Software Architecture Reconstruction via a Genetic Algorithm: Applying the Move Class Refactoring

Theodoros Maikantis, Angeliki-Agathi Tsintzira, Apostolos Ampatzoglou, Elvira-Maria Ar-Vanitou, Alexander Chatzigeorgiou
Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

Ioannis Stamelos
Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece

Stamatia Bibi
Department of Electrical and Computer Engineering, University of Western Macedonia, Kozani, Greece

Ignatios Deligiannis
Department of Information and Electronic Engineering, International Hellenic University, Thessaloniki, Greece

ABSTRACT

Modularity is one of the four key principles of software design and architecture. According to this principle, software should be organized into modules that are tightly linked internally (high cohesion), whereas at the same time as independent from other modules as possible (low coupling). However, in practice, this principle is violated due to poor architecting design decisions, lack of time, or coding shortcuts, leading to a phenomenon termed as architectural technical debt (ATD). To alleviate this problem (lack of architectural modularity), the most common solution is the application of a software refactoring, namely *Move Class*—i.e., moving classes (the core artifact in object-oriented systems) from one module to another. To identify Move Class refactoring opportunities, we employ a search-based optimization process, relying on optimization metrics, through which optimal moves are derived. Given the extensive search space required for applying a brute-force search strategy, in this paper, we propose the use of a genetic algorithm that re-arranges existing software classes into existing or new modules (software packages in Java, or folders in C++). To validate the usefulness of the proposed refactorings, we performed an industrial case study on three projects (from the Aviation, Healthcare, and Manufacturing application domains). The results of the study indicate that the proposed architecture reconstruction is able to improve modularity, improving both coupling and cohesion. The obtained results can be useful to practitioners through an open source tool; whereas at the same point, they open interesting future work directions.

PCI 2020, November 20–22, 2020, Athens, Greece

ACM ISBN 978-1-4503-8897-9/20/11…$15.00
https://doi.org/10.1145/3437120.3437292

CCS CONCEPTS

• **Software and its engineering**; • **Software creation and management**; • **Software development techniques**; • **Object-oriented development, Software verification and validation**; • **Empirical software validation**;

KEYWORDS

Architecture reconstruction, coupling, cohesion, move class

ACM Reference Format:
Theodoros Maikantis, Angeliki-Agathi Tsintzira, Apostolos Ampatzoglou, Elvira-Maria Ar-Vanitou, Alexander Chatzigeorgiou, Ioannis Stamelos, Stamatia Bibi, and Ignatios Deligiannis. 2020. Software Architecture Reconstruction via a Genetic Algorithm: Applying the Move Class Refactoring. In *24th Pan-Hellenic Conference on Informatics (PCI 2020), November 20–22, 2020, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3437120.3437292

1 INTRODUCTION

Quality assessment in early stages of software development (such as architecture) is prominent, in the sense that changing early design artifacts usually leads to substantial rework—increasing maintenance costs[13]. The magnitude of maintenance costs can accumulate up to 50%-70% of the complete lifecycle cost for software development[15] Therefore, it is of paramount importance to reduce such costs; usually caused by poor development quality. One of the main problems regarding architecture quality is architecture decay[8]: i.e., a phenomenon through which the quality of an architecture diminishes along evolution and starts to drift away from the original architectural decisions. According to van Vliet [15] architectural quality can be perceived through assessing three central object-oriented concepts: *abstractness*, *modularity*, and *complexity*. Among those, in this paper we focus on modularity, i.e., the level of internal coherence and independence of a software component. In the literature, the aforementioned two factors of modularity are termed as coupling and cohesion[1]. The desired evolution (in terms of modularity) would be that coupling decreases and cohesion increases over time; however, in practice this rarely occurs, since modules become larger, providing more diverse functionality, and become in need of more external modules [1].

To ensure the viability of software (i.e., decrease maintenance cost) by preventing architecture decay, we need to reconstruct the

