Information and Software Technology xxx (2014) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Learning from optimization: A case study with Apache Ant

Márcio de Oliveira Barros [a,*], Fábio de Almeida Farzat [b], Guilherme Horta Travassos [b]

[a] *Post-Graduate Information Systems Department, PPGI/UNIRIO, Av. Pasteur 458, Urca, Rio de Janeiro, RJ, Brazil*
[b] *Computers and System Engineering Department, COPPE/UFRJ, Cx Postal 68501, Cidade Universitária, Rio de Janeiro, RJ, Brazil*

A R T I C L E   I N F O

Article history:
Received 27 November 2013
Received in revised form 24 July 2014
Accepted 25 July 2014
Available online xxxx

Keywords:
Apache Ant
Heuristic search
Software module clustering
Experimental Software Engineering

A B S T R A C T

Context: Software architecture degrades when changes violating the design-time architectural intents are imposed on the software throughout its life cycle. Such phenomenon is called architecture erosion. When changes are not controlled, erosion makes maintenance harder and negatively affects software evolution.
Objective: To study the effects of architecture erosion on a large software project and determine whether search-based module clustering might reduce the conceptual distance between the current architecture and the design-time one.
Method: To run an exploratory study with Apache Ant. First, we characterize Ant's evolution in terms of size, change dispersion, cohesion, and coupling metrics, highlighting the potential introduction of architecture and code-level problems that might affect the cost of changing the system. Then, we reorganize the distribution of Ant's classes using a heuristic search approach, intending to re-emerge its design-time architecture.
Results: In characterizing the system, we observed that its original, simple design was lost due to maintenance and the addition of new features. In optimizing its architecture, we found that current models used to drive search-based software module clustering produce complex designs, which maximize the characteristics driving optimization while producing class distributions that would hardly be acceptable to developers maintaining Ant.
Conclusion: The structural perspective promoted by the coupling and cohesion metrics precludes observing the adequate software module clustering from the perspective of software engineers when considering a large open source system. Our analysis adds evidence to the criticism of the dogma of driving design towards high cohesion and low coupling, at the same time observing the need for better models to drive design decisions. Apart from that, we see SBSE as a learning tool, allowing researchers to test Software Engineering models in extreme situations that would not be easily found in software projects.

1. Introduction

Software architecture is the set of major design decisions governing the development of a system [66]. The architecture prescribes how the system is divided into subsystems, components and so forth. Architectural issues include the distribution of functionality amongst components, the interfaces through which they communicate with each other and the usage dependencies linking these components. At design time, the architecture postulates the major components the system will be divided into, the functionalities they are expected to provide, and the dependencies allowed amongst them. The source code must then follow these tenets, implementing the components according to the restrictions imposed by the design-time architecture.

Architecture erosion is the process through which the system's architecture gradually degrades as the maintainers make changes that violate the design-time architectural intents [5]. Erosion leads to architecture mismatch [23], a situation where the current implementation of a software system significantly differs from its original design, as shown in its design-time architecture. Assuming that the original architecture was implemented as proposed in the first system release, many factors contribute to architecture erosion during its maintenance, including short deadlines, high developer turnover, lack of proper developer training, and an absence of long-term developer commitment to the project. All types of software systems are subject to erosion. For instance, open-source systems, such as Apache Ant, Eclipse or OpenOffice, may be subjected to erosion as most of their developers are not dedicated on a full-time basis to the project and functionality must be

* Corresponding author.
E-mail addresses: marcio.barros@uniriotec.br (M.O. Barros), ffarzat@cos.ufrj.br (F.A. Farzat), ght@cos.ufrj.br (G.H. Travassos).

http://dx.doi.org/10.1016/j.infsof.2014.07.015
0950-5849/© 2014 Elsevier B.V. All rights reserved.

continually added to keep these systems useful and to allow them to retain a willing user community.

Erosion hinders maintenance and software evolution, as violations of the architectural tenets frequently add complexity to the design, making it harder to understand. Moreover, erosion has a positive feedback effect in promoting more erosion: as developers find a system whose structure is already compromised, they may feel less compelled to correct the structure and less guilty of adding to that corruption. After all, they are also probably pressed by similar short deadlines and other restrictions imposed upon former developers.

Considering architecture erosion in open-source systems, Apache Ant (or Ant) is an interesting case of software evolution. Ant is a build automation tool frequently used to support continuous integration (CI). CI is a software development practice in which members of a team frequently integrate their work, usually once a day and per person [21]. Integration involves the execution of tasks aimed at building and checking the software. Ant was conceived as a framework to organize abstract CI tasks in such a way that adding new types of tasks would be easy. It uses XML files to describe what tasks should be run (and in which order) during integration. These tasks typically involve the downloading of code from a version control system, copying files, compiling code, executing unit tests, building deployment descriptors, and so on. They have to be properly sequenced and executed, and specific people have to be notified by email, instant messaging or phone if errors are found during their execution. The acceptance of Apache Ant by developers shows that it can deal with a real problem and is, at least, an acceptable solution. Thus, many types of integration and build tasks were added to supplement the framework. Twenty-four versions of Ant (Java code) were released from July 2000 to March 2013 and the current version used in our analysis (v1.9.0) is over 1100 classes, organized in 60 packages.

Despite being accepted and used by many developers, evolution was not a smooth process for Ant. Its first version was based on a very simple architecture with three major components mapped into four packages. There were strictly defined connections displaying possible use dependencies among these packages. However, as the software evolved and grew to 10 times its original size, the number of packages also expanded, as well as the number of dependencies among them. Many dependencies that would not be expected in the original architecture were created and the boundaries separating the components of that architecture cannot be easily found in recent versions. This may have reduced reusability, understandability, and testability. In a typical case of architectural mismatch, small changes made by developers on a daily basis over the course of the software life-cycle have increased the conceptual distance between the current implementation and the original design. All of this affects the quality of module distribution.

The quality of module distribution is frequently addressed by two metrics: coupling and cohesion [71]. Coupling measures the dependency between clusters, while cohesion measures the internal strength of a cluster. The technical literature on software design suggests that a given module distribution should be considered good decomposition for a software system if it displays low coupling and high cohesion. These metrics are calculated from the dependencies seen among modules. A module A is said to depend on a module B if it requires some function, procedure, or definition declared in module B to perform its duties. However, observing the distribution of Ant's modules and identifying the most adequate module distribution for the system are not easy tasks, taking into account the evolution of Ant in 13 years and the amount of data that has to be analysed.

Search-based Software Engineering (SBSE) reformulates Software Engineering problems as optimization problems [28] and suggests using meta-heuristic search algorithms, such as local search or genetic algorithms, to find good solutions for these problems. One of the Software Engineering problems frequently addressed in a search-based perspective relates to the reorganization of system architecture to improve its long-term maintainability. This problem is known as software module clustering, as it addresses the grouping of modules into clusters that might be meaningful to developers. The search reorganizes the structure of a system by trying to maximize cohesion and minimize coupling at package-level. Despite its being studied for about two decades, the technical literature reporting on the software module clustering problem does not have many case studies on its application to large software systems. While the typical large instance used to address the problem in the technical literature has about 300 classes [42,30,56,7], Apache Ant represents a much larger case with approximately 1100 classes.

In this paper we report on an exploratory study in which we first characterize Apache Ant's evolution in terms of size, change dispersion, cohesion, and coupling metrics. We refer to the value of these metrics over distinct versions of Ant released from July 2000 to March 2013 to highlight the potential introduction of architecture-level and code-level problems that might affect the cost of changing the system. Then, we subject Apache Ant to software module clustering in the hope that optimization will reorganize its modules to minimize dependencies among packages and, possibly, allow the original architecture designed for the system to re-emerge. For this, we use a Hill Climbing search – one of the most frequently used algorithms to address the software module clustering problem. However, the algorithm produced very complex solutions, which indeed maximized the characteristics driving the optimization, while producing solutions that would hardly be acceptable (based on our experience) to a software developer maintaining Ant. Thus, we add evidence to the criticism regarding the dogma of high cohesion and low coupling [4], whilst observing the need for better models to drive design decisions and automated optimization. In this context, we see that optimization and SBSE can be learning tools, by allowing researchers to test Software Engineering models in extreme situations that would not be easily done in software projects. By exploring these limits, we can determine how our theories (and, subsequently, our models) behave and whether they might be acceptable as proxies of good developer intentions towards long-term maintenance.

A previous (and shorter) version of this paper [8] was presented at SSBSE 2013. This extended version brings new contributions, summarized as follows:

- An extensive analysis of a software evolution case for a large software system – Apache Ant.
- Adding evidence that maximizing cohesion and minimizing coupling may not lead to solutions that would resemble the developers' intention as regards software maintenance.
- Evidence that search-based optimization may be an useful learning tool in the sense that optimization takes our models to extremes which would not be explored by manually reorganizing the systems. Thus, limitations and flaws in our models can be detected to be further analysed and corrected, ultimately leading to better models and a more complete understanding of the nature of software systems.

This paper is organized in 7 sections, starting from this introduction. Section 2 presents background information on the physical manifestation of the problem we are addressing (architecture mismatch) and the intervention we have applied (search-based software module clustering). Section 3 describes our case for software evolution, Apache Ant, presenting the versions we analysed and depicting their architecture to visually present how complex it became over time. Section 4 presents the metrics we used to analyse the evolution of Ant and our findings regarding architectural and code-level issues found in the course of this evolution. Section 5 presents the application of search-based clustering for version 1.9.0 of Apache Ant and its effects on the former metrics and architecture of the system. Section 6 discusses the results found in Section 5 and suggests that SBSE can be a learning tool. Finally, Section 7 provides our conclusions and future directions.

2. Background

2.1. Software module clustering

Software module clustering addresses the distribution of modules representing domain concepts and computational constructs comprising a software system into larger, container-like structures. A proper module distribution aids in the identification of the modules responsible for a given functionality [13], provides easier navigation among software parts [24] and enhances source code comprehension [37,45] . Therefore, it supports the development and maintenance of a software system. Apart from that, experiments have shown a strong correlation between bad module distributions and the presence of faults in software systems [13,55].

To evaluate the quality of its module distribution, a software system is usually represented as a Module Dependency Graph, or MDG [42]. The MDG is a directed graph in which modules are shown as nodes, dependencies are shown as edges, and clusters are partitions. Weights may be assigned to edges in order to represent the strength of the dependency between modules. The coupling of a cluster is calculated by summing the weights of edges leaving or entering the partition (inter-edges), while its cohesion is calculated by summing the weights of edges whose source and target modules pertain to the partition (intra-edges) [56]. In non-weighted MDG, the weight of every edge is set to 1 and coupling and cohesion are calculated by counting intra and inter-edges.

Given a MDG, the software module clustering problem can be modeled as a partition problem aimed at minimizing coupling and maximizing cohesion. This problem is known to be NP-hard and bears two aspects which make it amenable for a search-based approach: a large search space and quickly calculated fitness functions. Mancoridis et al. [42] were the first to present a search-based assessment of the problem. They propose a single-objective Hill Climbing search to find the best module distribution for a system. The search is guided by a function called modularization quality (MQ), calculated by Eq. (1), where N represents the number of clusters, C_k represents a cluster, i and j respectively represent the number of intra-edges and inter-edges from C_k. MQ may also be calculated for weighted MDGs, where weights on an edge represent the number of sites on its source node on which methods from its target node are called. In such cases, i and j are calculated by summing the weights on intra-edges and inter-edges from C_k. MQ looks for a balance between coupling and cohesion, rewarding clusters with many intra-edges and penalizing them for dependencies in other clusters. The Hill Climbing search aims to find partitions of a MDG with the highest MQ possible, thus addressing software module clustering as a maximization problem. Hereafter, we use MQ as calculated for non-weighted graphs.

$$\mathrm{MQ} = \sum_{k=1}^{N} MF(C_k) \quad MF(C_k) = \begin{cases} 0, & i = 0 \\ \frac{i}{i+j/2} & i > 0 \end{cases} \tag{1}$$

Harman et al. [30] compare MQ with a clustering fitness function named EVM [68] regarding its robustness to the presence of noise in the MDG. EVM is calculated as a sum of cluster scores and is expected to be maximized by the search process. Its formulation is presented in Eq. (2), where N represents the number of clusters, C_k represents a cluster, $|C_k|$ represents the number of modules in C_k, and uses(i,j) is a Boolean function which yields true if and only if module i depends on module j. The score of a given cluster starts as zero, being calculated as follows: for each pair of modules in the cluster, the score is incremented if there is a dependency between the modules; otherwise, it is decremented. The fitness function rewards modules which depend on other modules pertaining to the same cluster and does not take into account dependencies on modules from other clusters. Differently from MQ, EVM does not take dependency weights into account. Therefore, it does distinguish weighted and non-weighted MDGs. When compared to a baseline module distribution, clusterings based on EVM were found to degrade more slowly in the presence of noise (represented by adding random dependencies to the original module dependency structure) than MQ-based clusterings.

$$\mathrm{EVM} = \sum_{k=1}^{N} CS(C_k) \quad CS(C_k) = \sum_{i=1}^{|C_k|} \sum_{j=i+1}^{|C_k|} USE(i,j)$$
$$USE(i,j) = \begin{cases} +1, & \mathrm{uses}(i,j) \vee \mathrm{uses}(j,i) \\ -1, & \textit{otherwise} \end{cases} \tag{2}$$

To illustrate the calculation of MQ and EVM, Fig. 1 shows the structural dependencies for a hypothetical software system made by 8 modules, 3 clusters, and 10 dependencies among modules. It also shows the MQ and EVM calculation for the system, decomposed into the individual modularization factors and cluster scores for each cluster. Both metrics will be used in our search-based analysis of Apache Ant.

2.2. Architectural mismatch

Although different factors can contribute to the decay of software quality [6], architecture erosion is a long-term process of software decay whose physical manifestation in the source code of a software system takes the form of modules, components and interfaces which are not acting in accordance to the design-time architecture. These architectural and code-level constructs may be implementing features due to other parts of the system, may be more connected to these other parts than they should be, or may present other undesirable characteristics. Many researchers have proposed classifications for these problems and some of these are briefly described in the following paragraphs.

Code smells [20,46] are low-level design flaws related to the implementation of functionalities within the components. Mäntylä and Lassenius [44] present 21 code smells which are grouped into 5 categories, namely *Bloater smells*, *Object-orientation Abusers*, *Change Preventers*, *Dispensable*, and *Coupler* smells. *Bloater* smells represent code elements (methods, parameters, data groups, and so on) that have grown so large developers cannot handle them effectively. *Object-Orientation Abusers* represent implementation constructs (declarations, commands, inheritance, amongst others) that do not properly use the design alternatives and constructs offered by object-orientation. *Change preventers* are smells that increase the effort and risk involved in changing or further developing a software system. *Dispensables* represent unnecessary code elements that should be removed from the code to make the latter more comprehensible. Finally, *Couplers* concern high-coupled classes, that is, classes that depend too much on other classes to perform their duties.

