to A but do not belong to B, and vice-versa, are allocated to a separate package. This is an upper bound of disruption because all possible classes that can be moved, added or deleted between A and B are taken into account.

Finally, our third method to assess the 'natural disruption' of a software system is based on the analysis of the proportional increase in the number of classes over releases. As the system evolves, the number of classes added in each release is a simple and straightforward way to asses how much of the modular structure changes during the system evolution.

Each of the three methods to assess the 'natural disruption' described above was computed for each pair of subsequent releases of the systems under study, and the results are presented in Table 6. For each system we report the minimum, maximum, median and mean values for each method. This way we can assess what is the biggest and smallest disruption levels each system has undergone during its lifecycle, and also what is the average disruption developers are used to introduce during systems' evolution.

As one can see, the minimum 'natural disruption' for all systems, according to all three estimation methods, is 0.00%. This means that for all systems, there is at least one pair of subsequent releases that has the same modular structure. The Itersection DisMoJo values are the smallest for all systems (as expected), and for `Flume` and `Procyon`, Intersection DisMoJo is always 0.00%. These results add evidence to the observation in RQ2.3 that existing classes rarely move between packages. Furthermore, all disruption values reported by both Intersection and Union DisMoJo lie within the range of the proportional addition of classes, which is a straightforward way for developers to understand the 'natural disruption'.

RQ5.2: How much modularity improvement can be achieved within lower and upper bounds of 'acceptable' disruption? In this analysis, the 'natural disruption' levels computed in RQ5.1 are used as proxies for the 'acceptable' level of disruption that developers would be willing to endure in order improve the modular structure of their systems. As previously mentioned, developers have different 'acceptance' levels at different moments of the software lifetime; there-

Table 6: 'Natural disruption' levels caused by developers during system's evolution, as assessed by three differement methods. Intersection DisMoJo computes the DisMoJo metric considering the intersection of classes between two subsequent releases, while Union DisMoJo computes the DisMoJo metric considering all classes of two subsequent releases. The proportional addition of classes accounts for the proportional increase in the number of classes between two subsequent releases of the same system. Each method was used to compute the 'natural disruption' of each release of each system, and we report the minimum, maximum, median and mean results for each system.

Systems	Intersection DisMoJo				Union DisMoJo				Proportional Addition of Classes			
	Min	Max	Median	Mean	Min	Max	Median	Mean	Min	Max	Median	Mean
Ant	0.00%	2.12 %	0.00%	0.07%	0.00%	36.92%	0.59%	6.48%	0.00%	72.54%	0.05%	11.05%
AssertJ	0.00%	0.79 %	0.00%	0.12%	0.00%	19.34%	3.53%	4.68%	0.00%	47.89%	3.78%	8.50%
Flume	0.00%	0.00 %	0.00%	0.00%	0.00%	36.55%	1.98%	8.60%	0.00%	59.59%	2.37%	14.52%
Gson	0.00%	16.45%	0.00%	1.23%	0.00%	38.71%	6.76%	10.60%	0.00%	64.70%	8.31%	14.09%
JUnit	0.00%	9.09 %	0.00%	0.79%	0.00%	41.89%	2.92%	9.34%	0.00%	196.93%	4.34%	23.30%
Nutch	0.00%	0.43 %	0.00%	0.03%	0.00%	35.15%	1.82%	5.67%	0.00%	51.77%	2.41%	7.09%
PDFBox	0.00%	2.06 %	0.00%	0.10%	0.00%	26.72%	1.57%	5.41%	0.00%	35.00%	2.54%	8.24%
Pivot	0.00%	0.79 %	0.00%	0.07%	0.00%	14.53%	1.14%	5.77%	0.00%	32.03%	1.08%	9.47%
Procyon	0.00%	0.00 %	0.00%	0.00%	0.00%	3.78 %	0.02%	0.54%	0.00%	3.03%	0.38%	0.63%
Proguard	0.00%	11.48%	0.00%	0.58%	0.00%	56.36%	1.28%	5.32%	0.00%	75.37%	1.55%	6.56%

fore, we report the modularity improvement that can be achieved at lower and upper bounds of the 'acceptable disruption'.