originally planned architecture, so as to optimize it current state (in this paper in terms of modularity): improved modularity would make the design more reusable and extensible [12]. In this study, we propose a method for refactoring existing software architectures aiming to improve modularity, by identifying conceptually similar artifacts, placing them in the same component, and creating a hierarchical component-based architecture. The outcome of the method is a modified architectural decomposition, which could be instantiated by applying the "*Move Class*" refactoring [5][1]. The proposed method, namely Design Reconstruction based on a Genetic Evolutionary Algorithm (DeRecGEA), which relies on Genetic Evolutionary Algorithms (GEA). This choice was influenced by several factors, since through GEA we are able of: (a) solving problems of varying complexity; and (b) identifying a possible solution faster than a Brute Force algorithm, given the complexity of a problem and the amount of necessary calculations. Until now, modularity has been intensively studied at both class and method level by applying the "*Move Method*", the "*Extract Class*", and the "*Extract Method*" refactorings. However limited research has been performed for supporting the application of the "*Move Class*" refactoring. We note that as input for DeRecGEA we provide the source code of the examined software. This decision was made since: (a) the intended architecture (as captured by design / architecture documents) rarely available in practice [9]—thus, the applicability of the method would be limited; and (b) all changes that are applied along evolution are applied in the implemented architecture—i.e., the source code. Therefore, the proposal of "*Move Class*" refactoring would be more straightforward; and would not imply the existence of traceability links between architecture elements and source code.

To evaluate the proposed approach, we have performed a case study on three industrial projects: one from the aviation domain (written in C++), one from the healthcare domain (written in C++), and the final one from the smart manufacturing domain (written in Java). In particular, we obtained the original code of the projects, and refactored them, using the proposed GEA algorithm. To explore if the application of the GEA improved modularity, we followed a pre-post analysis, i.e., we compared the values of modularity before the application of the treatment (i.e., the application of the architectural refactoring) to the values of modularity. The rest of the paper is organized as follows: in Section 2 we present background information necessary to understand the proposed process, and related work (i.e., automated approaches for architectural refactorings); and in Section 3 we present the DeRecGEA approach. In Section 4, we present the case study design and the obtained results. Finally, in Section 5 we conclude the paper by providing some implications for researchers and practitioners and threats to validity.

2 BACKGROUND INFORMATION AND RELATED WORK

Several attempts – methodologies exist for the identification of Architectural Refactorings, or otherwise known as solutions to Architectural Smells. The methodologies identified in the literature utilize either manual or semi-manual processes; therefore, there is a lack of purely automated processes. Garcia et al. [6] propose the use of schematic diagrams, utilized by architects to detect architectural smells in both conceptual and recovered / implemented architecture. After detection a manual analysis is performed by the architect, to assess the impact of the smell on relevant qualities. Marinescu [10] suggests the use of Detection Strategies (quantifiable expressions of a rule), to detect conforming source code fragments. These strategies, analyze source code models by using metrics. Tourwe and Mens [14] use a semi-automated approach, based on meta programming logic, to detect if refactorings are needed, identify which refactorings should be applied and finally automatically apply them. Finally, Arnold [2] remarks that system modularization currently requires much human judgment, and lists a few principles for manually performing the identification of smell and applying the refactoring. It has been argued that search-based software engineering can provide acceptable solutions to many problems with competing constraints employing metaheuristic approaches such as Genetic Algorithms, Simulated Annealing and Tabu Search[7]. The optimum assignment of responsibilities to classes has been for example tackled with the help of multi-objective genetic algorithms[4].

Genetic Evolutionary Algorithms are inspired by biology; thus, they adopt characteristics and processes found in nature. These stem mainly from Darwin's theory of evolution, and utilize the process of natural selection[3]. GEA is basically a random process of evolving solutions (each new solution stems from a combination of existing ones) that will eventually end-up with an optimal solution. In order for the GEA iterations to end we need to define a termination criterion. We note that the use of a Genetic Evolutionary Algorithm, contrary to the use of a Brute Force algorithm, will most likely not provide us with the best possible result (global optimum); however, the result will be adequate (local optimal), but it will be timely retrieved. The basic steps of any GEA are the following: (a) *Population Initialization*; (b) *Selection*; (c) *Crossover*; and (d) *Mutation*—see Figure 1; whereas the basic entities are: (a) the population, (b) the individual, and (c) the gene. The individual is a representation of a possible solution, the gene is the actual useful information about the solution (contained in each Individual), and the population is the collection of our calculated possible solutions (Individuals). Imitating nature, steps 2-4 are repeated over several generations, resulting in a final solution [3]. The basic steps are outlined below; whereas their application in DeRecGEA are detailed in Section 3:

- ***Selection*** refers to the evaluation of an Individual. The evaluation is based on the *fitness value* representing each Individual of the Population. The Fitness value is the score of the proposed solution (gene).
- ***Crossover*** is the function of combining two or more existing Individuals, usually the ones with already good Fitness, and producing a new one. The main purpose of this is the improvement of the solution pool, by making the population more diverse, and introducing new high Fitness Individuals to the population.
- ***Mutation*** process although different, serves a similar purpose as Crossover. The process uses a single Individual and mutates – changes its gene, in order to diversify it and produce new combinations.

