Comput Syst Sci & Eng (2019) 5: 283–296
© 2019 CRL Publishing Ltd

International Journal of
Computer Systems
Science & Engineering

Investigating The Effect of Software Packaging on Modular Structure Stability

Shouki A. Ebad[1]*, **Moataz Ahmed**[2]†

[1] *Computer Science Department, Faculty of Science, Northern Border University, Saudi Arabia*
[2] *Department of Information and Computer Science, King Fahd University of Petroleum and Minerals, Saudi Arabia*

In object-oriented development, packages form the basic modular structural components of large-scale software systems. Packaging processes aim to group classes together to provide well-identified functions/services to the rest of the system. In this context, it is widely believed that packaging quality has an influence on the software stability so that it should be useful predictors for modular structural stability. In this paper, we investigate the effect of packaging configurations on the modular structure stability of object-oriented systems. Using genetic algorithms, we conducted a series of experiments to find the relation between the packaging quality and modular structure stability. We conducted experiments on open source systems using an automatic packaging approach recently proposed by the authors. Results show that the stability of releases automatically packaged using that approach was better or at least comparable to those of the corresponding original releases manually packaged by the software developers. Moreover, the different parameters settings of the genetic algorithms used in our experiments play an important role to improve the overall quality. The experimental results suggest that the considered packaging approach is useful for practitioners to develop architecturally stable software systems.

Keywords: Software Engineering, Search-Based Software Engineering (SBSE); Object-Oriented Design & Analysis (OODA); Software Modularization; Software Architecture; Software Stability

1. INTRODUCTION

Software packaging (interchangeably referred to as modularization [1][2], clustering [3][4][5][6], and decomposition [7] is the process of grouping object-oriented (OO) classes into packages so that each package, desirably, offers a single service that is entirely offered by that package [8]. Because this process targets the software structure and architecture without affecting its internal behavior, packaging is mainly meant to achieve quality objectives such as high maintainability and high reusability [1][3]. Packaging is often employed during the architectural design stage of the software development life cycle [8]. The organization and modularization of code into components such as classes and files represent architectural views of a software system [9]. In such a context, packages form the basic architectural components of large-scale OO software systems [8][10].

Software continues to change during its lifetime due to evolution in the requirements and/or changes in the environment. System structure stability measures the extent to which software is flexible to endure requirement and environment changes while preserving the architecture [11]. Well-designed OO software systems should be able to evolve without major changes in their architecture [12]. This is highly desirable because implementing architectural changes is very expensive such changes are likely to have high-amplified changes through the design [13][14][15]. Consequently, the corresponding maintenance cost and effort is higher. Undesirable instability can exist at the different levels of software design ranging from architecture level to detailed class level [10][16]. A number of measures for stability at architecture level have been defined [10][14][17][18][19][20][21][22].

*Email: shouki.abbad@nbu.edu.sa
†moataz@kfupm.edu.sa

In order to optimize the software modular structure for maintainability, for instance, architects should target building highly stable software. To this end, we herein study the effect of packaging on software modular structure stability. Software packaging is often treated as a search-based software engineering (SBSE) optimization problem where a relevant software metric is used as the fitness function to guide the search using appropriate algorithms, popularly heuristic. In this paper, we use Genetic Algorithms (GA) to automatically re-package two real-world open-source Java frameworks: JHotDraw and AWT. We show that re-packaging the frameworks can improve stability over different releases, compared to the original manual packaging. A recent survey has shown that most relevant research in the literature have been focusing on using internal quality attributes (e.g., cohesion and coupling) to guide such a packaging process [23]. To the best of our knowledge, few studies [24] investigated the effect of packaging on the modular structure stability as an external quality attribute.

The rest of the paper is organized as follows. In Section 2 we give a background on the functionality-based packaging approach and the architectural stability metric which are used in the assessment conducted in this research. We also give a brief background on GA in the same section. Section 3.2 describes our process to extract the artifacts required for the assessment. In Section 3 we investigate the relation between packaging and modular structure stability using two real-world case studies of open-source Java frameworks: JHotDraw and AWT. Section 5 discusses some threats to the study's validity. Section 6 reviews the literature of packaging approaches and measures of stability at architecture level. We finally present the conclusions and future work in Section 7.