The lower bound denotes the smallest greater than zero disruption level we could ascribe from the average disruption caused by developers over the period of evolution of the systems studied. This is a reasonable lower bound because it is chosen to be the lowest possible value (median or mean, using either intersected or unioned DisMoJo) over all releases, for each system. If the developers are prepared to tolerate this amount of disruption during the system's development, on average, then it is not unreasonable that they might allow this amount of disruption when it can occasionally improve the modular structure.

The upper bound denotes the largest possible disruption value we can ascribe from the disruption caused by developers in any release of the system studied (using either intersected or unioned DisMoJo). This is a reasonable upper bound because we know that there does exist a release of the software that causes this level of disruption, and therefore we know that it was, at least on one occasion, tolerated by the developers.

Therefore, for each system, we identified the lower and upper bounds of 'acceptable disruption' as described above; then, we selected modularisations from the pareto fronts computed in RQ4 according to these lower and upper bounds. Consider the `Ant` system, for example. The lower and upper bounds for 'acceptable disruption' were identified as 0.07% and 36.92%, respectively. For each release of `Ant` we selected the solutions with best cohesion and MQ improvements found by both package-free and package-constrained search approaches that have a DisMoJo value equal or smaller the lower and upper bounds of 'acceptable disruption'. Results for all systems under study are reported in Table 7.

As can be seen from the table, the modularity improvements achieved within the lower bound disruption, for both package-free and package-constrained are small. In fact, for most of the systems, neither package-free nor package-constrained has found any improvement in neither cohesion nor MQ within the lower bound disruption. However, for some software systems it is possible to have modularity improvements even considering a lower bound disruption, such as `Gson`, where package-constrained search found a 4.70% cohesion improvement within the minimum 'acceptable disruption' level.

As expected, modular improvements within the upper bound disruption levels are the biggest

Table 7: Modularity improvement, as assessed by cohesion and MQ, that can be achieved within the lower and upper bounds of the 'acceptable disruption' level.

Systems	Package-Free				Package-Constrained			
	Lower Bound		Upper Bound		Lower Bound		Upper Bound	
	Coh	MQ	Coh	MQ	Coh	MQ	Coh	MQ
Ant	0.00%	0.00%	1.24%	148.00%	0.00%	0.00%	7.66%	35.21%
AssertJ	0.00%	0.00%	0.52%	157.01%	0.00%	0.00%	4.73%	41.02%
Flume	0.00%	7.82%	0.54%	168.65%	1.25%	15.21%	21.35%	59.73%
Gson	1.28%	8.57%	8.97%	365.62%	4.70%	17.09%	40.98%	121.53%
JUnit	0.10%	0.00%	3.16%	148.37%	0.76%	2.22%	22.21%	49.09%
Nutch	0.00%	0.00%	0.00%	159.76%	0.00%	0.00%	1.89%	70.03%
PDFBox	0.00%	0.00%	2.73%	66.65%	0.00%	0.00%	6.61%	45.47%
Pivot	0.00%	0.00%	0.46%	83.23%	0.00%	0.00%	9.80%	45.44%
Procyon	0.00%	0.00%	0.00%	8.32%	0.00%	0.00%	0.00%	0.00%
Proguard	0.00%	0.00%	6.19%	200.29%	0.21%	0.00%	110.00%	94.05%
All	0.13%	1.63%	2.38%	150.28%	0.69%	3.45%	22.52%	55.75%

for all systems. When considering the biggest disruption level the systems have already undergone, package-constrained search was able to find modularisations with considerable cohesion improvements, such as 40.98% and 110.00% for `Gson` and `Proguard`, respectively.

As an answer to RQ5, multiobjective search can find modularisations with improved modular structure, as assessed by both cohesion and MQ, even within lower and upper bounds of disruption introduced by developers between releases.

5 Qualitative Analysis

In this section we select one of the systems we studied in our empirical study and describe with more details some of the results we achieved throughout our research questions. Table 8 reports detailed results for each release of JUnit, including the cohesion and MQ values of the original developers' implementations and the results achieved by Bunch, package-constrained and multiobjective search. Finally, we also report the natural disruption between all releases of the system. We have chosen JUnit because it presented a wide range of modularity variation during its releases, enabling us to illustrate different aspects of the studies we performed.