[1]The move class refactoring suggests to move one class from one package to another, to which it is more conceptually relevant

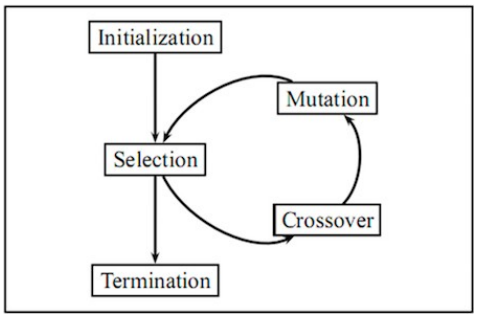

Figure 1: Genetic Evolutionary Algorithm process

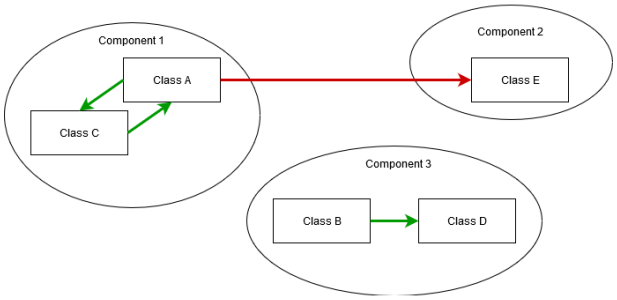

Figure 3: Correct (green) and False (red) edges between Classes

3 ARCHITECTURAL DESIGN RECONSTRUCTION BASED ON A GENETIC EVOLUTIONARY ALGORITHM (DERECGEA)

In this section, we present the proposed algorithm for applying the Move Class refactoring. In particular, we organize the section, based on the generic steps of the GEA, presented in Section 2. The goal of DeRecGEA is to group classes, into modular components, in a recursive manner (as dictated by GEA), resulting into sub-components. These sub-components, act as the input of GEA input, to achieve the intended hierarchical structure. The component structure is a non-restrictive one, in the sense that they only aim at characterizing groups of classes. As a result if two components of two different individuals have the same name, they are not considered the same, because they could be labeling two different class groups.

Entities Representation: The main entity of DeRecGEA, the ***Individual*** contains a representation of the classes that we want to group into components, the components themselves and the relation between the two—see Figure 2. In particular, the gene of the individual is represented as a vector, with size equal to the number of classes. The index of the vertex corresponds to an id of the class, whereas the value of the vertex to the component that it is mapped to. For the selection process the required information is the ***Fitness value***. The fitness value is a compound metric relying to coupling and cohesion (since we aim to increase modularity) [1]. To assess class Cohesion and Coupling, we are utilizing the classes' outgoing "Desired" and "Non-Desired" edges. By edges we mean the outgoing dependencies of a class. A Correct edge (3 *green dependency*) would be a dependency where the classes belong to the same component; similarly, a False edge (3 *red dependency*) would refer to one where the classes belong to different components. Ultimately the Cohesion of a class would be the total of its "Desired" edges and its Coupling would be that of its "Non-Desired" edges. The calculation of the fitness value relies on the following equation.

$$\begin{aligned} Modularity &= Fitness\ Value \\ &= \frac{|Desired\ Edges| - |Non\ Desired\ Edges|}{|Total\ Edges|} \end{aligned}$$

The fitness value is bounded in the [-1, 1] range; and follows the generic practice that modularity is positively affected by cohesion, and negatively affected by coupling [15]. Given the fact that both coupling and cohesion are calculated as edges (similarly to the total number of edges), we consider the aggregation through subtraction and division as appropriate. The component fitness is defined by the mean value of its contained classes. Similarly, the fitness of an Individual is the mean value of its components. The termination condition is the lack of improvement in the fitness score for 30 evolution generations.

Population Initialization is randomly performed in DeRecGEA, in the sense that all the Individuals comprising the population are created with a random structure regarding the number of components and the classes belonging to them. To accelerate the solution process, some restrictions (upper and lower component number limitations) are in place guided by the rules of good software engineering—e.g., a single component should contain at most 30 entities (methods to classes, classes to packages, etc.)[11]. Considering the aforementioned rule, we set as upper limit for our number of components the number of classes divided by 20 (instead of 30) to provide some leeway, considering that the dictated number of objects to components is not absolute and applicable to all the real-world implementations. As the lower limit for the number of

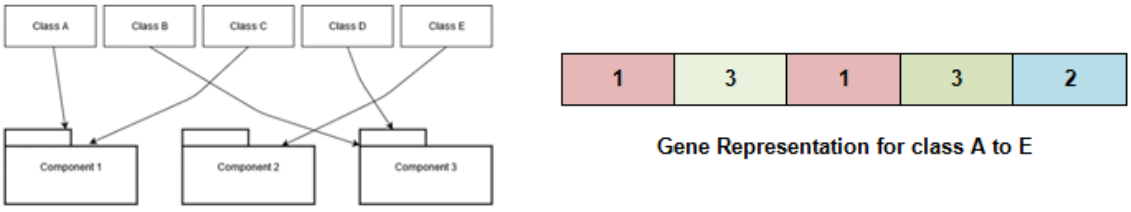

Figure 2: Component Class distribution

components we simply chose the number 2, to enable at least one split.