While code smells are low-level design flaws related to the implementation of functionalities within the components, Garcia et al. [22] introduce the concept of *architecture smells* to denote design fragments which are recurring in software systems and that may have a long-term negative impact on their maintainability. Architectural smells are high-level design flaws related to a problematic division of system functionality into components

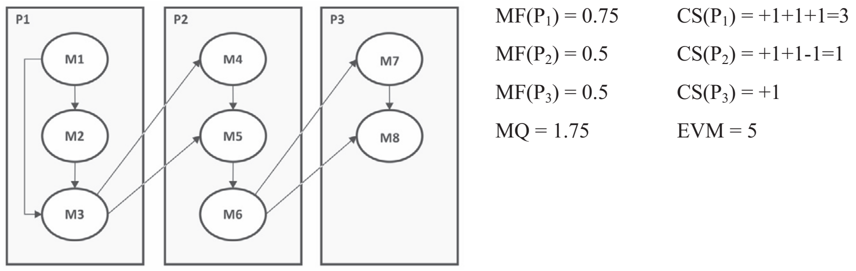

Fig. 1. A software system, along with its modularization quality and EVM.

and connectors. The authors argue that many maintenance issues originate from the poor use of architecture-level abstractions, such as components and connectors. These problems may be introduced at design time, due to an inefficient distribution of load and features throughout the components and connectors, or during maintenance, as manifestations of architecture erosion. The authors identified four major architecture-level design flaws while examining a large set of industrial software systems: *ambiguous interface*, *extraneous connector*, *connector envy*, and *scattered functionality*. *Ambiguous interfaces* offer a general entry-point to a component with no strong semantic description of the control or data-related information to be presented or collected from this entry-point. An *extraneous connector* is observed when two components use more than one type of connector to exchange control and data, thus accumulating the negative aspects of each connector. Components with *connector envy* encompass extensive interaction-related functionality (communication, coordination, conversion, and facilitation) that should be delegated to a connector. Finally, *scattered functionality* describes a system where more than one component is responsible for carrying out a given high-level functionality and some components are also responsible for infrastructure services to be used by other components.

In our analysis of Apache Ant's evolution profile, we are most interested in the ability to change the system and how it might be affected by reorganizing the distribution of classes that build the system into packages. Ant is a framework for continuous integration and the ability to incorporate new automated software development tasks is a keystone for the system to remain useful. So, the cost of changing and evolution in the framework has to be controlled. On the other hand, we will apply search-based optimization upon the distribution of classes into packages and do not expect to change other design features. Given the scope of our analysis, we concentrate on the *Change Preventer* and *Coupler* code smells, along with the *Scattered Functionality* architecture smell.

Change Preventers and *Scattered Functionality* smells violate the rule that suggests that classes (or at least groups of classes) and possible changes should have a one-to-one relationship. For instance, changes to the database should only affect a given set of classes, while changes to calculation formulas should only affect other groups of classes. *Shotgun Surgery* [36] is a *Change Preventer* smell detected when a change requires cascading changes in several related methods and classes. While the original change might be restricted to a few classes, dependencies among them and the rest of the system promote much more code churn[1] [17] than expected, thus affecting classes that were previously tested and working. Taking a package for an architectural component, in this paper we view the *Scattered Functionality* smell as a projection of *Shotgun Surgery* on a package level: signs of *Scattered Functionality* are found if a change requires editing classes residing in many different packages. *Divergent Changes* [36] is another *Change Preventer* smell and it is detected when a class is frequently changed in different ways for different reasons. In such cases, the class seems to lack a clear and well-defined objective and may be redesigned. *Divergent Changes* is a class-level smell unaffected by the moving of classes to other packages. Thus, it is outside the scope of the present analysis.

Coupler smells are concerned with strong relationships that (parts of) a class presents to other classes. The *Feature Envy* smell (also known as *Intensive Coupling* or *Disperse Coupling*) [36] is observed when a method calls more methods from other classes than from itself. Since this method is tightly coupled to other classes, it seems to be misplaced in the current one. A second smell, *Inappropriate Intimacy*, happens when two classes exchange many method calls among each other, instead of carrying out their duties by themselves. As those classes are tightly coupled, developers might consider implementing these features as a single class. In our analysis, coupling and cohesion are measured at package level, considering dependencies among classes to infer dependencies among packages and to evaluate the effects of moving classes between packages upon these dependencies.

3. Apache Ant as a case for software module clustering

Ant is primarily used for building and deploying Java projects under the Apache Software Foundation, but can be used for every possible repetitive task, such as unit testing execution or documentation generation. As a build automation tool, it typically runs without a graphical user interface, receiving parameters directly from the command line. This characteristic allows it to be easily integrated into different continuous integration environments. So, many open-source and industrial projects have adopted Ant around the world, creating the need to continuously evolve the software and fix the defects found by the users.

Twenty-four versions of Ant were released since its inception in July 2000 to March 2013, including 9 major and 15 minor versions. All versions were implemented in Java. The first version (v1.1.0) had 102 classes distributed into 4 packages. The software grew over the last years, as a large number of task definitions were added. The last version available in the period of our analysis (v1.9.0, out in March 2013) is over 1,100 classes organized in 60 packages. Fig. 2 shows a timeline for the released versions of Apache Ant.

Fig. 3 shows Apache Ant's design-time architecture, with its major packages and allowed dependencies among them. The architecture has three major components: an utilities library (*util*), an automation and notification framework (*ant*), and a set of task definitions (*taskdefs*). The utilities library provides a set of general use services which may be useful for several build automation tasks. The task definitions package contains several independent task definitions, each representing a build automation step. Finally, the central package represents a scheduler, which reads a XML file

[1] Code churn is defined as lines added, modified, or deleted to a file from one version to another.

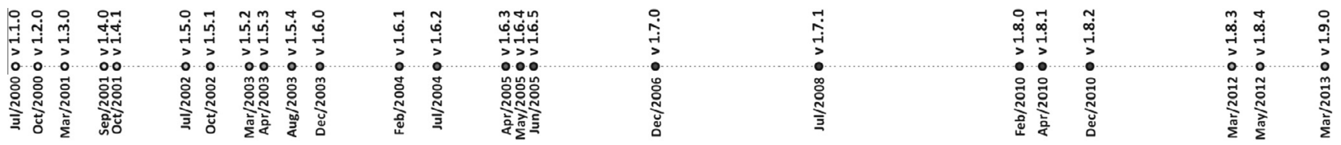

Fig. 2. Timeline for Apache Ant's released versions, from version 1.1.0 to version 1.9.0.

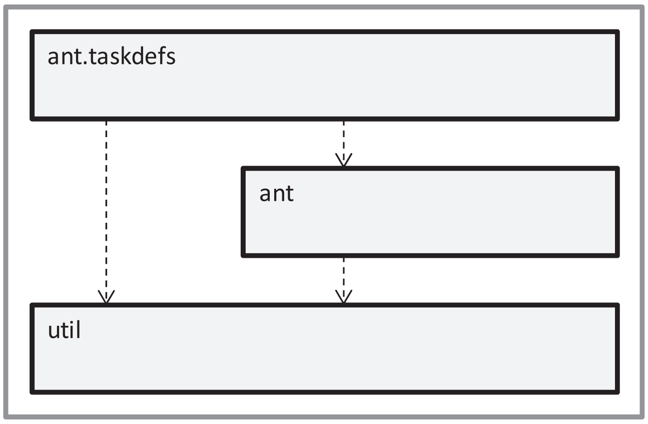

Fig. 3. Reference architecture for Ant.

stating the proper sequence of tasks to be executed during a build and coordinates their execution, notifying selected users by email when certain events happen during their execution. Task definitions may use the utilities library and the framework to perform their tasks, while the framework may use the libraries for notification and task execution.

Fig. 4 shows package dependency charts for all 24 versions of Apache Ant. In each of these charts, points represent packages and lines represent use dependencies between two packages. A package A depends on a package B if at least one class from A depends on at least one class from B to implement its features. Due to space limitations, dependency direction and package names are suppressed. Instead of providing a detailed description of the project's architecture, the charts in Fig. 4 intend to show how complexity was incorporated into an originally simple software, as it evolved.

Thirteen years of maintenance and a growth to 10 times its original size are a testament for Ant's usefulness. But both also provide the conditions for the dynamics of software evolution to play its role over the course of the software's life-cycle. As pointed by Lehman [38], as a software system is used, demands for new features grow. These demands must be quickly attended; otherwise, users will lose their interest and possibly abandon the software project for other products or their former way to deal with the problem. To quickly respond to user demands, a project leader needs more developers. These developers, in their turn, may not be so versed on the principles of the architecture designed for the software project and may (intentionally or not) violate some of these principles to implement those new features. Increasing workload, lack of documentation, and distance to peers increase the difficulties in passing the knowledge and the rationale about the assumptions and restrictions that underpin the architecture. So, as features are added to new versions, the architecture of the software system may become progressively more complex and, in the long run, may completely lose the connections with its original design.

These dynamics can be perceived in Table 1, which shows the yearly-grouped number of new issues registered by users and developers in Ant's trouble ticket system[2] for its first line. The number of tickets is a compound indicator of both project usage and maturity. In its early stages, a system may have defects that were not found during testing and are eventually found by the user community, thus generating a large number of issue reports. Later, as the system matures and not many defects remain to be found, the number of new issues is expected to drop. A large number of tickets were registered in the early years of Apache Ant: 51% of the 5783 issues registered up to November, 2013 were opened in 2003 or earlier. The trend in adding new issues to the trouble ticket system has been decreasing since then.

The second line in Table 1 depicts the number of commits to the source code version control repository over the years. This data is related to project activity, as developers can only commit code that was changed. The number of code commits is strongly and positively correlated ($\rho = 0.9440$ for the non-parametric Spearman rank order correlation) to the number of issues registered in the trouble ticket system on a yearly basis, possibly indicating developers were responding to these issues. However, we cannot be absolutely sure of the relationship between issues and commits as the dates the issues were resolved are not precisely recorded in the trouble ticket system (for instance, 4190 issues were changed in 2008) or were not easily accessible for collection and processing. A non-parametric correlation measure was used because the number of issues and commits do not resemble the normal distribution, as confirmed by the low p-value (0.05) resulting from the application of the Shapiro–Wilk normality test to the dataset covering the number of issues.

The third line in Table 1 shows the size of Ant's development team in each year. This data was collected from the log of the version control system and represents the number of developers which have made at least one source code commit in that year. The last line in Table 1 was collected on the same basis of the third line and represents the number of developers who performed their first code commit in that year. It is important to notice that the team developing Ant (13 developers in 2000) received 13 new developers from 2001 to 2003 and lost 11 of its original developers in the same period. Only 4 out of 13 original developers were in the team in 2002 when the number of issues started to rise and only 2 in 2003 when this number peaked. So, the effects of solving a huge number of issues with a group of new developers can be seen in the complexity added from version to version in Fig. 4. The simple structure of functionality distribution and communication channels conceived by the first developers (first picture, in the top most-left corner) was lost over the course of evolution. The architecture of the tool became gradually more complex, with many dependencies appearing among a growing number of packages.

Thus, the research question addressed in our exploratory study is: how can a software system conceptually as simple as observed in the architecture shown for version 1.1.0 in Fig. 4 become as complex as the one observed for version 1.9.0 in the same figure? The characteristics of this software throughout its life-cycle will be examined in the next section, with the code and architectural smells shown in Section 2.2 as background. Then, we evaluate

[2] http://ant.apache.org/bugs.html.

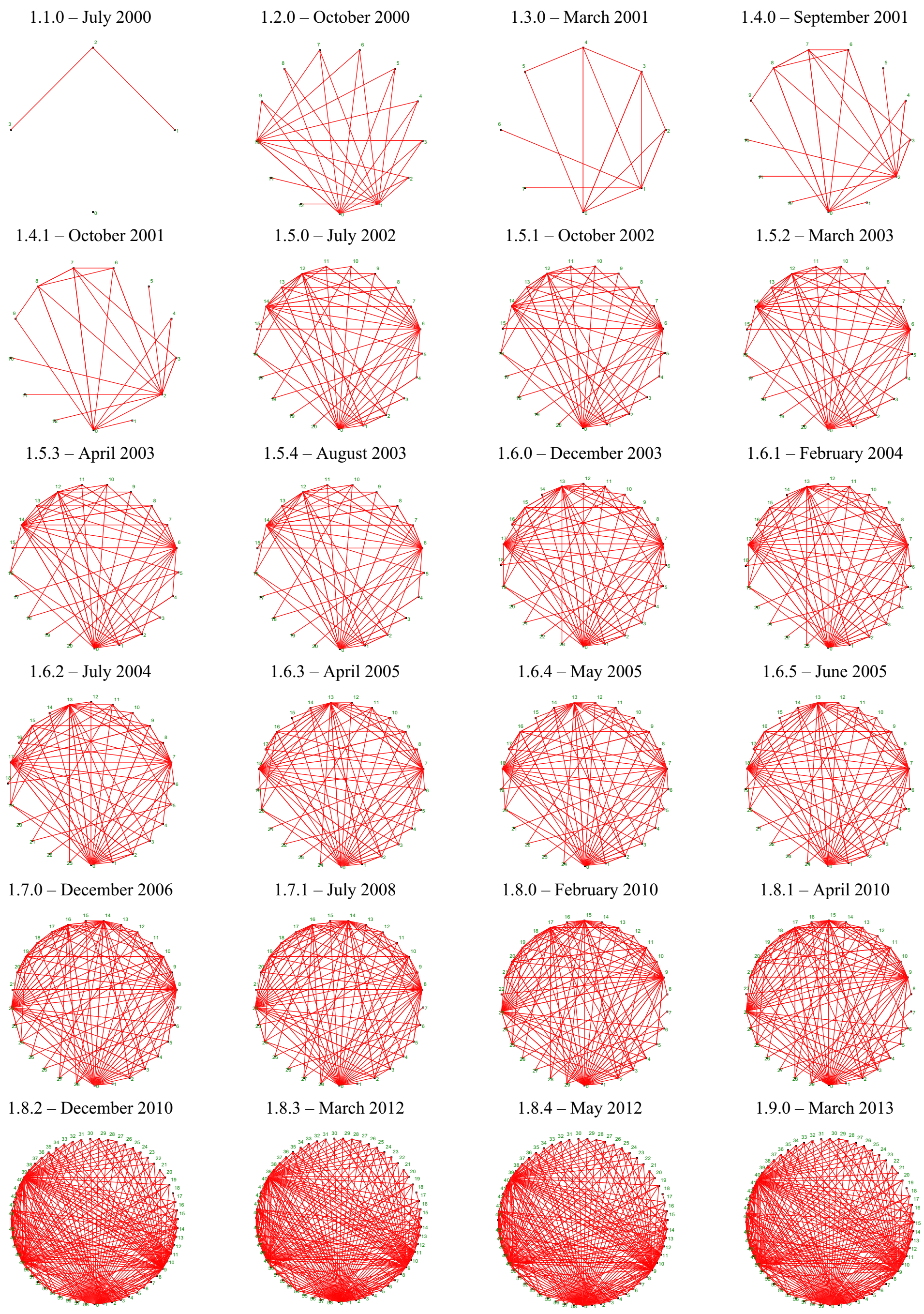

Fig. 4. The evolution of Apache Ant's architecture from version 1.1.0 to version 1.9.0.