The second and third columns of Table 8 report the cohesion and MQ values of the original modularisation implemented by JUnit developers for the 20 subsequent releases we collected. Both cohesion and MQ metrics are affected by the size of the system, where a higher number of classes and dependencies usually leads to a higher cohesion and MQ; therefore, different releases of JUnit cannot be compared by neither MQ nor cohesion. However, the techniques described in this paper optimise the modular structure for each particular release, so that comparisons 'within release' are valid.

The average and standard deviation values of cohesion and MQ for Bunch search are reported in the fourth and fifth columns of the table, while the results for package-constrained search are reported in the sixth and seventh columns. As discussed in RQ2, the cohesion of the Bunch optimised modularisations is always lower than their original counterparts, even though the MQ is considerably higher. Differently, all package-constrained solutions are able to improve upon the original implementation in both cohesion and MQ.

In release 4.2, for example, the average Bunch solution has a cohesion value of 107 while the developers' implementation has a cohesion value of 164, which corresponds to a difference of -34.94%. On the other hand, the average package-constrained modularisation for release 4.2 has a cohesion value of 214, which represents an improvement of 30.16% over the original modularisation. This particular case elucidates the MQ 'inflation effect' discussed in RQ2, showing how

Table 8: Detailed results for all releases of JUnit. For each release, we report the raw cohesion and MQ values for the original developers' implementation and the results achieved by both Bunch and Package-constrained search. In addition, we report the lower and upper bounds of the natural disruption between releases, computed by Intersection and Union DismoJo, respectively. Finally we report the cohesion and MQ results achieved by the proposed multiobjective approach to maximise modularity improvement and minimise disruption, where we use the natural disruption of each release to pick a solution from the pareto front.

Release	Original Implementation		Bunch		Package-constrained HC		Multiobjective Package-constrained				Natural Disruption	
							Lower Bound		Upper Bound			
	Cohe-sion	MQ	Cohesion	MQ	Cohesion	MQ	Cohesion	MQ	Cohesion	MQ	Lower Bound	Upper Bound
3.7	175	2.92	118 ± 3	10.69 ± 0.15	186 ± 9	4.78 ± 0.17	175	2.92	175	2.92	0.00%	0.00%
3.8	175	3.02	116 ± 2	11.00 ± 0.07	198 ± 7	4.28 ± 0.15	175	3.02	183	3.55	0.00%	7.50%
3.8.1	176	3.03	115 ± 2	11.06 ± 0.04	194 ± 7	4.21 ± 0.13	176	3.03	177	3.13	0.00%	1.27%
3.8.2	183	3.10	126 ± 2	11.60 ± 0.06	201 ± 8	4.33 ± 0.17	183	3.10	183	3.24	0.00%	2.50%
4.0	147	3.83	97 ± 6	11.02 ± 0.07	193 ± 9	6.31 ± 0.29	167	5.27	185	6.07	9.09%	39.86%
4.1	164	4.01	105 ± 6	11.72 ± 0.06	215 ± 11	6.62 ± 0.30	164	4.01	174	4.72	0.00%	3.66%
4.2	164	3.98	107 ± 7	11.95 ± 0.06	214 ± 9	7.14 ± 0.31	164	3.98	169	4.34	0.00%	1.20%
4.3	541	3.82	365 ± 15	28.90 ± 0.06	711 ± 23	9.15 ± 0.27	541	3.82	668	5.14	0.00%	2.92%
4.3.1	168	4.03	114 ± 6	12.01 ± 0.06	213 ± 9	6.93 ± 0.33	168	4.03	168	4.03	0.00%	1.10%
4.4	232	6.85	173 ± 5	17.40 ± 0.05	323 ± 13	10.67 ± 0.30	256	7.58	298	9.83	2.60%	41.89%
4.5	265	7.39	220 ± 14	20.89 ± 0.05	373 ± 16	13.20 ± 0.37	282	8.48	345	11.15	2.40%	32.42%
4.6	297	8.53	223 ± 10	22.99 ± 0.06	412 ± 13	14.31 ± 0.35	297	8.53	339	10.54	0.00%	5.43%
4.7	320	9.06	236 ± 8	25.78 ± 0.10	447 ± 17	14.90 ± 0.39	320	9.06	365	11.18	0.00%	5.34%
4.8	327	9.70	243 ± 8	26.62 ± 0.07	452 ± 12	15.48 ± 0.37	327	9.70	327	9.70	0.00%	0.00%
4.8.1	327	9.70	246 ± 16	26.58 ± 0.08	455 ± 17	15.51 ± 0.33	327	9.70	327	9.70	0.00%	0.00%
4.8.2	327	9.70	236 ± 9	26.59 ± 0.09	455 ± 18	15.49 ± 0.37	327	9.70	327	9.70	0.00%	0.00%
4.9	334	9.49	248 ± 9	27.70 ± 0.09	469 ± 12	16.08 ± 0.29	334	9.49	352	10.75	0.00%	2.33%
4.10	336	9.48	278 ± 14	27.72 ± 0.09	469 ± 15	15.99 ± 0.40	336	9.48	340	10.39	0.00%	1.83%
4.11	311	8.71	249 ± 9	27.34 ± 0.06	404 ± 16	15.45 ± 0.49	312	9.34	339	11.80	0.49%	6.19%
4.12	471	11.00	334 ± 11	33.94 ± 0.08	603 ± 14	18.04 ± 0.42	472	11.61	509	14.72	0.50%	22.05%