The *Selection* starts by calculating the ***Fitness value*** of each Individual. It happens on two occasions, before the Crossover and before the Mutation. For the Crossover, the 10% of the best performing Individuals are selected to undergo the process. Similarly, for the Mutation the 20% of the best performing Individuals are selected. All the new Individuals that are produced, as a result of the two aforementioned processes, are put back into the population pool. The population pool has a fixed number of Individuals; therefore any excess ones are automatically deleted. This selection is aided again by the fitness value, where the Individuals that are performing the worst are discarded.

During *Crossover* process, two (parent) Individuals are combined, resulting in the creation a new (child) Individual. The new Individual is reinserted into the population pool, thus representing part of the population's new generation. The child inherits characteristics from its parents, mainly the number of components and the way that the classes are distributed to them. The inherited characteristics are not exact copies of the ones the parents had, but a combination of the two. The two important elements of our Algorithm's Crossover process are: (a) the selection of the ***component number*** of the new Individual; (b) the ***distribution*** of ***classes*** to the available Components. For the number of components, the choice is restricted between the parents' number of Components, but the exact number is randomly chosen. To increase the GEA's performance, the new random component number value has an increased probability to be closer to value of the parent with the best fitness. This is achieved by using a shifted Gaussian Distribution in favor of the better performing parent's component number, to decide the resulting component number. For the class distribution we check in which parent component each class had a better fitness, and then assigned it to that.

The implemented *Mutation* function is a simple component change for a class. The mutation process is repeated for a number of times, equal to that of a fifth of the total Individual classes. The class that will undergo a component change is selected by the Fitness value. Classes with the lowest value are selected for mutation; the selection of its new component however is random. There are complimentary functions to the mutation process, which are used on special occasions and help with the avoidance of local optimums. These functions either split a component into two separate ones (in case a component has low cohesion), or join two components together to create a larger one (in case the two components have high coupling). On a final note, as mentioned before the GEA is executed recursively for the entirety of the project, including its newly made components. Except for the GEA's termination condition, there also needs to be a recursion termination condition. By following the aforementioned rules of good software engineering, we set the recursion to terminate when all components contain 30 or less classes.

4 EMPIRICAL VALIDATION

Study Design: To validate DeRecGEA, we have performed an exploratory case study on three industrial codebases, retrieved as part of the SKD4ED consortium[2]. SDK4ED is a research project that focuses (among others) on the management of technical debt, in embedded systems. Therefore, the retrieved codebases are embedded applications that can be classified as ***Aviation*** (App1), ***Healthcare*** (App2), and ***Smart Manufacturing*** (App3). Some demographics of the applications are presented in Table 1. In the table apart from the descriptive statistics, we also report the initial scores of coupling, cohesion, and modularity—i.e., the metric scores before the applying DeRecGEA.

Given the initial versions of the three codebases, we have applied DeRecGEA, through the SDK4ED toolbox[3]. The source code of the application is also available online[4]. Subsequently, we have recalculated the three metrics: coupling, cohesion, and modularity and contrasted the results. By considering the small sample size, we were not able to perform any statistical analysis further than descriptive statistics. The results aim to answer the following high-level question: "***Is the application of DeRecGEA able to improve the modularity of the implemented architecture?***".

Results: Upon the application of the DeRecGEA algorithm, we have observed that in 2 out of 3 cases the modularity of the systems has improved, whereas in the other it was slightly decreased. The results are visualized in Figure 4 in the figure, as positive or negative, we do not refer to the increase or decrease in absolute numbers, but to positive or negative effects (as positive we consider increase in the score of modularity and cohesion, and decrease in the scores of coupling and size). By focusing on specific quality properties, regarding cohesion we can observe effects ranging from -17% to 67%; whereas for coupling from -50% to 28%. The fact that DeRecGEA performs better in optimizing cohesion rather than coupling (in all three cases) suggests that there might be some room for calibration of the algorithm (e.g., the fitness function or the mutation) to favor moves that decrease coupling. Regarding modularity the impact of DeRecGEA is ranging from -13% to 167%, whereas in terms of size the impact was ranging from no impact to 22% improvement. Thus, we can claim that the proposed algorithm