Table 1
Yearly account of the number of trouble tickets registered for Apache Ant over the course of its evolution.

	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012
Issues registered	637	1042	1211	772	529	337	305	222	171	200	122	130
Code commits	683	840	888	481	810	562	367	341	241	197	65	75
Team size	13	13	11	11	13	11	9	9	9	10	6	5
New developers	6	3	4	3	2	2	0	1	1	1	0	0

whether applying a search-based reorganization of Ant's architecture would reduce the number and dispersion of dependencies among packages and bring the system back to its original simplicity. The results of our analysis are provided in Section 5.

4. Measuring Ant's evolution

We used different measures to build an understanding of what happened between successive versions of Apache Ant. We started

Table 2
Metrics used to evaluate the evolution of Apache Ant, separated in the 3 categories under which our analysis will be carried on in the following subsections.

Size	Number of packages
	Number of classes
	Number of attributes
	Number of methods
	Number of public methods (cost function)
	Number of dependencies (cost function)
	Class elegance (cost function)
Change dispersion	Number of single class commits (maximization function)
	Number of classes per commit (cost function)
	Number of single package commits (maximization function)
	Number of packages per commit (cost function)
Coupling and cohesion	Coupling between objects (CBO) (cost function)
	Afferent coupling (cost function)
	Efferent coupling (cost function)
	Lack of cohesion (cost function)
	Modularization quality (maximization function)
	EVM and cluster scores (maximization functions)

by analysing selected size and complexity metrics to determine the increase in functionality between consecutive system versions (Section 4.1). From that analysis, we selected the versions which represented large increases in functionality for further analysis. Then, we applied change dispersion metrics (Section 4.2), along with coupling and cohesion metrics (Section 4.3) to understand whether these properties have significantly changed over time, thus providing some evidence that the smells presented in Section 2.2 have been acting throughout the versions of interest.

As described in the following subsections, Ant has steadily grown over the course of its 12-year existence, it has shown limited effects of the *Shotgun Surgery* and *Scattered Functionality* smells, increasing coupling and low cohesion. Table 2 summarizes the metrics used for each evaluation carried on in this section, indicating whether they are cost functions (which should be minimized by design) or maximization functions when such classification applies.

4.1. Size metrics

We have selected the following system size and complexity measures to describe the growth of Apache Ant over its released versions: number of packages, number of classes, number of attributes, number of methods, number of public methods, number of dependencies, and class elegance. Most of these metrics are simple counts and their data were collected using the PF-CDA[3] static analysis tool (number of packages, classes, dependencies, and elegance) or directly accessing the binary code that runs over the Java virtual machine using the BCEL[4] byte-code analysis library.

The number of packages metric counts the number of packages holding at least one class in a program. The number of classes metric counts the number of classes that form a program, including nested and inner classes. The number of attributes metric counts the number of attributes held by the classes that form a program, despite their visibility (public, protected, or private) and scope (class or instances attributes). Similarly, the number of methods metric counts the number of methods held by the classes that take part in the source code, despite their visibility and scope. All of these metrics are related to the bulk size of the system, either as an overall measure (data and functions observed together, for packages and classes), a data measure (for attributes), or a functional measure (for methods).

The number of public methods metric, on the other hand, counts only the public methods held by the classes comprising a released version of Apache Ant, despite their being class-level or instance-level methods. This is related to the number of messages that can be exchanged amongst classes in the program and, thus, is a measure of the complexity of the system. The number of public methods should be kept as low as possible to allow for systems with simpler class-interaction protocols. Given that developers must understand these protocols to change or evolve the system, reducing the number of public methods tends to lead to systems which are less error-prone and easier to maintain.

The number of dependencies metric counts the number of class-to-class use dependencies, which relates to the number of links among classes and, indirectly, among packages, as shown in Fig. 4. The number of dependencies should be kept as low as possible in order to provide a system whose classes have few connections to other classes and, thus, are more autonomous to perform their duties.

Class elegance was put forward by Simons and Parmee [63] as a way to evaluate class concentration in a few packages in software design. It is calculated as the standard deviation for the number of classes in each package and is expected to be a small number, denoting that all packages have roughly the same number of classes.

Table 3 shows the selected size and complexity metrics for the 24 released versions of Apache Ant. The *Packages* column shows the number of packages in a given version. The *Classes* column shows the number of classes in a given version. The *Attributes* column shows the total number of attributes distributed along the classes for a given version. The *Methods* column shows the number of methods distributed along the classes for a given release, while the *Public Methods* column shows how many of these methods are public. The *Dependencies* column shows the number of dependencies among classes in a given release. Finally, the *Class Elegance* column represents the Simon and Parmee's class elegance measure for the version under study. Below the name of each metric, we present a symbol denoting whether such metric should be reduced (∇) or increased (Δ) to improve the design of the system. Metrics which are not affected by our analysis procedures are marked with a (=) symbol. This notation will be used in tables presented in further sections of this paper.

The data in Table 3 shows that Ant has been growing steadily from its early versions in terms of number of packages and classes. The mean number of classes per package[5] is 21.7 (median = 21.1, standard deviation = 3.72), but this number has not been stable[6] over the years, becoming as low as 13.3 classes/package (in version 1.2.0) or as high as 29.1 classes/package (in version 1.8.1). Lanza and Marinescu [36] studied 45 Java programs and found that the mean number of classes per package is 17, ranging from a minimum 6 to a maximum 26 in their dataset. Thus, Ant has been above average for all but one (version 1.2.0) of its released versions. Most importantly, two versions (1.8.0 and 1.8.1) were above the observed upper limit. Major increases in the number of classes between consecutive releases were found in early major versions (a 70% increase from version 1.1.0 to 1.2.0, 51% from version 1.4.0 to 1.5.0, and 42% from

[3] http://www.dependency-analyzer.org/.
[4] http://commons.apache.org/proper/commons-bcel/.

[5] Statistics presented in this subsection are summaries of release summaries. For instance, we calculated the mean number of classes per package for each release of Apache Ant and then presented the mean, median, standard deviation, minimum, and maximum values for these means over the course of different releases. A similar procedure was performed for all size metrics, except class elegance which is represented by a single number for each release.
[6] For the sake of our analysis, a measure is considered to be stable if all its observations remain at most 10% apart from its mean value.

Table 3
Size and complexity measures for 24 released versions of Apache Ant from July 2000 to March 2013. The selected metrics show that the system is growing steadily both in size and complexity.

Version	Packages	Classes	Attributes	Methods	Public methods	Dependencies	Class elegance ∇
1.1.0	4	102	141	816	633	362	8.3
1.2.0	13	173	249	1521	1186	803	11.9
1.3.0	8	187	294	1548	1223	855	13.0
1.4.0	13	265	410	2193	1729	1228	16.8
1.4.1	13	265	410	2202	1735	1230	16.8
1.5.0	21	401	584	3379	2639	1893	20.8
1.5.1	21	401	584	3389	2644	1894	20.8
1.5.2	21	406	591	3457	2703	1942	21.1
1.5.3	21	407	593	3465	2708	1947	21.2
1.5.4	21	407	597	3465	2708	1947	21.2
1.6.0	24	523	797	4651	3639	2557	26.0
1.6.1	24	524	797	4679	3665	2562	26.1
1.6.2	24	553	843	4884	3817	2445	27.7
1.6.3	25	576	880	5124	3971	2566	29.3
1.6.4	25	576	880	5126	3973	2566	29.3
1.6.5	25	576	880	5129	3973	2568	29.3
1.7.0	29	752	1162	6581	4978	3505	34.5
1.7.1	29	769	1191	6793	5088	3583	34.9
1.8.0	30	870	1361	7725	5686	4127	38.5
1.8.1	30	873	1363	7770	5720	4138	38.6
1.8.2	59	1090	1719	9879	7248	5329	38.3
1.8.3	59	1093	1723	9922	7275	5351	38.4
1.8.4	59	1094	1725	9934	7275	5354	38.3
1.9.0	60	1116	1875	10,103	7475	5522	37.3

version 1.3.0 to 1.4.0) and were closely followed by increases in the number of packages.

The distribution of data or functionality among classes, on the other hand, is stable over different releases. The mean number of attributes per class is 1.5 (median = 1.5, standard deviation = 0.06, minimum = 1.4, and maximum = 1.7), the mean number of methods per class is 8.7 (median = 8.8, standard deviation = 0.3, minimum = 8.0, and maximum = 9.1), and the mean number of public methods per class is 6.7 (median = 6.7, standard deviation = 0.18, minimum = 6.2, and maximum = 7.0). A class depends on average on 4.7 other classes (median = 4.7, standard deviation = 0.28, minimum = 3.5, and maximum = 4.9) and this number has also been rising since the first release of the system. Finally, class elegance has also deteriorated since the first releases of the system, denoting that a few packages concentrate a large number of classes and showing a lack of equilibrium on the distribution of classes throughout the packages that form the system. This data is presented in the box-plots in Fig. 5, which show the distribution of the number of classes per package (top-left chart), the number of attributes per class (top-right chart), the number of methods per class (bottom-left chart), and the number of public methods per class (bottom-right chart) for each release of Apache Ant. Outliers are presented as small dots over the top whiskers of the box-plots or below their lower whiskers. These values are either larger than the third quartile plus one inter-quartile range (IQR) or smaller than the first quartile minus one IQR.

Table 4 presents non-parametric Spearman rank-order correlation coefficients between the number of classes and all other size and complexity measures shown in Table 3. A non-parametric correlation measure is used because the number of classes over the releases of Apache Ant, being defined in the integer $[0,+\infty[$ interval, could not resemble a normal distribution. A small p-value for the Shapiro–Wilk normality test (p-value = 0.09) applied to the dataset comprising the 24 observations of number of classes confirmed the departure from the Gaussian curve. All metrics are strongly and positively correlated to the number of classes, thus carrying the same information about variations in version size and complexity. Therefore, we have selected a single metric – the number of classes – for further analysis. In regards to this metric, not all released versions of Apache Ant present a similar increase in size or complexity. Major percentile increases from an immediate former version (greater than or equal to a 10% increase) can be seen for all major versions (but version 1.3.0, which added only 8% new classes to version 1.2.0) and for version 1.8.2. Thus, in concentrating our analysis on the most important releases, the next subsections will take only the 9 major versions and version 1.8.2 into account. We have included version 1.3.0 in our selected versions because all other major versions were included.

4.2. Change dispersion metrics

We have selected change dispersion metrics to evaluate whether signs of the *Shotgun Surgery* code-level smell and the *Scattered Functionality* architecture-level smell could be found during the evolution of Apache Ant. For the purpose of our analysis, we consider that *Shotgun Surgery* is observed whether implementing a change requires changing a large number of classes, instead of a single class. Similarly, mapping a package to the concept of a software component (see Section 2.2), *Scattered Functionality* is observed whether implementing a change requires changing classes from a large number of different packages, instead of a single package incorporating the functionality related to the required change.

The source for both smells is the set of changes made to the system. To measure the extent to which these smells have affected Apache Ant, we first have to determine which changes were made to the system. Then, we must count the number of classes and packages involved in implementing each change. Identifying changes is not easy as the reason driving these changes may not have been explicitly documented: some changes were recorded in the issue tracking system, in response to demands and to correct errors found by users, but other changes may have been due to errors found by developers, to the addition of new features, refactoring of the source-code, and so on.

Thus, we rely on a proxy to the real changes made to the system: we have collected information from the version control system and treated each commit as a change. However, commits to a version control system are related to the implementation of

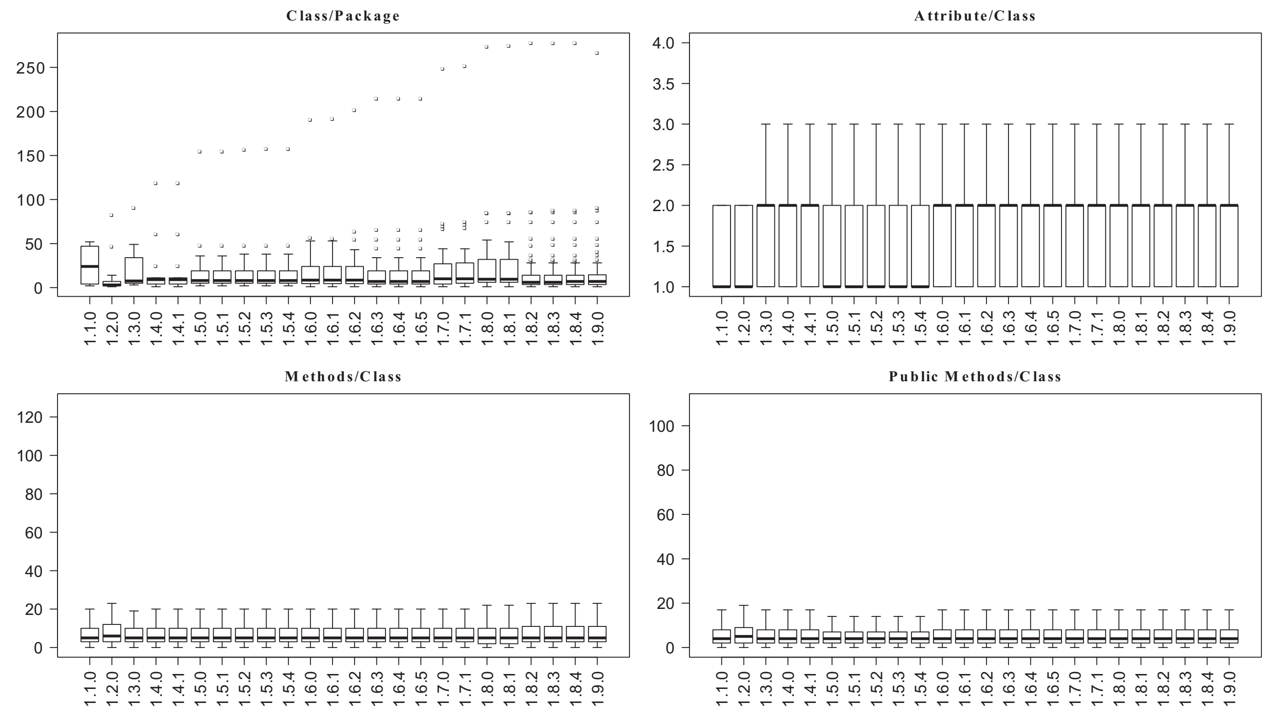

Fig. 5. Box-plots for the distributions of the number of classes per package, attributes per class, methods per class, and public methods per class over the 24 versions of Apache Ant.