package-constrained search can be used to avoid this undesirable behavior and improve structural cohesion of software systems.

In the last columns of Table 8 we report the natural disruption caused by each release of JUnit, in comparison to the previous immediate release. For each release, we computed lower and upper bound levels of disruption according to Intersection and Union DisMoJo (described in RQ5), where the first is a disruption measurement that considers only the classes that remained between releases and the latter considers not only classes that remained but also classes that were added or removed between releases.

As one can see, the most disruptive release of JUnit was release 4.4 with an upper bound disruption of 41.89%, yet still smaller than the average disruption caused by Bunch and package-constrained search (see Table 5). This observation adds evidence to the claim that even though cohesion and coupling optimisation is achievable, complete re-modularisations are unrealistic in real world software development, so that approaches that seek for a compromise between modularity improvement and familiarity to previously stablished structure are more likely to be adopted by software developers.

Therefore, we report on columns 8-11 of Table 8 the results achieved by the proposed multiobjective approach for modularity improvement and disruption minimisation. For each release of JUnit, we used the lower and upper bounds of natural disruption to pick solutions from the pareto front. Consider release 4.5, for example. For the lower bound cohesion value, we picked the modularisation from the pareto front with highest cohesion and disruption smaller than 2.40%. Similarly, for the upper bound cohesion improvement, we picked the solution with highest cohesion and disruption smaller than 32.42%. This way we are able to suggest modularity improvements that are bounded by the same range of disruption that is already familiar to the system's developers.

As an example, we report part of the pareto front found by the proposed multiobjective approach for release 4.0 of JUnit in Table 9, where duplicate or very similar solutions were omitted.

Table 9: Modularisations suggested by the package-constrained multiobjective search for JUnit 4.0. For each solution, we report the raw cohesion, the disruption given by DisMoJo and the number of moved classes in comparison to the original developers' implementation. We also highlight the solutions that best match the lower and upper bounds disruption levels of release 4.0.

Modularisation	Cohesion	Disruption	Number Of Moved Classes
(Original) 1	147	0.00%	0
2	150	3.70%	2
3	155	4.94%	3
(Lower Bound) 4	**162**	**6.10%**	**4**
5	167	9.76%	7
6	168	10.98%	8
7	169	14.81%	11
8	172	17.28%	13
9	175	18.52%	14
10	177	22.22%	17
11	183	23.46%	18
(Upper Bound) 12	**185**	**28.40%**	**22**

For each solution in the table, we present the cohesion value, disruption given by DisMoJo and the number of classes developers would need to move to a different package in comparison to the original implementation. As one can see, the multiobjective approach proposed in this paper is able to suggest modularisations with different levels of improvement and disruption, in a way that developers can choose the one the better suits the project needs in a specific scenario.