2. BACKGROUND

In this section we provide the necessary background about the packaging approach [8] and the architecture stability metric [10] which we use in assessing the impact of the functionality-based packaging on the software architecture stability. We conclude the section with a brief background on GA.

2.1 Functionality-Based Packaging Approach

According to Jacobson [25], the functionality (i.e., the services) that users require of the OO system is documented in use cases (UCs). Each UC is realized by at least one sequence diagram (SD) that depicts how the objects interact and work together to provide services. Unlike other packaging approaches which are applicable at source code level, Ebad and Ahmed [8] have recently presented a functionality-based approach that can be applied during the architecture design phase to group classes into packages using SDs. The rationale behind this was that packaging should decompose the system into packages where each package performs a single task that is, as much as possible, entirely carried out in that package. To this end, they designed a packaging metric to consider two aspects: (1) each UC should be covered by the least number of packages, and (2) classes in each package should be related in the sense that the intersection of the sets of UCs they contribute to is as large as possible. The first aspect (degree of UC coverage) reflects loose coupling; while the second (degree of class relevancy) reflects high cohesion. Consequently, to reflect both aspects, Ebad and Ahmed consider the weighted sum of two measures: UC coverage and class relevancy. Conceptually, and in a nutshell, UC coverage by package Pi considers, for each UC, the percentage of UC interactions and percentage of UC methods available in Pi compared to all interactions and methods needed to offer that UC. For example, let's assume that offering a use case UCj requires five interactions and four methods; if only 25% of interactions and 50% of methods are available in Pi, then the coverage of UCj by Pi, is the weighted sum of the two components; it is reported as 0.38 (with weight 0.5 for each component). Similarly, the degree of class relevancy of Pi considers, for each pair of classes in Pi, the percentage of the UCs that involves the pair together compared to the number of UCs that involve any or both classes of the pair (this measure is named functionality). It also considers, for each class in Pi, the percentage of methods interacting with other methods in Pi (this measure is named utilization). For example, consider two classes, C_1 and C_2, exist in Pi; with two and three methods for each, respectively. Let's assume that C_1 and C_2 are required together in 7 UCs. If 10 UCs require either C_1 or C_2, or the two together; and only one method in C_1 interacts with two methods in C_2, then the functionality is 0.7 and utilization is 0.5 (for C_1) and 0.67 (for C_2); the class utilization is obtained by taking the average of utilization for each class i.e., 0.59. The degree of class relevancy of Pi is then the weighted sum of the two components; it is reported as 0.64 (with weight 0.5 for each component). Actually, each of these measures considers a number of components in its calculation. For example, the distinction between direct interactions vs. indirect interactions between classes is considered in the calculations of class relevancy. The corresponding components are designed in a way that credits loose coupling and high cohesion, respectively.

Actually, the objective of using "weights" in our context is not for traditional multi-objective optimization. The weights in our case are semantically relevant as you have seen in the above example, the considered approach uses weighted sum to establish the "semantically strength" of some of the metric components than the others. For example, featuring "strong" intra-package dependencies (i.e., highly cohesive) leads to higher weights of some components in the metric (i.e., class relevancy) than other components. Similarly, a direct interaction reflects a "stronger" relation than an indirect one. Therefore, the weight of direct interaction should have a higher value than that of indirect one. Setting weights to 0.5 means the considered components have equal participation or strength. Using weighted sum is already used in some of the exiting works such as [26]. The aggregate of these two measures is used to calculate the quality of a package P_i, *PackagingQlty*(P_i)

$$\begin{aligned}\text{PackagingQlty(Pi)} = {} & \text{w}_\text{U} \times \text{degree of UC Coverage by Pi} \\ & + \text{w}_\text{C} \times \text{degree of Class Relevancy of Pi}\end{aligned} \tag{1}$$

where w_U is the weight of the UC's coverage in a package and w_C is the weight of the class relevancy in a package, so that $w_U, w_C \in [0, 1]$ and $w_U + w_C = 1$. These weights have been optimized through trial and error to allow maximum positive correlation with stability [27]. The overall packaging metric is defined as the average PackagingQlty of all packages in the system. Formally, it is calculated as:

$$\text{OverallPackaging(system)} = \text{Avg}\ (\text{PackagingQlty}(P_j)) \quad \forall \text{ package } P_j \text{ in the system} \tag{2}$$