[2]https://sdk4ed.eu/
[3]http://160.40.52.130:3000/
[4]https://github.com/teomaik/DeRecGEA

Table 1: Sample Demographics

Project	*Programming Language*	*Initial Cohesion*	*Initial Coupling*	*Initial Modularity*	*Size in Packages*	*Size in Lines of Code*
App1	C++	0.30	2.20	-0.76	152	398578
App2	C++	2.40	3.50	0.00	2	5964
App3	Java	1.20	7.70	-0.73	7	2700

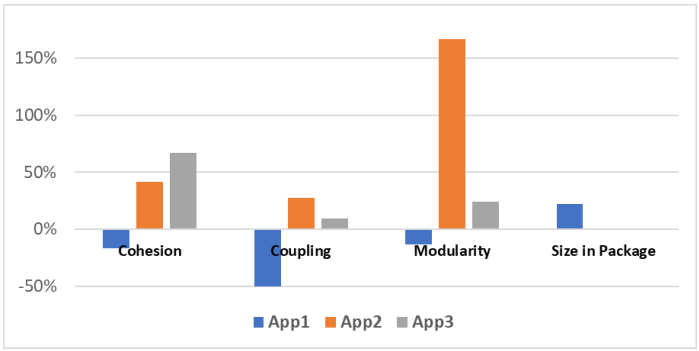

Figure 4: Application of DeRecGEA on Software Modularity

tends to not increase the number of components in systems, but either just re-arranges classes or merges packages. Nevertheless, the negative effect of DeRecGEA optimization on one pilot case, suggests that the aforementioned refinement, might be necessary.

5 CONCLUSIONS

In this paper, we present the results of an initial attempt to improve software architecture modularity, through the application of a metrics-driven Evolutionary Genetic Algorithm. The proposed algorithm has been tested in three industrial projects: two small-scaled and one large scale. The results on the two small-scale projects are satisfactory, in the sense that their modularity is significantly improved. However, regarding the large-scale project the results are encouraging, but still not satisfactory. In particular, we have been able to reduce the size of the project (in terms of number of components), by only getting a limited modularity penalty. Nevertheless, the degradation of coupling in the retrieved solution is alerting, and points into an interesting research direction, on how the algorithm of the fitness function can be calibrated to lead to more coupling-wise viable solutions. Nevertheless, we encourage practitioners (esp. of small- and medium-scale applications) to experiment with the developed tool and Move Class refactorings, since they seem as promising solutions for architecture decay in terms of modularity.

ACKNOWLEDGMENTS

Work reported in this paper has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement No 780572 (project: SDK4ED).

REFERENCES

[1] Apostolos Ampatzoglou, Angeliki Tsintzira, Elvira Maria Arvanitou, Alexander Chatzigeorgiou, Ioannis Stamelos, Alexandru Moga, Robert Heb, Oliviu Matei, Nikolaos Tsiridis, and Dionysios Kehagias. Applying the single responsibility principle in industry: Modularity benefits and trade-offs. 04 2019.
[2] R. S. Arnold. Software restructuring. *Proceedings of the IEEE*, 77(4):607–617, 1989.
[3] Thomas Bäck. Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms. Oxford University Press, Inc., USA, 1996.
[4] Michael Bowman, Lionel Briand, and Yvan Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Trans. Software Eng.*, 36:817–837, 11 2010.
[5] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.
[6] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In*2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, 2009.
[7] Mark Harman. Search based software engineering. In Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, pages 740–747, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
[8] Duc Le, Carlos Carrillo SÎ´nchez, Rafael Capilla, and Nenad Medvidovic. Relating architectural decay and sustainability of software systems. 04 2016.
[9] Christian Manteuffel, Dan Tofan, Paris Avgeriou, Heiko Koziolek, and Thomas Goldschmidt. Decision architect - a decision documentation tool for industry. *Journal of Systems and Software*, 112, 10 2015.
[10] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In*20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359, 2004.
[11] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction.* Best Practices for Developers. Microsoft Press, Redmond, WA, 2 edition, 2004.
[12] Markus Pizka. Straightening spaghetti-code with refactoring? pages 846–852, 01 2004.
[13] Ian Sommerville. *Software Engineering.* Pearson, 10th edition, 2015.
[14] Tom Tourwe and Tom Mens. Identifying refactoring opportunities using logic meta programming. pages 91– 100, 04 2003.
[15] Hans van Vliet. *Software Engineering: Principles and Practice.* Wiley Publishing, 3rd edition, 2008.