Release 4.0 of JUnit has a lower bound disruption of 9.09%; therefore, modularisation number 4 is the one that presents the most similar level of disruption, as depicted in Table 9. This solution moves only 4 classes from the original implementation, affecting only 3 out of the 11 packages in the system. More specifically, class `Description` is moved from package `org.junit.runner` to package `org.junit.internal.runners`, which is a reasonable refactoring because this class is used to describe different test runners in package `org.junit.internal.runners`. Moreover, class `Request` is moved from package `org.junit.runner` to package `org.junit.internal.requests`, which contains all classes related to requests in the system.

On another hand, developers can select the solution that is more similar to the upper bound disruption caused by release 4.0, which is modularisation 12 in Table 9. Interestingly, this solution performs the same modifications discussed above plus some "follow ups" to improve the cohesion even more, such as moving other classes related to `Request` to the `org.junit.internal.requests` package. In total, this solution moved 22 classes and affected 8 out of 11 packages of the system, achieving a cohesion improvement of 25.85%.

This case study illustrates how multiobjective search can be used in conjunction with longitudinal analysis of disruption to propose a set of modularisation solutions that present a compromise between modularity improvement and familiarity to existing structure, yet still bounded by the level of disruption inflicted by the developers of the system.

6 Threats to the Validity

This section describes the threats that might affect the validity of the empirical study reported in this paper and discusses our attempts to mitigate these threats.

Conclusion Threats are related to the analyses we performed and the conclusions we drawed from these analyses. Random and k-neighbourhood searches were executed one million times for each release, while the systematic search covered the whole nearest neighbourhood of the releases under study. Furthermore, both Bunch and Package-constrained search were executed 30 times for each release. In total, our analyses of RQ1-3 were based in more than 466 million different modularisations of the 10 systems and 233 releases under study, which we believe thoroughly accounts for the random nature of the algorithms we applied. In RQ4, the multiobjective approaches were only executed once for each release due to the large computation effort required to run the multiobejctive GA for 233 releases of medium to large real world software systems. However, we applied the Two-Archive GA, which was demonstrated to be stable and perform well in previous work [47][44].

Internal Threats consider the design of the experiments we carried out and the effects our design choices might have in our analyses. All the algorithms, fitness functions and parameters were based on previous and widely used literature on automated software modularisation [31][38][47][56]. Moreover, our data collection was performed based on a clear selection criteria and involved manual validation of all systems, releases and modularity data that were extracted.

External Threats are related to the generalisation of the findings reported by the empirical study. We performed the largest empirical study on automated software modularisation to date, involving subsequent releases of medium to large real world software systems. Furthermore, we make available in our supporting web page[4]all the modularity data we used in our empirical study to facilitate replications and extensions.

7 Conclusion and Future Work

The notions of software modularisation and cohesion/coupling have been proposed as good practices for software development since the 1970s, and many SBSE techniques have been proposed and evaluated since late 1990s to automate the decomposition of softaware systems in highly cohesive and loosely coupled modules. However, after surveying more than 30 related papers, we could not identify any study that has investigated the trade-off between the modularity improvement these automated techniques offer and the inherently disruption they cause to the original modular structure of software systems. Moreover, most of the surveyed papers only consider a single version of the systems under study, ignoring the previous releases. Therefore, we performed the largest empirical study on search based software re-modularisation so far, involving 233 subsequent releases of 10 medium to large real world software systems.

This study revealed that the modular structure of existing systems respect the raw cohesion and the MQ quality metrics, where the developers' implementation have better cohesion and/or MQ of more than 96% of the alternative modularisations created by random and neighbourhood search. However, we noticed that raw cohesion and MQ do not commonly agree when assessing the modularity of software systems due to the 'inflation effect' of the MQ metric that we exposed by applying the Bunch tool to the systems under study. Modularisations with more packages favour the MQ metric; therefore, Bunch creates an average of 493.11% new packages and decreases the cohesion of the systems in -46.25%, on average. As an attempt to mitigate the MQ's 'inflation effect', we introduced the package-constrained approach for automated re-modularisation, in which the search algorithm is constrained by the number of packages implemented by the developers. The package-constrained search was able to find modularisations with an average cohesion improvement of 25%.