The packaging effort should try to maximize the proposed metric. The *OverallPackaging* metric would be used as the fitness function to guide the search using appropriate algorithms. The metric being used in the research has different weight structures at different levels. The weights themselves are being optimized; however, this is beyond the scope of this paper. We cite the relevant references for interested readers.

2.2 Modular Structure Stability Metric

The stability metric also proposed by Ebad and Ahmed [10] focused on inter-package message passing connections. In this context, inter-package connection (IPC) is a connection where the caller and callee belong to different packages. The metric considers the number of changed and unchanged IPCs between release $i + 1$ and release i. For example, consider two releases r_1 and r_2 with two packages P_1 and P_2, Let A be the set of added IPCs in r_2, D be the set of deleted IPCs in r1, and O be the set of IPCs that exist in $r1$. Let |D ∪ A| be the cardinality of the union of the sets D and A. Let |O ∪ D ∪ A| be the cardinality of the union of the three sets: O, D, and A. The former represents the number of changed IPCs while the latter represents the total number of IPCs. As D is included in O, then |O ∪ D ∪ A| could be simplified as |O ∪ A|. The change ratio (*ChRatio*) between r_1 and r_2 is then calculated as in Eq. (3):

$$\text{ChRatio} = \frac{|D \cup A|}{|O \cup A|} \tag{3}$$

where $|O| > 0$, $|D| \geq 0$, $|A| \geq 0$, $D \subseteq O$, $O \cap A = \emptyset$.

Thus, the architectural stability metric ASM is then calculated as in Eq. (4):

$$\text{ASM} = 1 - \text{ChRatio} \tag{4}$$

Both Eq. (3) and Eq. (4) could be simplified as in Eq. (5):

$$\text{ASM} = \frac{|0| - |D|}{|0| + |A|} \tag{5}$$

The ASM value ranges from 0 to 1 where ASM value of 0 indicates the highest possible amount of changes between r_1 and r_2 (i.e., unstable architecture) and ASM value of 1 indicates the lowest possible amount of changes between r_1 and r_2 (i.e., stable architecture).

2.3 Heuristic Algorithms

Heuristic algorithms have been successfully applied to solving a software packaging optimization problem in the presence of many extrema along with many parameters and in the presence of conflicting constraints [23]. Examples of such algorithms are hill-climbing (HC), simulated annealing (SA), tabu search, and genetic algorithms (GA). In our experiments, we use GA as it has been one of the most popular among others in solving complex problems in general and SBSE problems in specific; especially in case of packaging medium and big-size systems [23][28].

GA is inspired by the survival-of-the-fittest phenomena in biological evolution. It searches for optimal solutions by sampling the search space at random and creating a set of candidate solutions called a ‘population’. These candidates (called individuals) are combined and mutated to evolve into a new generation of solutions that is sought be fitter. Combination is done through crossover, which is fundamental to GA and provides a mechanism for mixing individuals within the population. Mutation is instrumental in introducing new individuals, thereby preventing the search from stagnating. The next population of individuals is chosen from the parent and offspring generations in accordance with a survival strategy that normally favors fit individuals but nevertheless does not preclude the survival of the less fit. Interested readers are advised [29] to consult or details on the fundamentals of GA.

3. RESEARCH/EXPERIMENT DESIGN

3.1 Objective and Motivation

We can apply the goal-question-metric (GQM) template [30] to express the goal of our study as follows:

“The purpose of this study is *to investigate* the impact of *software packaging* on *modular structure stability* through *real-world systems* from the point of view of *software architects* in the context of *architectural design phase*. Accordingly, we use JHotDraw and AWT as a representative of state-of-the-art for real-world systems”.