Table 4
Correlations between selected size/complexity measures and the number of classes. As all correlations are strong and positive, all metrics carry the same information regarding system size and complexity.

Correlated metric	Packages	Attributes	Methods	Public methods	Dependencies	Class elegance
Classes	0.9914	0.995	0.9986	0.9986	0.9962	0.9729

a change, not to its causes, rationale, or reason. Due to this relation, commits may not be perfectly aligned with changes in the following scenarios: (i) a commit may incorporate changes to the source code which are related to more than one change in system functionality and (ii) many commits may be required to send the edited versions of all classes involved in a single change to the version control system. Nevertheless, commits are the most reliable source of information on changes available in public documents describing Apache Ant. Therefore, to measure the dispersion of changes made to the system we count the number of classes and packages involved in each commit. A large number of commits involving a large number of classes would show signs of *Shotgun Surgery*. A large number of commits involving classes residing on a large number of different packages would show signs of *Scattered Functionality*.

Table 5 provides information on the number of classes and packages involved in commits made to Ant's version control system over time. The *Commits* column shows the number of commits made before releasing each version of the system. A total of 6094 commits were submitted to the version control system from January 13, 2000 to March 10, 2013. Each value of the *Commits* column counts the number of commits from the last version released before the version under analysis (commits that ultimately differentiate the version under analysis from the prior version of the system) to the last version released as part of the version under analysis (commits made as part of the current version and the minor ones that followed it). For instance, the number of commits related to version 1.8.2 in Table 5 shows those made from version 1.8.1 to 1.8.2 (109 commits, 1.8.1 being the last version before 1.8.2), then to 1.8.3 (91 commits), and finally to 1.8.4 (7 commits, 1.8.4 being the last version treated as part of version 1.8.2). As it can be seen, versions 1.5.0–1.7.0 concentrate most commits (69% of the commits registered for the system).

4.2.1. Shotgun surgery

The *Single Class Commits* column in Table 5 counts the number of commits submitting changes to the source code of a single class, while the % *Single Class Commits* column shows this number as a percentile of the total number of commits for that version. On average, 60% of the commits involved a single class over the course of Ant's evolution. Also, the percentile number of single class commits seemed to grow from version to version, thus indicating that most changes made to the system involved editing a single class. This is an important indication that change dispersion was under control, despite the huge changes in the composition of the development team.

The *Classes per Commit* column shows the mean number of classes involved in commits and its standard deviation (after the ± sign). The distribution of this measure over the releases of Apache Ant is presented in the box-plots in Fig. 6. These box-plots are presented with and without outliers. Only outliers larger than the third quartile happen in our analysis. Since a few commits concentrate a huge number of classes (for instance, a single commit involved more than 750 classes in version 1.7.0), presenting outliers reduces the vertical space available to show the central part of the distributions (between the first and third quartiles). Thus, we

Table 5
Number of commits and their distribution over classes and packages comprising the system.

Version	Commits	Single class commits Δ	% Single class commits Δ (%)	Classes per commit ∇	Single package commits Δ	% Single package commits Δ (%)	Packages per commit ∇
1.1.0	172	100	58	2.4 ± 4.6	136	79	1.2 ± 0.5
1.2.0	242	156	64	2.0 ± 2.2	195	81	1.3 ± 0.6
1.3.0	212	135	64	3.1 ± 12.2	176	83	1.4 ± 1.7
1.4.0	348	230	66	2.1 ± 3.1	287	82	1.3 ± 1.1
1.5.0	1704	1087	64	4.6 ± 21.7	1342	79	1.8 ± 3.6
1.6.0	1296	943	73	4.1 ± 36.3	1099	85	1.5 ± 3.9
1.7.0	1238	897	72	2.9 ± 21.8	1034	84	1.5 ± 2.4
1.8.0	611	436	71	2.1 ± 4.3	499	82	1.4 ± 1.6
1.8.2	207	161	78	2.1 ± 6.7	175	85	1.4 ± 1.8
1.9.0	64	36	56	7.7 ± 22.0	47	73	2.5 ± 5.8

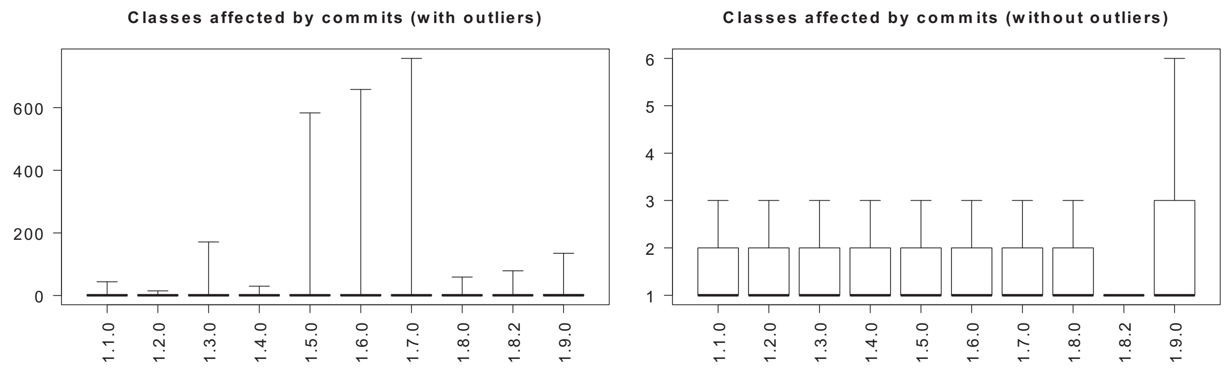

Fig. 6. Box-plots for the number of classes involved in commits for the 24 releases of Apache Ant. Notice that a few commits have involved a large number of classes, widening both the mean and standard deviation for this measure.

present box-plots with outliers to bring forth these extreme values and without outliers to highlight the short IQR presented by the distributions.

Despite the growing number of single class commits, the mean number of classes involved in commits showed a growing trend from version 1.1.0 to 1.6.0. Also important, the standard deviation for the data points of this growing trend also rose over time, denoting that a few changes may have involved a large number of classes (much larger than the already inflated mean), as depicted in the box-plots with outliers in Fig. 6. This number fell from version 1.7.0 to 1.8.2, possibly due to refactoring efforts, and showed a strong increase for version 1.9.0. However, we may not take version 1.9.0 into consideration because, being the last version on the scope of our analysis, it has few changes (64 commits against a mean number of 609 commits between major releases) that may have been casually spread over many classes. We have compared the number of classes involved in commits between pairs of consecutive versions using the non-parametric Wilcoxon–Mann–Whitney test with 95% significance level and found differences only between versions 1.5.0/1.6.0 (p-value < 0.001) and 1.8.0/1.8.2 (p-value = 0.05).[7] A non-parametric inference test was used because all observations were integer numbers (leading to discrete distribution on the domain of frequency) greater or equal to one and, thus, could not resemble a normal distribution. Small p-values collected from the application of the Shapiro–Wilk normality test (p-value < 0.001) upon all datasets asserted that the data might not be considered to fit a normal curve.

[7] Since the datasets about classes involved in commits for version 1.8.0 and 1.8.2 were used twice in these analysis, the number of classes involved in commits may not be considered significantly different from version 1.8.0 to 1.8.2 if an alpha-adjustment Bonferroni test is applied upon the data.

Overall, the data in Table 5 does not tell a straight story on the presence of the *Shotgun Surgery* smell: the number of commits involving a single class is large (lacking the symptom of each change affecting many classes) and the mean number of classes changed in each commit has increased up to version 1.6.0 (showing signs of the smell), but the latter has declined in recent versions (possibly showing the positive results of refactoring efforts).

4.2.2. Scattered functionality

Regarding the number of packages involved in commits, the *Single Package Commits* column in Table 5 counts the number of commits involving classes from a single package, while the *% Single Package Commits* column shows this number as a percentile of the total number of commits for the version. On average, 81% of the commits submitted to Ant's version control system involved a single package. This is a second indication that change dispersion was under control: even if multiple classes are being edited as part of a commit, they usually come from the same package, thus limiting the dissemination effects of the change at package level.

The *Packages per Commit* column shows the mean number of packages involved in commits and its standard deviation (after the ± sign). The distribution of this measure over the releases under analysis is presented in the box-plots in Fig. 7. Except for version 1.9.0, the mean number of packages involved in commits remained below the mark of two packages per commit throughout the selected versions of Apache Ant. The standard deviation shows a steep increase from version 1.1.0 to 1.6.0, returning to lower values in later versions (the same behavior shown by the mean number of classes involved in commits).

We have compared the number of packages involved in commits between consecutive versions using the non-parametric Wilcoxon–Mann–Whitney test with 95% significance level and found differences only between versions 1.5.0 and 1.6.0

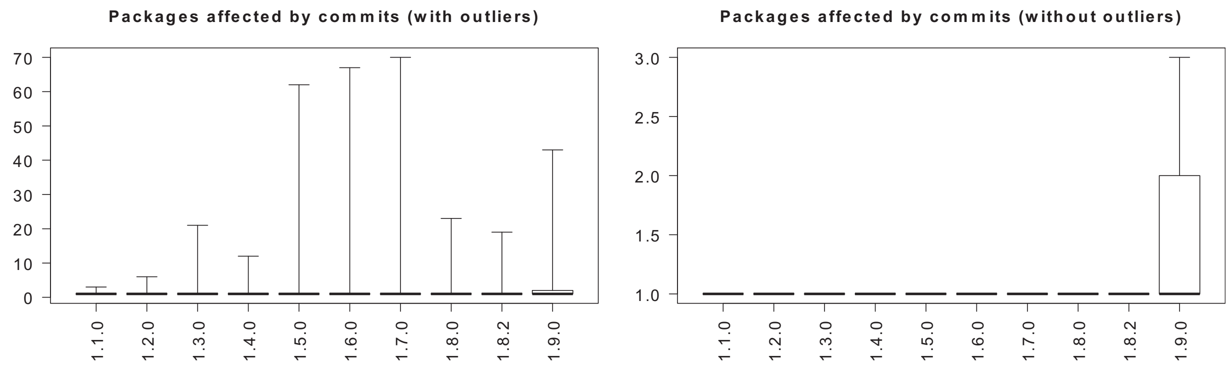

Fig. 7. Box-plots for the number of packages involved in commits for the 24 releases of Apache Ant. Again, a few commits involved a large number of packages, increasing both the mean and standard deviation for this measure.

Table 6
Coupling, cohesion, and compound metrics for the selected versions of Apache Ant.

Version	CBO ∇	AFF ∇	EFF ∇	LCOM ∇	MQ Δ	MF Δ	EVM Δ	CS Δ
1.1.0	0.50	11.8	3.8	0.67	1.79	0.45	−1867	−466
1.2.0	2.23	12.1	7.8	0.74	4.11	0.32	−4011	−308
1.3.0	2.00	21.5	8.6	0.83	4.02	0.50	−4843	−605
1.4.0	2.15	20.2	9.0	0.83	5.83	0.45	−8443	−649
1.5.0	3.33	22.2	13.2	0.81	9.85	0.47	−13,471	−641
1.6.0	3.75	25.4	15.7	0.80	10.94	0.46	−21,519	−896
1.7.0	4.72	33.8	19.9	0.83	11.77	0.41	−39,057	−1346
1.8.0	4.93	38.5	22.1	0.81	12.56	0.42	−49,381	−1646
1.8.2	4.58	26.1	17.5	0.79	21.37	0.36	−51,615	−874
1.9.0	4.65	26.4	17.7	0.79	22.05	0.37	−49,639	−827

(p-value < 0.001). As in our analysis for the number of classes per commit, a non-parametric inference test was used to determine whether there might be significant differences on the number of packages involved in commits because all observations were integer numbers, greater or equal to one. Small p-values collected from applying the Shapiro–Wilk normality test (p-value < 0.001) on datasets for all releases under analysis confirmed that none of these datasets could fit a normal distribution. Therefore, the data in Table 5 indicates that the *Scattered Functionality* smell had a limited effect over the evolution of Apache Ant, despite the huge demand for new features in its early years and its having a high turnover on the development team.

4.3. Coupling and cohesion metrics

We have used 6 different metrics to examine coupling and cohesion over the course of the selected versions of Apache Ant, 3 metrics being for coupling, 1 metric for cohesion, and 2 compound metrics that take both coupling and cohesion into consideration. Table 6 presents mean values for these metrics over the selected releases of Apache Ant, as well as values for the Modularization Factor and Cluster Score components of the compound metrics (see Section 4.3.3). The complete distribution for these metrics is presented in the box-plots depicted in Fig. 8.

4.3.1. Coupling

The first metric is an extension of the *Coupling between Objects* (CBO) metric for packages. CBO was introduced by Chidamber and Kemerer [15] and calculates the coupling of a class A by counting the number of classes upon which A depends to implement its features. Its extension to measure package coupling relies on usage dependencies between packages, which we derive from the concept of usage dependencies between classes: a package A depends on a package B if at least one class residing in package A depends on at least one class residing in package B to perform its duties. The CBO for a given package A is then calculated as the number of packages conveying classes upon which classes residing in package A depend to implement their features. Common sense design rules state that CBO should be minimized, that is, classes and packages should be designed to depend on as few other classes or packages as possible.

Afferent coupling (AFF) is a second coupling measure that is computed by counting the number of classes from outside a package A that depend on at least one class from package A. A package with high afferent coupling has strong responsibility as part of a system, since many classes depend on it to deliver the features associated to them.

Efferent coupling (EFF) is a third coupling measure that is computed by counting the number of classes from outside a package A upon which classes residing on package A depend to implement their features. The smaller the efferent coupling for a given package, the more independent this package is from the rest of the system. Thus, the easier it will be to test and reuse this package in other systems.

The *CBO* column in Table 6 presents the mean CBO for packages comprising each version of the system, while the *AFF* column presents the mean value for afferent coupling, and the *EFF* column presents the mean value for efferent coupling. These values are strongly and positively correlated, showing that, at least for Apache Ant, different coupling views are consistent with each other. The Spearman's rank-order correlation coefficient between AFF and EFF is 0.9879, between AFF and CBO is 0.9515, and between EFF and CBO is 0.9636. These values show a clear growing trend over the course of Ant's evolution, peaking at version 1.8.0.