Even though search based approaches can be used to improve the modular structure of software systems as assessed by both cohesion and MQ, we showed that the disruption caused by these approaches is high. On average, developers would have to change 80.39% and 57.82% of the structure to adopt modularisations suggested by Bunch and package-constrained search,

[4]`http://www0.cs.ucl.ac.uk/staff/mpaixao/cohCop/index.html` (Please note the url will be activated upon peer-reviwed publication)

respectively. Surprisingly, the disruption caused by Bunch and package-constrained solutions with very best modularity, as assessed by both cohesion and MQ, caused less disruption than the average. Motivated by this opportunity, we employed a multiobjective optimisation approach for automated software re-modularisation that attempts to maximise the modularity improvement and minimise disruption.

We showed that modularity improvement and disruption have a clear and constant trade-off over the pareto fronts of all systems under study. Moreover, based on a longitudinal analysis of developers implementations over releases, we estimated lower and upper bounds of 'acceptable' levels of disruption that developers have introduced. We found that our new multiobjective approach was able to improve, on average, 3.45% and 22.59% of the cohesion of the systems within this range of 'acceptable' disruption.

Finally, we performed a more detailed and qualitative analysis of some of the results we achieved for the JUnit system, where we presented in a series of case studies how the experiments and analyses carried out in this paper can be used together to provide a full picture of cohesion and coupling optimisation for a certain system. Among other things, we showed the evolution of cohesion throughout JUnit's releases, providing insights on how package-constrained search is able to avoid MQ's 'inflation effect', and how multiobjective search can be used in conjunction with longitudinal analysis of disruption to suggest re-modularisation solutions.

As future work, we discuss a set of extensions to the empirical study presented in this paper alongside further research that can be done by employing the techniques described in this work. In RQ1 and RQ2 we performed an incremental analysis of the respect developers have for measurements of cohesion and coupling by employing a range of search procedures. However, due to our experiment design and space constraints, we restrained this analysis to state of the art single objective search and did not assess developers' solutions regarding multiobjective approaches using the MCA and ECA suite of metrics. Such extension would benefit the search based modularisation community by providing insights on the fitness functions being employed by other approaches.

Our disruption analysis is based on the widely used MoJoFM metric, which in spite of its popularity, only considers "move class" and "join package" refactoring operations to measure the distance between two modularisations. The incorporation of other refactoring operations, such as "split package", in the distance metric may lead to a better assessment of the disruptive effect of search based re-modularisation.

We employed the Two-Archive GA as an optimisation algorithm to find solutions with a trade-off between modularity improvement and disruption. However, in spite of the good results achieved in this paper and also in previous research, a further comparison of the Two-Archive GA with other multiobjective algorithms and a simple weighted GA is needed to assess the most suitable optimisation algorithm for the task of finding software modularisations with high improvement and low disruption.

Finally, we plan to extend the investigations performed in this paper to consider not only structural measurements of software systems, but also other metrics of cohesion and coupling, such as semantic, co-changes and information theoretic. We also plan to use/adapt the techniques and analyses described in this paper to assess/measure structural and architectural debt of software systems.

References

[1] Hani Abdeen, Stéphane Ducasse, Houari Sahraoui, and Ilham Alloui. Automatic package coupling and cycle minimization. In *Reverse Engineering, 2009. WCRE'09. 16th Working*

Conference on, pages 103–112. IEEE, 2009.

[2] Hani Abdeen, Houari Sahraoui, Osama Shata, Nicolas Anquetil, and Stéphane Ducasse. Towards automatically improving package structure while respecting original design decisions. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 212–221. IEEE, 2013.

[3] Programmer's Friend Class Dependency Analyzer. `http://www.dependency-analyzer.org/`, 2016. Accessed in: May 2016.

[4] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[5] Marcio de Oliveira Barros. An analysis of the effects of composite objectives in multiobjective software module clustering. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 1205–1212. ACM, 2012.

[6] Márcio de Oliveira Barros, Fábio de Almeida Farzat, and Guilherme Horta Travassos. Learning from optimization: A case study with apache ant. *Information and Software Technology*, 57:684–704, 2015.

[7] Gabriele Bavota, Filomena Carnevale, Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. Putting the developer in-the-loop: an interactive ga for software re-modularization. In *Search Based Software Engineering*, pages 75–89. Springer, 2012.

[8] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology*, 23(1):1–33, feb 2014.

[9] James M Bieman and Linda M Ott. Measuring functional cohesion. *Software Engineering, IEEE Transactions on*, 20(8):644–657, 1994.

[10] William H Brown, Raphael C Malveau, and Thomas J Mowbray. Antipatterns: refactoring software, architectures, and projects in crisis. 1998.

[11] Ivan Candela, Gabriele Bavota, Barbara Russo, and Rocco Oliveto. Using Cohesion and Coupling for Software Remodularization : Is It Enough ? *ACM Transactions on Software Engineering and Methodology*, 25(3):1–28, 2016.

[12] Diego Doval, Spiros Mancoridis, and Brian S Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Software Technology and Engineering Practice, 1999. STEP'99. Proceedings*, pages 73–81. IEEE, 1999.

[13] Mathew Hall, Muhammad Ali Khojaye, Neil Walkinshaw, and Phil McMinn. Establishing the Source Code Disruption Caused by Automated Remodularisation Tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 466–470. IEEE, sep 2014.

[14] Mathew Hall and Phil McMinn. An analysis of the performance of the bunch modularisation algorithms hierarchy generation approach. In *4 th Symposium on Search Based-Software Engineering*, page 19, 2012.

[15] M. Harman, P. McMinn, J. T. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, pages 1–59. Springer, 2012.

[16] Mark Harman, Robert M Hierons, and Mark Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO*, volume 2, pages 1351–1358, 2002.

[17] Mark Harman, Stephen Swift, and Kiarash Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1029–1036. ACM, 2005.

[18] Jinhuang Huang and Jing Liu. A similarity-based modularization quality measure for software module clustering problems. *Information Sciences*, 2016.

[19] Jinhuang Huang, Jing Liu, and Xin Yao. A multi-agent evolutionary algorithm for software module clustering problems. *Soft Computing*, feb 2016.

[20] Kawal Jeet and Renu Dhir. Software Architecture Recovery using Genetic Black Hole Algorithm. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5, feb 2015.

[21] Kawal Jeet and Renu Dhir. Software module clustering using bio-inspired algorithms. *Handbook of Research on Modern Optimization Algorithms and Applications in Engineering and Economics*, page 445, 2016.

[22] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: from metaphor to theory and practice. *Ieee software*, (6):18–21, 2012.

[23] A. Charan Kumari and K. Srinivas. Hyper-heuristic approach for multi-objective software module clustering. *Journal of Systems and Software*, 117:384–401, jul 2016.

[24] A. Charan Kumari, K. Srinivas, and M. P. Gupta. Software module clustering using a hyper-heuristic based multi-objective genetic algorithm. In *Proceedings of the 2013 3rd IEEE International Advance Computing Conference, IACC 2013*, pages 813–818. IEEE, 2013.

[25] Duc Minh Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. An empirical study of architectural change in open-source software systems. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 235–245. IEEE Press, 2015.

[26] Meir M Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[27] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32. IEEE, 1997.

[28] Kiarash Mahdavi, Mark Harman, and Robert M Hierons. A multiple hill climbing approach to software module clustering. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 315–324. IEEE, 2003.

[29] Ali Safari Mamaghani and Mohammad Reza Meybodi. Clustering of software systems using new hybrid algorithms. In *Computer and Information Technology, 2009. CIT'09. Ninth IEEE International Conference on*, volume 1, pages 20–25. IEEE, 2009.

[30] Spiros Mancoridis, Brian S Mitchell, Yihfarn Chen, and Emden R Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 50–59. IEEE, 1999.