The observed trend for coupling is supported by the data provided in Table 7, which shows that the growing means for coupling presented in Table 6 are not driven by a few, specific packages with large increases in coupling, but by a large number of packages

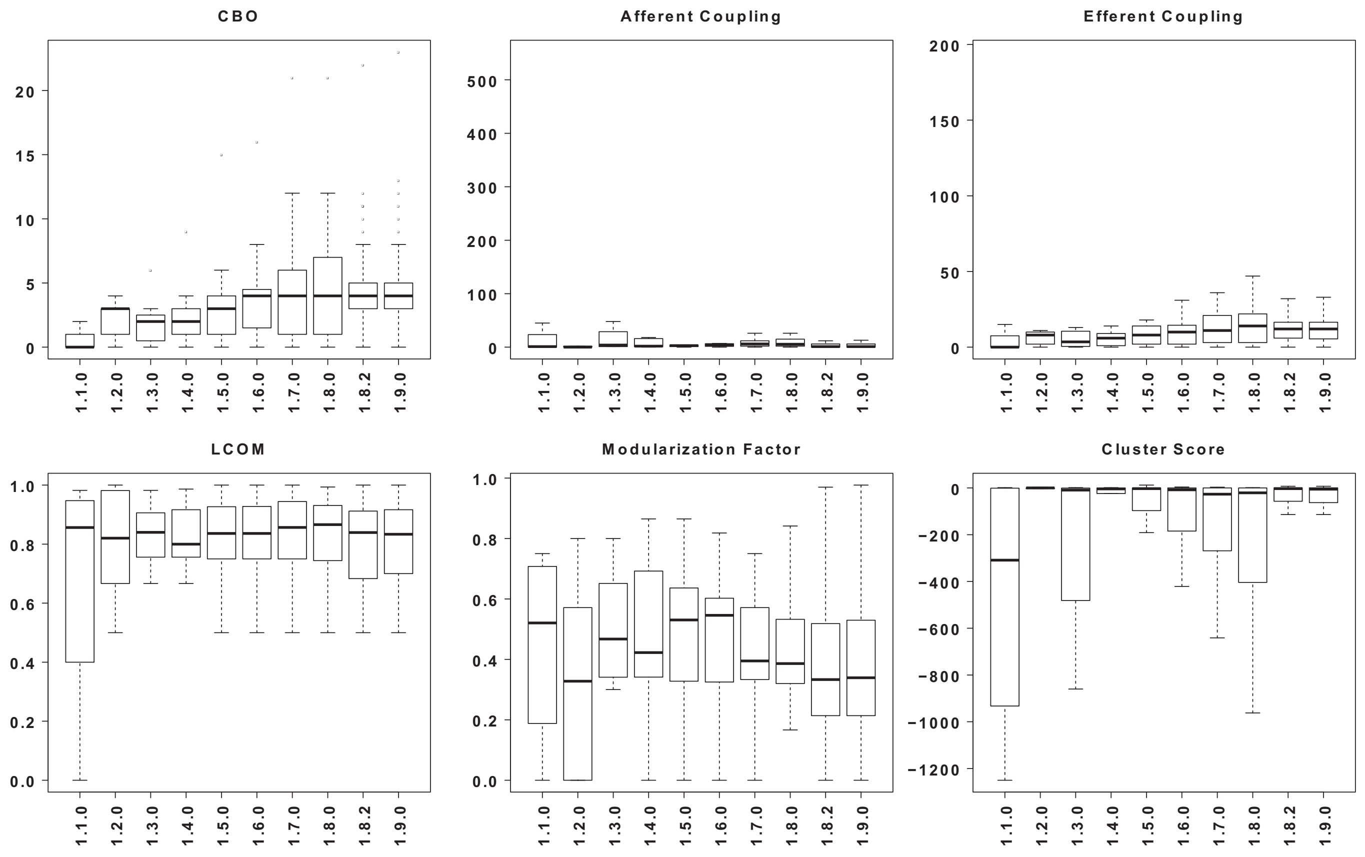

Fig. 8. Box-plots for coupling, cohesion, and compound metrics over the 10 selected releases of Apache Ant.

Table 7
Proportion of the selected packages increasing, maintaining, and reducing their coupling from one version to the subsequent one.

	CBO			Afferent coupling			Efferent coupling		
	Growth (%)	Stability (%)	Reduction (%)	Growth (%)	Stability (%)	Reduction (%)	Growth (%)	Stability (%)	Reduction (%)
1.1.0 → 1.2.0	67	33	0	67	33	0	67	33	0
1.2.0 → 1.3.0	60	40	0	40	40	20	60	40	0
1.3.0 → 1.4.0	25	75	0	50	50	0	62	38	0
1.4.0 → 1.5.0	58	42	0	58	42	0	67	33	0
1.5.0 → 1.6.0	38	62	0	48	48	4	52	43	5
1.6.0 → 1.7.0	48	48	4	61	35	4	61	26	13
1.7.0 → 1.8.0	22	78	0	39	57	4	61	30	9
1.8.0 → 1.8.2	13	87	0	48	48	4	35	57	8
1.8.2 → 1.9.0	17	83	0	39	61	0	30	70	0

suffering increases in their coupling on the same version. We have picked 23 packages that participated in 6 or more versions (out of the 10) selected for Apache Ant and measured their CBO, afferent, and efferent coupling over the selected versions. We then counted how many of these packages had their coupling increased, maintained, or reduced from one version to the subsequent one. As outlined by the growing mean values in Table 6 almost no package had its coupling reduced from one version to another, except in transitions from version 1.2.0 to 1.3.0 (for afferent coupling) and from version 1.6.0 to 1.7.0 (for efferent coupling). Except for those, all other transitions had no more than 10% of their packages with reduced coupling and most packages show an increase in coupling.

4.3.2. Cohesion

The next metric is an extrapolation of the *Lack of Cohesion of Methods* (LCOM) metric to determine the cohesion of a software package. LCOM was originally proposed by Henderson-Sellers [31] and calculates the lack of cohesion in a class according to the common usage of attributes by its methods – the more methods accessing more attributes, the smaller (more desirable) the value for this metric is. LCOM is defined in the [0, 1] interval and values close to zero denote greater cohesion, that is, great synergy between the methods and attributes comprising a class. We have adapted the concept for package cohesion: instead of methods referencing attributes, we considered the number of classes from a given package that depend on other classes residing in the same package. A lack of cohesion for packages is observed when most classes in a package do not depend on other classes from the same package, leading to a large LCOM value.

The *LCOM* column in Table 6 presents the mean LCOM for packages comprising each version of the system. Besides version 1.1.0, mean LCOM is stable over the selected versions and correlates positively, though weakly, with CBO ($\rho = 0.2$). This data implies that packages have shown stable but weak cohesion over time.

Table 8
Proportion of the selected packages in increasing, maintaining, and reducing their cohesion, modularization factor, and cluster score from one version to subsequent one.

	LCOM			Modularization factor			Cluster score		
	Growth (%)	Stability (%)	Reduction (%)	Growth (%)	Stability (%)	Reduction (%)	Growth (%)	Stability (%)	Reduction (%)
1.1.0 → 1.2.0	33	34	33	33	34	33	0	33	67
1.2.0 → 1.3.0	80	20	0	60	40	0	0	40	60
1.3.0 → 1.4.0	63	38	0	13	37	50	0	38	62
1.4.0 → 1.5.0	42	42	17	0	25	75	8	42	50
1.5.0 → 1.6.0	48	52	0	29	38	33	0	52	48
1.6.0 → 1.7.0	74	26	0	43	22	35	4	22	74
1.7.0 → 1.8.0	48	39	13	35	17	48	9	39	52
1.8.0 → 1.8.2	22	69	9	13	44	43	0	70	30
1.8.2 → 1.9.0	26	57	17	26	44	30	13	57	30

However, it is interesting to see that cohesion has remained almost unchanged even in periods of growing coupling. As these metrics are theoretically competing, one might expect a negative and strong correlation between them.

Table 8 uses the same structure of Table 7. Its LCOM columns show the proportion of the 23 packages participating in at least 6 releases of Apache Ant which have shown growth, stability, and reduction on cohesion over different releases. Cohesion increased slightly from one version to the next, though some packages show desirable reductions in LCOM[8] for specific versions (1.2.0, 1.5.0, 1.8.0, and 1.9.0).

4.3.3. Compound metrics

Coupling and cohesion are competing metrics. If an architect aims to maximize package cohesion, one might consider putting every class in a separate package, so that the objective of the package becomes clear and strictly related to that of the class. On the other hand, if the architect aims to minimize package coupling, one might consider putting all classes into a single package so that no dependency between packages would be possible. Anyway, none of these solutions would seem reasonable in a large software development scenario, in which developers require ways to group related classes in packages to make the system easier to comprehend and to restrict different types of changes to different parts of the system. Thus, we need some balance between coupling and cohesion. The MQ and EVM compound metrics, shown in Section 2.1, aim for that balance and supplemented our analysis. Larger MQ or EVM represent better designs.

The MQ and EVM columns in Table 6 show, respectively, the modularization quality and the EVM measure for the whole system. MQ has steadily grown from version 1.1.0 to 1.8.0 and almost doubled from the later to version 1.8.2. EVM, on the other hand, has substantially decreased over the selected versions. However, the absolute value of these metrics grows with the number of packages and comparing their values over different versions may take the growth in the number of packages into consideration. Thus, we present the mean modularization factor presented by packages of a given version in the *MF* column, and the mean cluster score of these packages in the *CS* column. The modularization factor shows an oscillating behavior (mean = 0.43, median = 0.45, standard deviation = 0.05), growing and diminishing from version to version, while MQ has shown a consistent growth trend. The cluster score, on the other hand, persists in a negative and decreasing trend (mean = −898, median = −876, standard deviation = 345), now peaked in version 1.8.0, being reduced to half of its absolute value in version 1.8.2.

The last six columns in Table 8 present the proportion of the 23 selected packages showing growth, stability, and reduction trends for the compound metrics over the 10 releases of Apache Ant. Modularization factor shows stability, with a few more packages showing a decrease in this metric (39% over all versions) than stability (33%) or growth (28%) over the selected versions. Cluster score, on the other hand, shows a tendency to decrease or stagnate, with few packages increasing their score over time.

5. Applying search-based module clustering to Apache Ant

Our former analysis shows that Apache Ant has evolved from a very simple architecture, with components having clear purposes and communication channels, to a complex grid of components and dependencies. Over the course of this evolution, the average change to Ant's source code affected a growing number of classes, until the system was refactored to contain such growing change dispersion. Moreover, all versions of Apache Ant presented growing coupling and low cohesion. With this scenario as a background, we now address the first question underlying our research: could a search-based reorganization of the distribution of Ant's classes to packages (software module clustering) reduce the number and dispersion of dependencies among packages and bring the system back to its original simplicity?

As presented in Section 2.1, the most commonly used mono-objective optimization models for the software clustering problem rely on graphs describing dependencies amongst classes and optimize (maximize) one of two alternative fitness functions: Modularization Quality (MQ) and EVM. Modularization Quality aims for a balance between coupling (related to the number of dependencies between packages, to be minimized) and cohesion (related to the number of dependencies within packages, to be maximized), while EVM aims primarily at maximizing cohesion and searching for module distributions in which packages consist of classes with many dependencies amongst each other, despite their connections to classes residing on other packages.

We have optimized Apache Ant's version 1.9.0 in a search-based perspective according to both EVM and MQ, to streamline its module dependency structure and observe whether the search process might reverse the signs of architectural erosion shown in Fig. 4. Fig. 9 presents packages dependency charts for the solutions found after executing the search based on EVM (Fig. 9a) and MQ (Fig. 9b).

We used a Hill Climbing search with random restarts, consuming a fitness function evaluation budget of $1000N^2$, where N is the number of classes in the program (1116 classes for version 1.9.0). Given a software system with N classes, the search started by creating N clusters and randomly assigning each class to a cluster. A solution was represented as a vector with one entry per class, each entry containing an integer number in the $[0, N-1]$ interval indicating the cluster to which the related class is associated. Solution fitness was calculated, stored as the best solution found so far, and the main loop of the search followed. The main loop attempted to find solutions with better fitness by iteratively moving a single

[8] LCOM represents "lack of cohesion". So, reducing LCOM means to increase cohesion.

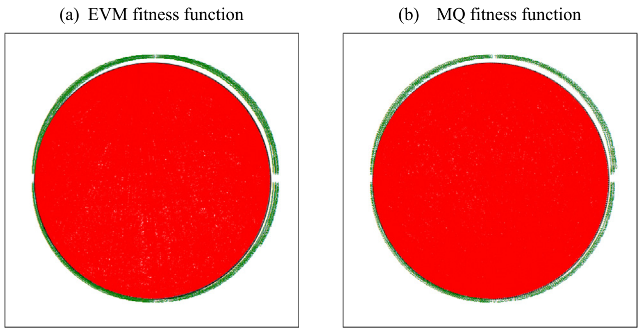

Fig. 9. Optimized architecture for Apache Ant version 1.9.0.

class to a distinct cluster. The first class was selected and a move to any cluster other than the currently occupied one was evaluated. After all clusters had been evaluated for the first class, the search followed a similar procedure for the second one, and the third one, and so on. Whenever a higher-fitness solution was found, the new solution replaced the best known so far, and the main loop repeated its trials from the first class. If no movement could improve fitness, a new, random solution was created and the main loop restarted from that solution. The search also stopped after the predefined budget of fitness evaluations was consumed.

To take into account the stochastic component related to the generation of the initial and restart solutions, we have performed 30 independent optimization turns for each fitness function. Fig. 9 provides a random sample of these turns (they do not differ much in terms of number of dependencies among packages, as it will be discussed later). This was a large-scale optimization process, considering that published papers addressing the software module clustering problem usually report on instances up to 300 classes or modules (that is, less than one third of Ant's version 1.9.0 size). Each optimization turn took about one hour[9] in a dedicated 2.6 GHz Core i7 4 GB RAM machine.

5.1. Size metrics

As it can be seen in both sides of Fig. 9, search-based optimization did not provide less complex package distributions than those provided in Fig. 4. Instead, a large number of packages were created for both fitness functions and a huge number of dependencies exists amongst these packages. Each dependency cannot be clearly observed in the charts, due to the number of crossing lines. The distribution for the number of packages, along with other properties discussed in the next paragraphs, for each fitness function over the 30 independent optimization turns is shown in Fig. 10. We have also executed the optimization process departing from a solution that replicates the distribution of classes into packages from the design of version 1.9.0 (without using random restarts) and optimization results were similar in terms of number of packages.

The mean number of packages for EVM-optimized versions is 605.8 (median = 606, standard deviation = 5.9, minimum = 594, maximum = 617), ten times larger than version 1.9.0. The large number of packages leads to a smaller class concentration: we observe a mean number of 1.8 classes per package (median = 1.8, standard deviation = 0.02). It also leads to smaller class elegance (mean = 1.4, median = 1.4, standard deviation = 0.01). While a smaller number for class elegance is desirable, denoting equilibrium in the number of classes contained in each package, it is also a by-product of having a large number of packages, most of them with no more than 3 classes. In the specific case of EVM-optimized versions, we have on average 352 packages with a single class, standing for about 60% of the total number of packages and, after all, losing their purpose of grouping together related classes.

Similarly, the mean number of packages for MQ-optimized versions is 390 (median = 390, standard deviation = 4.6, minimum = 378, maximum = 397), still more than six times larger than the number of packages observed in version 1.9.0. Mean class concentration is 2.9 classes per package (median = 2.9, standard deviation = 0.03) and class elegance is 1.5 (median = 1.5, standard deviation = 0.02). Differently from EVM-optimized versions, the number of single class packages on MQ-optimized versions is just a small fraction of the total number of packages (mean = 6.8, median = 6.0, standard deviation = 2.8).

5.2. Change dispersion metrics

Table 9 compares version 1.9.0 with its optimized versions in terms of package-level change dispersion metrics. Class-level change dispersion metrics are unaffected by reorganizing classes into packages, but package-level dispersion might be affected, since classes changed together in given commits may become part of the same package or may be separated in distinct ones. We assume that functionality distribution over the classes comprising the system would remain the same after optimization findings are reflected in the class-to-package distribution used in the system: over the course of many releases, a different class-to-package distribution might encourage developers to move parts of the source-code from one class to another, thus influencing the specific classes committed to the version control system as part of the implementation of the required changes.

The data provided in Table 9 was created by determining the number of packages affected by 64 commits made from version 1.8.2 to 1.9.0, taking into account how classes were distributed among packages for the original version (first row, replicated from the last row in Table 5) and the optimized ones. It shows the mean and standard deviation for the number of commits involving a single package, as well as the mean and standard deviation for the number of packages involved on each commit. The complete distribution of these data is presented in the box-plots depicted in Fig. 11.

The number of commits affecting a single package has strongly decreased in both optimized versions, meaning that a large number of commits that affected a single package in the original version would have changed classes in more than one package in the optimized ones. The number of single package commits for EVM-optimized versions is significantly different from the number of single package commits in the original version (p-value < 0.001, for the non-parametric Wilcoxon–Mann–Whitney test). The same happens while comparing MQ-optimized versions with the original 1.9.0 (p-value = < 0.001). Finally, the number of single package commits from both optimized versions is also statistically different from each other (p-value < 0.001). Again, non-parametric tests are used because the data strongly departs from a normal distribution, as confirmed by small p-values (<0.001) for the Shapiro–Wilk test for both optimized versions.

In the same direction, the mean number of packages involved in each commit has more than doubled from the original version to the optimized ones. p-Values smaller than 0.001 are found while comparing the series of mean number of packages per commits,

[9] Our previous paper [8] reports that each optimization turn took about 4 days. However, extensive improvements were made to the algorithms supporting the calculations performed during the optimization, allowing us to reduce the time of each turn to a single hour. The most important improvement was recalculating only part of the modularization factors and cluster scores used to calculate MQ and EVM: in the former paper, we recalculated all scores and factors for all clusters of any candidate solution, while in the implementation from which results reported here were collected we stored previously known scores and factors, recalculating these values only for clusters affected by the repositioning of modules. This approach was formerly presented for MQ by Mitchell [47] as the "Incremental TurboMQ".

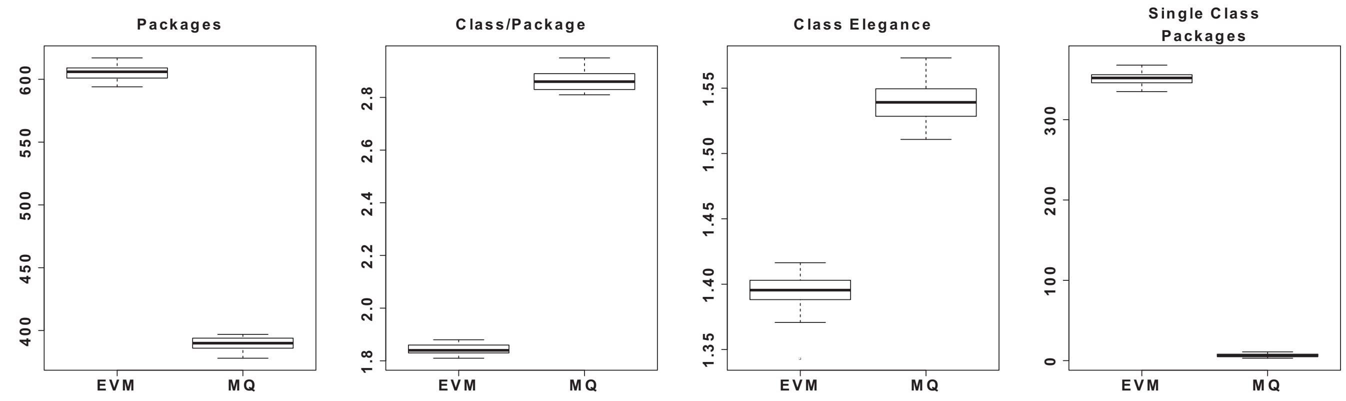

Fig. 10. Box-plots for the number of packages, class per package ratio, class elegance, and single class packages for optimized versions of Apache Ant 1.9.0.

Table 9
Package-level change dispersion metrics for the original and optimized versions of Apache Ant. All metrics have deteriorated while comparing the original version to the optimized ones.

Version	Single package commits Δ	% Single package commits Δ	Packages per commit ∇
Original v1.9.0	47	73	2.5 ± 5.8
EVM-optimized	38.5 ± 1.1	60	5.3 ± 15.1
MQ-optimized	37.5 ± 0.5	59	5.6 ± 16.3

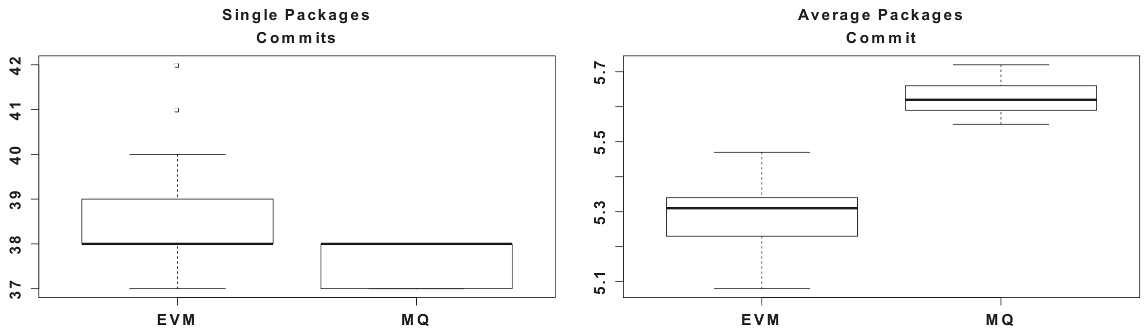

Fig. 11. Box-plots for number of commits affecting a single package and the mean number of packages involved on each commit for optimized versions of Apache Ant 1.9.0.

Table 10
Coupling and cohesion metrics for the original and optimized versions of Apache Ant.

Version	CBO ∇	AFF ∇	EFF ∇	LCOM ∇	MQ Δ	EVM Δ
Original v1.9.0	4.65	26.4	17.7	0.79	22.05	−49,639
EVM-optimized	4.48 ± 0.14	5.64 ± 0.19	5.88 ± 0.06	0.41 ± 0.01	73.62 ± 1.14	773.9 ± 7.89
MQ-optimized	7.78 ± 0.15	10.49 ± 0.18	8.68 ± 0.09	0.46 ± 0.01	101.63 ± 1.20	289.1 ± 15.27

both considering each optimized version against version 1.9.0 or one optimized version against the other. Thus, from a change dispersion point-of-view, optimization has not improved the system design towards a one-to-one relation between related sets of changes and related sets of classes: a large number of changes would have to affect the source code of classes in a larger number of packages.

5.3. Coupling and cohesion metrics

Table 10 summarizes the values collected for coupling and cohesion metrics for version 1.9.0 and their mean and standard deviation on both optimized versions of Apache Ant, while the complete distributions for these metrics is presented in Fig. 12. Almost all coupling metrics have been improved by optimization, the only exception being the CBO for MQ-optimized versions. In particular, EVM-optimized versions have significantly improved CBO, afferent and efferent coupling. The CBO in EVM-optimized versions is significantly smaller than the original version's CBO (p-value < 0.001), the same holding for the other coupling measures. That is, even being a cohesion-centric fitness function and disregarding coupling, EVM was able to improve all coupling metrics significantly. Cohesion and compound metrics were also significantly improved in EVM-optimized versions.

MQ-optimized versions performed poorly as regards CBO (significantly larger than version 1.9.0's measure), but improved both afferent and efferent coupling – an unexpected result, given the high and positive correlations amongst the three coupling metrics reported in Section 4.3.1. MQ-optimized versions have also improved LCOM, though not as much as EVM-optimized versions.

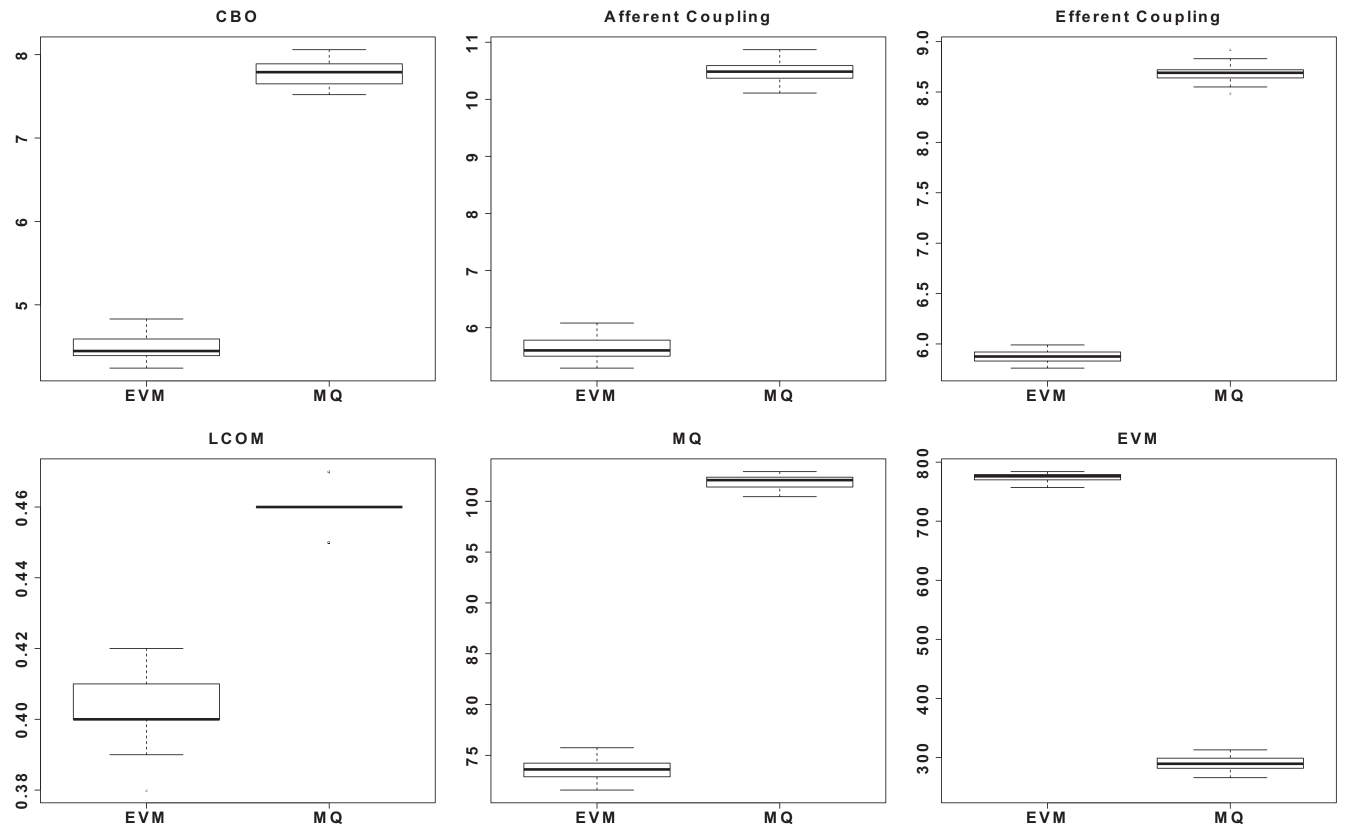

Fig. 12. Box-plots for coupling, cohesion, and compound metrics for optimized versions of Apache Ant 1.9.0.

Both optimized versions have also optimized the contesting fitness function, but values observed for EVM are always higher in EVM-optimized versions and vice versa for MQ.

6. Discussion

The numbers shown in Section 5 suggest that optimizing for coupling and cohesion may not bring forth the original architecture of a software system and may increase the incidence of change-related architectural or code smells. From coupling and cohesion metrics perspectives, both optimized versions seem superior to version 1.9.0, but their design seems far more complex in terms of the number of packages and inter-package dependencies. One might wonder whether the optimized versions take the software to a state in which maintenance, reusability, and testability are easier than in version 1.9.0. The visual impression given by the meshes of dependencies in Fig. 9 do not contribute much to such a perception. Also, the general increase in change dispersion plays against the desirable concentration of changes to a few packages. And finally, most of the packages in the optimized versions showing the highest increase in cohesion and strongest reduction in coupling are lacking their purpose of grouping together related classes by keeping only a few classes, if not a single one.

One major problem with the optimized versions is the excessive number of packages in their design. As a by-product of the larger number of packages, the average package has fewer classes than version 1.9.0. Those few classes within each package are indeed strongly interconnected, showing a large number of dependencies amongst each other and leading to a decrease in the average LCOM. The higher number of packages also decreases CBO and other coupling metrics on average: there are so many packages and so few classes in each package that the mean number of dependencies leaving or entering each package is relatively small, reducing afferent and efferent coupling, although the total number of dependencies crossing package borders has increased. Optimization has, thus, improved cohesion and coupling metrics by spreading the classes comprising the system into a large number of very small packages and, as a result, increased change dispersion.