[31] Spiros Mancoridis, Brian S Mitchell, Chris Rorres, Yih-Farn Chen, and Emden R Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC*, volume 98, pages 45–52. Citeseer, 1998.

[32] Antonio Martini, Jan Bosch, and Michel Chaudron. Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology*, 67:237–253, 2015.

[33] Brian Mitchell, Martin Traverso, and Spiros Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 181–190. IEEE, 2001.

[34] Brian S Mitchell. *A heuristic search approach to solving the software clustering problem.* PhD thesis, Drexel University, 2002.

[35] Brian S Mitchell and Spiros Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 744. IEEE Computer Society, 2001.

[36] Brian S Mitchell and Spiros Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO*, volume 2, pages 1375–1382, 2002.

[37] Brian S Mitchell and Spiros Mancoridis. Modeling the Search Landscape of Metaheuristic Software Clustering Algorithms. In *Genetic and Evolutionary Computation Conference, GECCO 2003*, pages 2499–2510, Chicado, IL, USA, 2003. Springer Berlin Heidelberg.

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

[39] Brian S Mitchell and Spiros Mancoridis. On the evaluation of the bunch search-based software modularization algorithm. *Soft Computing*, 12(1):77–93, 2008.

[40] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-Objective Software Remodularization Using NSGA-III. *ACM Transactions on Software Engineering and Methodology*, 24(3):1–45, may 2015.

[41] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. Experimental assessment of software metrics using automated refactoring. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 49–58. ACM, 2012.

[42] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1):47–79, mar 2013.

[43] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. Multi-Criteria Code Refactoring Using Search-Based Software Engineering. *ACM Transactions on Software Engineering and Methodology*, 25(3):1–53, jun 2016.

[44] Matheus Paixao, Mark Harman, and Yuanyuan Zhang. Multi-objective module clustering for kate. In *Search-Based Software Engineering*, pages 282–288. Springer, 2015.

[45] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In *Genetic Programming*, pages 137–149. Springer, 2014.

[46] Kata Praditwong. Solving software module clustering problem by evolutionary algorithms. In *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, pages 154–159. IEEE, 2011.

[47] Kata Praditwong, Mark Harman, and Xin Yao. Software module clustering as a multi-objective search problem. *Software Engineering, IEEE Transactions on*, 37(2):264–282, 2011.

[48] Kata Praditwong and Xin Yao. A new multi-objective evolutionary optimisation algorithm: the two-archive algorithm. In *Computational Intelligence and Security, 2006 International Conference on*, volume 1, pages 286–291. IEEE, 2006.

[49] Roger S Pressman. *Software engineering: a practitioner's approach.* Palgrave Macmillan, 2005.

[50] Amir M Saeidi, Jurriaan Hage, Ravi Khadka, and Slinger Jansen. A search-based approach to multi-view clustering of software systems. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 429–438. IEEE, 2015.

[51] Olaf Seng, Markus Bauer, Matthias Biehl, and Gert Pache. Search-based improvement of subsystem decompositions. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1045–1051. ACM, 2005.

[52] Ali Shokoufandeh, Spiros Mancoridis, Trip Denton, and Matthew Maycock. Spectral and meta-heuristic algorithms for software clustering. *Journal of Systems and Software*, 77(3):213–223, 2005.

[53] Chris Simons, Jeremy Singer, and David R. White. Search-Based Refactoring: Metrics Are Not Enough. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9275, pages 47–61. 2015.

[54] I. Sommerville. *Software Engineering.* Addison Wesley, 2011.

[55] J. T. Souza, C. L. Maia, F. G. Freitas, and D. P. Coutinho. The human competitiveness of search based software engineering. In *Search Based Software Engineering (SSBSE), 2010 Second International Symposium on*, pages 143–152. IEEE, 2010.

[56] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 194–203. IEEE, 2004.

[57] Michel Wermelinger, Yijun Yu, Angela Lozano, and Andrea Capiluppi. Assessing architectural evolution: a case study. *Empirical Software Engineering*, 16(5):623–666, 2011.

[58] Edward Yourdon and Larry L Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design*, volume 5. Prentice-Hall Englewood Cliffs, NJ, 1979.