Overall, the optimized versions seem to be ‘cheating’ the software engineer: they are ‘improving the design’ from the structural metrics standpoint, but these changes do not correspond to the expectations of software developers or do not lead to other benefits, such as recovering the design-time architecture or sand-boxing changes to a few packages. Optimized versions followed the tenets described in the model, but the results produced by the optimization process may not represent real improvements in the eyes of the development team.

Then, can we conclude that SBSE or, more specifically, search-based software module clustering is not helpful? Certainly not! First and foremost, without performing the optimization we would never have been able to observe the behavior of even simple metrics while rearranging a large system, such as Apache Ant. Even using simple heuristic techniques, it took 60 h to optimize the program (approximately one hour for each optimization turn and fitness function). Due to the huge number of alternatives, finding (close to) optimal solutions would probably be an unfeasible task in any other way except with the use of heuristic algorithmic optimization.

Moreover, we have to acknowledge that the optimization worked as commanded, though not as expected. EVM and MQ values increased significantly for all optimized versions, but as one might pick the shortest path to a given destination, the optimization process opted to increase the number of packages in order to improve these metrics. Although a large number of packages with few classes may not be considered good design [36], the optimization process leveraged a poor-design aspect that was not covered by the selected metrics underlying our model for a good software design. Like untested source code, the optimization does what the researcher tells it to do, though not necessarily what the researcher wants. Optimization may find unexpected ways to improve the fitness function and knowing these new ways may bring new insights to the researcher on what is exactly wanted when we design software systems. The dogma of increasing cohesion and reducing coupling, which currently underlies much design-thinking in Software Engineering, is being questioned and not for the first time [4]. Software clustering optimization based solely on these metrics is under suspicion, and there are some possible reasons for this. Optimization was made based on structural criteria. However, software engineers usually decide about clustering considering different context variables, such as conceptual cohesion and domain knowledge. This way, current optimization models cannot deal with this holistic view of the problem, given that they do not take into account the same criteria because we do not yet know what the context variables affecting clustering in software projects could be.

Design comes to life much more as a Craft (a tacit activity) than Engineering, with skilled programmers and architects pouring their experience and perceptions on the source code in ways that are not necessarily systematic and repeatable. The researcher comes in, to learn from different sources and organize the forces driving the use of one or another design strategy, thus detailing a knowledge base that will ultimately turn craftsmanship (unsystematic and empirical actions) into "Engineering". Building this knowledge base is modeling in essence, and Search-based Software Engineering (together with experimental techniques) may be used as the setting, as a laboratory to evaluate and register evidence to support this knowledge. Thus, optimization in Software Engineering can be thought as a learning tool, much like simulation has been earlier perceived as a learning tool [62].

Optimization, high-performance computers (such as multi-core processors and GPUs), and scalable environments (such as the "cloud" and grids) allow us to test our Software Engineering models and theories to extremes that could not be tested before. They challenge our theories in the large scenarios where they are most needed. They may show that "optimum solutions" in the model sense may not be optimal from the developers' perspective. Therefore, we have to improve our models. We may need metrics addressing design aspects that are not covered by current metrics. We may need more than metrics. In this sense, SBSE-style optimization can help improving our body of knowledge on software architecture and software design. The research reported in this paper shows an example of how this can be performed in a "destructive" way, that is, using SBSE to find evidence that a given theory may not provide adequate results when taken to an extreme case. We believe this is a valid contribution because finding the limits of a theory represents a step towards the formulation of more complete theories in a Kuhnian [34] perspective of science. Nevertheless, large-scale investigation made possible by the diversity of software projects available in open repositories and the possibility of automated analysis provided by SBSE might allow a more constructive usage of optimization towards evolving a software design theory.

At least for software module clustering, we need models and metrics that better reflect the developer's intention in large software systems. We need to find ways to introduce multiple user perceptions in the optimization process (and there seems to be a lot of gut feeling here!). We may learn from developers using automated learning processes and build models to drive optimization on these uncovered trends. Some steps toward this goal have been taken [27,9], but problems must be taken onto a level where developers can contribute the most and let optimization fill in the gaps. One direction we believe is worth investigating is the use of Dependency Structure Matrices (DSM), as proposed by Sangal and Waldman [58]. The authors, who have also analysed Apache Ant, suggest that dependencies between packages might be shown in the lower triangular part of a square dependency lattice and that dependencies appearing out of this region might be signaled to and corrected (refactored) by developers.

A problem arising from this suggestion is determining which refactoring transformations should be applied to eliminate undesired dependencies and in which order they might be employed. Considering that we might be able to handle large programs with severely mismatched architectures, the number of possible transformations (along with their permutations) may be huge. Supported by a history of successful attempts to use SBSE approaches to select refactoring strategies [49], search-based techniques may be useful to produce permutations of refactoring transformations sought to recover a project towards its reference architecture.

Other key issue is visualization. If SBSE is to be used as a learning tool, we should have proper ways to present the huge amount of data produced during optimization processes. Also, visually comparing different solutions and showing the landscape of search spaces may present interesting research opportunities, both to improve optimization processes, to increase and improve developers' interaction with optimization processes, and to allow researchers to improve their optimization models.

7. Related works

7.1. Software module clustering

The first search-based approaches to the software module clustering problem have extended the work of Mancoridis et al. [42] to explore different strategies to search the solution space provided by the MQ fitness function. Doval et al. [16] present a genetic algorithm to address the software module clustering problem, using MQ as a single-objective fitness function. The genetic algorithm has been found less effective and less efficient than the Hill Climbing search presented in the original work. Later, Mahdavi et al. [41] used a procedure consisting of two sets of Hill Climbing searches to find solutions for the clustering problem. Firstly, 23 parallel and independent Hill Climbing searches are executed upon the MDG representing the software. Next, a partial solution is constructed by fixing the distribution of modules sharing the same cluster in at least α% solutions produced by the initial hill climbers, where α is a parameter defined in the [10%,100%] interval. Afterwards, a second round of Hill Climbing searches is performed to distribute the modules not assigned to clusters during the partial solution construction phase. Finally, the best solution found by the second round of hill climbers is selected as the best solution for the problem. The authors saw improvements in solutions found by the second run of hill climbers when compared to those produced in the first round, particularly for large instances or small instances combined with small values for the α parameter.

The software module clustering problem has also been addressed using a multi-objective optimization perspective. Praditwong et al. [56] presented the first multi-objective approach for the problem by describing two models with 5 objectives each, including MQ. Later, Barros [7] showed that these models could be reduced to 4 objectives without loss, by removing MQ from their formulation. Recently, Kumari and Srinivas [35] proposed a hyper-heuristic (MHypGA) – a heuristic to choose heuristics [14] – to find the best operators for a multi-objective genetic algorithm addressing the software module clustering problem. The authors proposed twelve different operators (two mutations, three crossovers, and two selection operators) which are dynamically selected for each generation based on a weight-adjustment process that learns from their effectiveness in previous generations. After calculating fitness values, the chances of selecting the same operators used in that generation are adjusted according to the quality of the population. According to the authors the algorithm can produce good modular structures with high cohesion and low coupling. In the experimental study reported by the authors, the MHypGA was able to outperform the results produced by [56] in terms of MQ for six real-world instances, consuming a fraction of the processing time to reach the same quality in terms of fitness function value. Abdeen et al. [2] also present a multi-objective optimization approach to maximize cohesion and minimize coupling while changing as little as possible the original design. The optimization merges small packages into larger ones and allows developers to set restrictions upon classes and/or packages that should not be affected by optimization, set a specific package for a class, and limit the maximum number of classes that may be affected by the optimization. The authors have conducted an empirical study to compare their results against their former work, which has focused on cyclic dependencies [1], and found that the most recent approach usually led to superior modularizations.

Other approaches that do not rely on search have also been used to address the software module clustering problem. Schwanke [59] describes a tool (Arch) to support software modularization based on Parnas's information hiding principle [54]. Arch is a graphical and textual "structure chart editor" which provides a clustering algorithm to group related procedures into modules. It also criticizes such distributions, identifying individual procedures that apparently violate good information hiding principles. Based on some experiments, the author concludes that the tool is promising to support the clustering problem. Muller et al. [48] use a reverse engineering approach and tool (Rigi) to retrieve high-level abstract representations of a software system. Rigi consists of a distributed graph editor and a parsing system to support the software engineer in the task of decomposing a system in subsystems. It discovers related components and dependencies, layered subsystem structures, and interfaces among subsystems based on coupling and cohesion between modules. The tool was used in several projects, some with up to 57 KLOC of COBOL code. The authors conclude that finding proper subsystem structures is an art and cannot be fully automated, though their tool can help in the task by providing a set of clustering methods to support subsystem decomposition. Their conclusions are supported by Glorie et al. [26] who have applied the Bunch tool in five scenarios and found several limitations for automated software clustering, including generating different clustering schemas in distinct runs based on the same input, generating clusters which significantly differ in size (number of lines of code), contain a single module, or are not recognized by domain experts. These results are closely related to our findings on regard of automated clustering.

Interactive search has also been used to find solutions to software clustering that make sense to developers. Koschke [33] conducted an experiment to evaluate different approaches to detect atomic components that might be viewed as cohesive logical modules, including analysis of the domain model, the dataflow model, method-call relationships, data sharing, coupling, graph and concept representations, as well as collecting information and perceptions from the maintainer. The overall result was that none of the automated or semi-automated techniques had the required precision, leading the author to propose a method in which human and computer interact to detect atomic components. Hall et al. [27] present software module clustering as a constraint satisfaction problem on which constraints are introduced by designers as corrections to a baseline module distribution. Bavota et al. [9] introduce an interactive genetic algorithm for software clustering that is driven both by fully automated measures (such as modularization quality) and by a human-evaluated factor provided by developers on their impressions about solutions produced during the optimization process. Abdeen et al. [2], as discussed above, also collects information from developers to guide their optimization process.

Concept analysis provides a way to group objects that have common characteristics [61] and has also been applied to software module clustering. Tonella [67] proposes an approach to module restructuring focused on the use of concept analysis for module identification. Combinations of the concepts extracted by concept analysis are used to create candidate modules. To evaluate the quality of candidate modules, the trade-off between encapsulation violations and decomposition is considered. In order to assess the cost of restructuring a software system, a measure of distance between the proposed and the original module structure is established. The technique was applied to 10 public domain and 10 industrial programs. Glorie et al. [25] reported their experiences in applying formal concept analysis (FCA) on a system comprised of 30 KLOC. They discussed how FCA can be applied to non-trivial software by using a real-world system, presented the leveled approach (a technique to cope with scalability issues using FCA), and a measure of concept quality (to choose the optimal concepts from a set of candidate concepts). The authors concluded that their approach is useful for recognizing certain patterns but not indicated to support clustering decisions.

Finally, Lutz [39] used an information-theoretical complexity measure to evaluate the quality of module distribution and drive a genetic algorithm towards finding a proper clustering. Amoui et al. [3] used the Distance from Main Sequence measure for the same purpose. The clustering problem is also closely related to the distribution of attributes and methods through classes comprising an object-oriented design. This problem was explored using a multi-objective genetic algorithm by Bowman et al. [12] and iterative optimization by Simons et al. [65]. These issues lie outside the scope of this paper. A broader survey on search-based applications for software design is provided in [57], while further details on other search-based approaches to software clustering can be found in [29]. Shtern and Tzerpos [60] also present a survey on Software Engineering related clustering which includes a section for optimization-based software module clustering.

7.2. Architectural mismatch

The research and practical use of code and architectural smells to improve software design is ongoing. Mäntylä [43] presents a survey in which 12 developers were asked to quantify to which extent their modules presented a set of code smells. Answers were given in an integer-interval from 1 to 7, where 1 meant the smell would not be found in the code and 7 indicated that the code certainly contained the smell. Developers were also asked to state, using a similar scale, to which extent they knew each module. The author observed that smells in a given class are strongly correlated, that is, classes presenting signs of *Shotgun Surgery* are also likely to present signs of *Divergent Changes* (Spearman rho = 0.612). Later, Olbrich et al. [52] analysed historical data from two open-source projects focusing on the *God Class* and *Shotgun Surgery* code smells and concluded that the evolution of a system undergoes different stages in which the number of smells could be increasing or decreasing. Thus, an overall conclusion on whether the total number of smells increases steadily or not could not be safely reached.

Fontana et al. [18,19] compare five tools on regard of how effective they were in detecting code smells in six versions of a medium-sized Java project. They evaluated whether different tools would lead to different results and the relevance of their responses to maintenance and software evolution. The authors concluded that metrics and code smells alleviate the effort required to find opportunities for improvement, but human judgment is important to access the relevance of each smell. Zazworka et al. [72] also investigated different tools and approaches to measure the technical debt of a software project (including the presence of code smells) and found that different techniques would point to problems in distinct parts of the code base.

Palomba et al. [53] use information from version control systems to complement the code analysis in finding occurrences of the *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob*, and *Feature Envy* code smells. Their approach resembles our use of commit patterns to identify occurrences of *Shotgun Surgery* and *Scattered Functionality* smells. They have compared their smell detection approach with state-of-the-art smell detectors based only on structural information and attaining more than 60% precision and 60% recall for 8 Java projects.

A recent survey [70] showed that about one third of a sample of 85 developers did not know about code smells and that even those developers concerned with such smells would concentrate their attention in smells related to code size, such as *Large Class* and *Long Method*. This should come as no surprise, since the definition of many code smells is still subjective and requires interpretation [18] and further research is yet required to determine to what extent the presence of code and architectural smells have a significant effect on product quality and effort required for software maintenance. For instance, Yamashita [69] observed that while the presence of code smells can point to modules deserving more attention from a practical maintenance perspective, the amount of code smells in a given module is not a better predictor of required maintenance effort than simpler measures, such as module size. Sjoberg et al. [64] present a study in which six developers performed maintenance tasks on four functional equivalent Java systems. Twelve code smells were investigated for increasing the effort required to perform the selected tasks, but no strong correlation was found between the presence of such smells and the effort spent by developers in the module where they reside.

Code smells have been extensively used as means to identify refactoring needs. Bourquin and Keller [10] present an experience report on a post-mortem analysis over the evolution of a Java program that grew from 50 to 140 KLOC over three years. The authors observed that metrics and code smell detection techniques were useful to find refactoring targets while evolving the system under analysis, but they should be complemented with an analysis of architectural violations to select the most important refactoring opportunities. Macia et al. [40] support their findings by noting that current detection strategies find many code smells that are not related to software architecture problems, while ruling out many anomalies which could be related to those problems.

O'Keeffe and Ó Cinnéide [50,51] proposed a search-based approach to support finding the best sequence of refactoring operations to improve the design of a software project. They compared different heuristic search algorithms and found that multiple-ascent Hill-Climbing outperforms simulated annealing and genetic algorithms for that purpose. Kessentini et al. [32] propose a search approach to correct code smells based on examples of well-designed code. In their approach, JHotDraw is used as reference of good software design and a genetic algorithm is configured to find sequences of refactoring operations maximizing the similarity between target program's parts and code examples from the reference program. They evaluated their approach over four open-source projects and found that the genetic algorithms are able to attain more than 70% precision and 80% recall over a set of manually selected refactoring operations. Finally, Boussaa et al. [11] present a predator–prey co-evolutionary search to produce a set of code smell detection rules. The approach evolves two competing populations, the first to generate a set of detection rules maximizing the coverage of a group of smell examples and the second to maximize the number of artificially-created code smells that cannot be covered by rules produced by the first population. The authors have compared the co-evolutionary approach with a single-population genetic algorithm and an artificial immune system using four Java programs as benchmarks (including Apache Ant). The predator–prey model has shown higher precision and higher recall than both its competing algorithms.

8. Conclusions

In this paper we have addressed whether search-based software module clustering optimization might help recover the design-time architecture of a large, open-source software system – Apache Ant – which has evolved over the course of the last thirteen years. We have briefly presented the evolution of Ant based on size, complexity, change dispersion, coupling, and cohesion metrics. Next, we have applied optimization upon the latest release of the system and perceived that while the values of metrics underlying the optimization process were increased, the design of the optimized versions was more complex and would hardly be acceptable to the development team.

We conclude that we need better models to drive search-based software module clustering optimization. However, despite the limitations of current models shown in our analysis, we see that optimization was useful to highlight these limitations and allow researchers to improve the knowledge base on good design principles. The simple dogma of increasing cohesion and decreasing coupling does not seem to work if taken to extreme scenarios. So, we need to improve our models of what is considered good design. In this sense, Search-based Software Engineering may be useful as a learning tool, evaluating the behavior of Software Engineering models in extreme situations which would not be usually found in software development, allowing the observation and fixing of their limitations.

Some limitations apply to the present work. We have used a selected set of metrics to represent coupling and cohesion, while a large number of alternative metrics for the same purpose can be found in the literature and may lead to distinct results. Among these measures, the normalized versions of MQ and EVM deserve particular attention. We have opted for the non-normalized versions of these metrics because they are more frequently used in SBSE experiments addressing the software module clustering problem, though some work within and outside the field has investigated the problem using their normalized versions (especially MQ). Finally, we have used a flat clustering technique instead of grouping packages into hierarchies. Applying a hierarchical clustering algorithm may lead to different conclusions than those reported in this paper, since dependencies and packages will be distributed in layers and some of them might be hidden from developers interested in different parts of the system as each layer becomes a container-like abstraction by itself. Further research on the cognitive advantages or limitations imposed to developers by using such a layered layout of a program structure are required to compare flat and hierarchical clustering schemas.

Acknowledgement

The authors would like to thank CNPq, the Brazilian research agency that promotes science in Brazil, for their financial support to this project.

References

[1] H. Abdeen, S. Ducasse, H. Sahraoui, I. Alloui, Automatic package coupling and cycle minimization, in: Proceedings of the Working Conference on Reverse Engineering, 2009, pp. 103–112.

[2] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, S. Ducasse, Towards automatically improving package structure while respecting original design decisions, in: Proceedings of the Working Conference on Reverse Engineering, 2013, pp. 212–221.
[3] M. Amoui, S. Mirarab, S. Ansari, C. Lucas, A genetic algorithm approach to design evolution using design pattern transformation, Int. J. Inform. Technol. Intell. Comput. 1 (2) (2006) 235–244.
[4] N. Anquetil, J. Laval, Legacy software restructuring: analyzing a concrete case, in: Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11), Oldenburg, Germany, 2011, pp. 279–286.
[5] N. Anquetil, J. Royer, P. André, G. Ardourel, P. Hnetynka, T. Poch, D. Petrascu, V. Petrascu, JavaCompExt: extracting architectural elements from java source code, in: Proceedings of the 16th Working Conference on Reverse Engineering, Lille, France, 2009, pp. 317–318.
[6] M.A. Araújo, V.F. Monteiro, G.H. Travassos, Towards a model to support studies of software evolution, Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '12), vol. 1, ACM Press, Lund, Sweden, 2012, pp. 281–290. <http://dx.doi.org/10.1145/2372251.2372303>.
[7] M. Barros, An analysis of the effects of composite objectives in multiobjective software module clustering, in: Proceedings of the 21st Genetic and Evolutionary Computation Conference (GECCO 2012), Philadelphia, USA, 2012.
[8] M. Barros, F. Farzat, What can a big program teach us about optimization?, in: Proceedings of the 5th International Symposium on Search-based Software Engineering, SBSE Challenge Track, St. Petersburg, Russia, 2013.
[9] G. Bavota, F. Carnevale, A. De Lucia, M. Di Penta, R. Oliveto, Putting the developer in-the-loop: an interactive GA for software re-modularization, in: Proceedings of the 4th Symposium on Search Based Software Engineering (SSBSE 2012), Riva del Garda, Italy, 2012, pp. 75–89.
[10] F. Bourquin, R. Keller, High-impact refactoring based on architecture violations, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07), Amsterdam, 2007, pp. 149–158.
[11] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, S. Ben Chikha, Competitive co-evolutionary code-smells detection, in: Proceedings of the 5th International Symposium on Search-based Software Engineering, St. Petersburg, Russia, 2013, pp. 50–65.
[12] M. Bowman, L. Briand, Y. Labiche, Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms, IEEE Trans. Software Eng. 36 (6) (2010) 817–837.
[13] L. Briand, S. Morasca, V. Basili, Defining and validating measures for object-based high-level design, IEEE Trans. Software Eng. 25 (5) (1999).
[14] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, S. Schulenburg, Handbook of metaheuristics, in: Hyper-Heuristics: An Emerging Direction in Modern Search Technology, Kluwer Publishers, 2003, pp. 457–474 (Chapter 16).
[15] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Trans. Software Eng. 20 (6) (1994) 476–493.
[16] D. Doval, S. Mancoridis, B. Mitchell, Automatic clustering of software systems using a genetic algorithm, in: Proceeding of the International Conference on Software Tools and Engineering Practice (STEP'99), 1999.
[17] S. Elbaum, J. Munson, Code churn: a measure for estimating the impact of code change, in: Proceedings of the 14th International Conference on Software Maintenance, Maryland, USA, 1998.
[18] F. Fontana, P. Braione, M. Zanoni, Automatic detection of bad smells in code: an experimental assessment, J. Object Technol. 11 (2) (2012) 1–38.
[19] F. Fontana, P. Braione, A. Marino, M. Mäntylä, Code smell detection: towards a machine learning-based approach, in: IEEE International Conference on Software Maintenance, 2013, pp. 396–399.
[20] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.
[21] M. Fowler, Continuous Integration <http://www.martinfowler.com/articles/continuousIntegration.html> (accessed 13.11.13).
[22] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: Proceedings of the 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, 2009, pp. 255–258.
[23] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch or, why it's hard to build systems out of existing parts, IEEE Software 12 (6) (1995) 17–26.
[24] S. Gibbs, D. Tsichritzis, E. Casais, O. Nierstrasz, X. Pintado, Class management for software communities, Commun. ACM 33 (9) (1990) 90–103 (New York, USA).
[25] M. Glorie, M. Zaidman, L. Hofland, A. van Deursen, Splitting a large software archive for easing future software evolution: an industrial experience report using formal concept analysis, in: Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR), 2008.
[26] M. Glorie, A. Zaidman, A. van Deursen, L. Hofland, Splitting a Large Software Repository for Easing Future Software Evolution – An Industrial Experience Report, Technical Report TUD-SERG-2009-002, Delft University of Technology, 2009.
[27] M. Hall, N. Walkinshaw, P. McMinn, Supervised software modularisation, in: Proceedings of the International Conference on Software Maintenance, Riva del Garda, Italy, 2012, pp. 472–481.
[28] M. Harman, B. Jones, Search-based software engineering, Inf. Software Technol. 43 (14) (2001) 833–839.
[29] M. Harman, S. Masouri, V. Zhang, Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications, Department of Computer Science, King's College London, Technical Report TR-09-03, April 2009.
[30] M. Harman, S. Swift, K. Mahdavi, An empirical study of the robustness of two module clustering fitness functions, in: Proceedings of the 14th Genetic and Evolutionary Computation Conference (GECCO'05), Washington, USA, 2005.
[31] B. Henderson-Sellers, Software Metrics, Prentice Hall, Hemel Hempstead, UK, 1996.
[32] M. Kessentini, R. Mahaouachi, K. Ghedira, What you like in design use to correct bad-smells?, J Software Qual. 11 (4) (2013) 551–571.
[33] R. Koschke, An incremental semi-automatic method for component recovery, in: Proceedings of the Working Conference on Reverse Engineering, 1999, pp. 256–267.
[34] T. Kuhn, The Structure of Scientific Revolutions, third ed., The University of Chicago Press, 1996. ISBN-10: 0226458083.
[35] A. Kumari, K. Srinivas, Software module clustering using a fast multi-objective hyper-heuristic evolutionary algorithm, Int. J. Appl. Inform. Syst. 5 (6) (2013) 12–18.
[36] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, Springer-Verlag, Berlin, Heidelberg, 2006.
[37] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and the Unified Process, Prentice Hall, Upper Saddle River, NJ, 2002.
[38] M. Lehman, Programs, life cycles, and laws of software evolution, Proc. IEEE 68 (9) (1980) 1060–1076.
[39] R. Lutz, Evolving good hierarchical decompositions of complex systems, J. Syst. Architect. 47 (2001) 613–634.
[40] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, A. von Staa, Are automatically-detected code anomalies relevant to architectural modularity? An exploratory analysis of evolving systems, in: Proceedings of the 11th International Conference on Aspect-oriented Software Development (AOSD'12), Potsdam, Germany, 2012, pp. 167–178.
[41] K. Mahdavi, M. Harman, R. Hierons, A multiple hill climbing approach to software module clustering, in: 19th IEEE International Conference on Software Maintenance (ICSM'03), Amsterdam, 2003, pp. 22–26.
[42] S. Mancoridis, B. Mitchell, Y. Chen, E. Gansner, Bunch: a clustering tool for the recovery and maintenance of software system structures", in: Proceedings of the IEEE International Conference on Software Maintenance, 1999, pp. 50–59.
[43] M. Mäntylä, Bad Smells in Software: A Taxonomy and an Empirical Study, Master Thesis, Helsinky University of Technology, 2003.
[44] M. Mäntylä, C. Lassenius, Subjective evaluation of software evolvability using code smells: an empirical study, J. Emp. Software Eng. 11 (3) (2006) 395–431.
[45] S. McConnell, Code Complete, second ed., Microsoft Press, 2004.
[46] T. Mens, T. Tourwe, A survey of software refactoring, IEEE Trans. Software Eng. 30 (2) (2004) 126–139.
[47] B.S. Mitchell, A Heuristic Search Approach to Solving the Software Clustering Problem, Ph.D. Thesis, Drexel University, Philadelphia, PA, USA, 2002.
[48] H.A. Muller, M.A. Orgun, S.R. Tilley, J.S. Uhl, A reverse engineering approach to subsystem structure identification, Software Maint. Res. Pract. 5 (4) (1993) 181–204.
[49] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, I. Moghadam, Experimental assessment of software metrics using automated refactoring, in: Proceedings of the 6th International Symposium on Empirical Software Engineering and Measurement, Sweden, 2012, pp. 49–58.
[50] M. O'Keeffe, M. Ó Cinnéide, Getting the most from search-based refactoring", in: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07), London, 2007, pp. 1114–1120.
[51] M. O'Keeffe, M. Ó Cinnéide, Search-based refactoring: an empirical study, J. Software Maint. Evol.: Res. Pract. 20 (5) (2008) 345–364.
[52] S. Olbrich, D. Cruzes, V. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, in: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, 2009.
[53] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. de Lucia, D. Poshyvanyk, Detecting bad smells in source code using change history information, in: Proceedings of the 28th International Conference on Automated Software Engineering (ASE), CA, USA, 2013, pp. 268–278.
[54] D.L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, Technical Report, Computer Science Department, Camegie-Mellon University, August 1971.
[55] M. Perepletchikov, C. Ryan, A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software, IEEE Trans. Software Eng. 37 (4) (2011) 449–465.
[56] K. Praditwong, M. Harman, X. Yao, Software module clustering as a multiobjective search problem, IEEE Trans. Software Eng. 37 (2) (2011) 262–284.
[57] O. Räihä, A Survey on Search-Based Software Design, Technical Report D-2009-1, Department of Computer Sciences University of Tampere, March 2007.
[58] N. Sangal, F. Waldman, Dependency models to manage software architecture, CROSSTALK: J. Def. Software Eng. (2005) 8–12.
[59] Robert W. Schwanke, An Intelligent Tool for Re-engineering Software Modularity, vol. 10, Siemens Corporate Research, 1991.
[60] M. Shtern, V. Tzerpos, Clustering methodologies for software engineering, Advan. Software Eng. 2012 (2012) 1–18. Article 792024.
[61] M. Siff, T. Reps, Identifying modules via concept analysis, IEEE Trans. Software Eng. 25 (6) (1999).
[62] H. Simon, The Sciences of the Artificial, third ed., MIT Press, 1996.
[63] C. Simons, I. Parmee, Elegant object-oriented software design via interactive, evolutionary computation, in: IEEE Trans. Syst., Man, Cybernet., Part C (Appl. Rev.) 42 (6) (2012) 1797–1805.

[64] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, T. Dyba, Quantifying the effect of code smells on maintenance effort, IEEE Trans. Software Eng. 39 (8) (2013) 1144–1156.
[65] C. Simons, I. Parmee, R. Gwynllyw, Interactive, evolutionary search in upstream object-oriented class design, IEEE Trans. Software Eng. 36 (6) (2010) 798–816.
[66] R. Taylor, N. Medvidovic, E. Dashofy, Software Architecture: Foundations, Theory, and Practice, John Wiley & Sons, 2008.
[67] P. Tonella, Concept analysis for module restructuring, IEEE Trans. Software Eng. 27 (4) (2001).
[68] A. Tucker, S. Swift, X. Liu, Grouping multivariate time series via correlation, IEEE Trans. Syst., Man, Cybernet., B: Cybernet. 31 (2) (2001) 235–245.
[69] A. Yamashita, How good are code smells for evaluating software maintainability? Results from a comparative case study, in: Post-doctoral Symposium at 29th International Conference on Software Maintenance (ICSM), 2013.
[70] A. Yamashita, L. Moonen, "Do developers care about code smells? An exploratory survey, in: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), Koblenz, Germany, 2013, pp. 242–251.
[71] E. Yourdon, L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Yourdon Press, 1979.
[72] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, Comparing four approaches for technical debt identification, J. Software Qual. (2013) 1–24.