Often, the original packaging was done manually. Practitioners typically try to structure their systems into components where each component is internally cohesive and loosely coupled to other components. We assume that this was the case in the systems we considered in our experiments. Our experiments are meant to investigate the effect of different software packaging on modular structure stability of the system. We use heuristic optimization GA to show that different structures may be able to achieve better stability.

We conducted experiments on two case studies: JHotDraw and AWT. We used consecutive releases of the software in each case study. Good software structuring is expected to result in small structure changes from one release to another. Intuitively, assessing the stability across non-consecutive releases might be a bit misleading as the software might have significantly evolved overtime where original structure might not be viable anymore. Accordingly, we assessed the stability across consecutive releases as it is desirable that moving from one release to another should not have significant structural changes. While, it would not matter much whether we consider older releases vs. new releases, we experimented with older releases in our case studies as they might be less mature and may experience more changes than later releases.

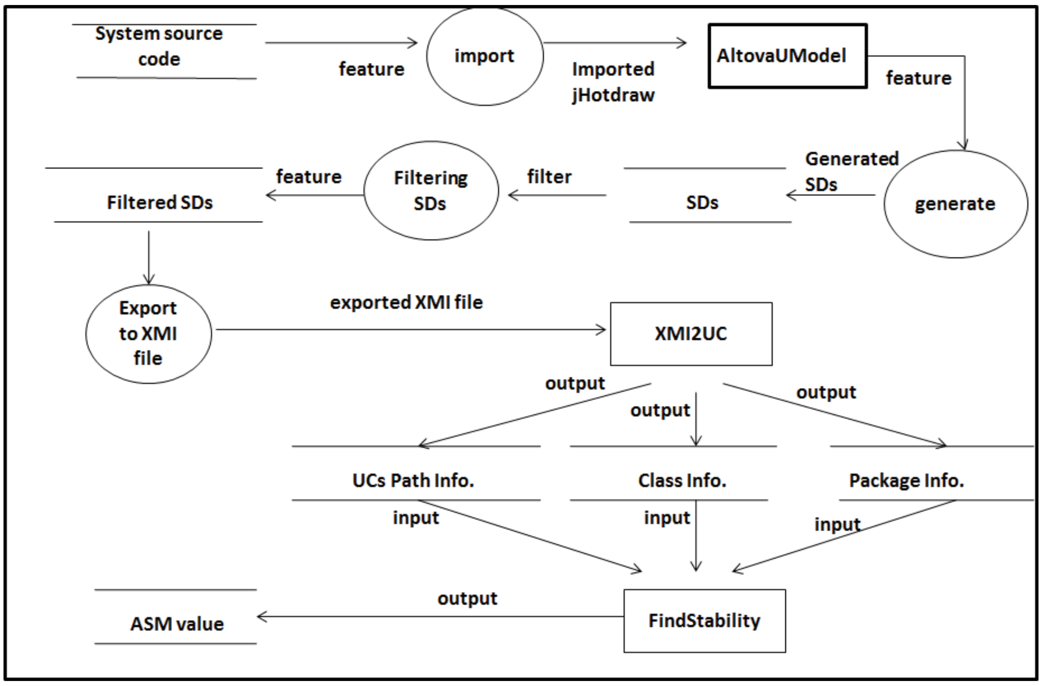


Figure 1 DFD to describe the reverse engineering process.

3.2 Data Collection

Clearly, packaging becomes much more important and more difficult as the size of the software increases. In trying to apply the Ebad and Ahmed's approach to real-world systems, we faced the "data scarcity" problem common to many software engineering research. We were not able to find requirement/design level artifacts such as UCs and SDs were available for public access. Accordingly, we considered the source code of an open-source project, JHotDraw and reverse engineered it to obtain the required SDs along with the corresponding list of classes. We applied the considered packaging approach and analyzed the observations according to ASM values. We used AltovaUModel[1] (Enterprise Edition version 2012 sp1); it is an OO unified modelling language (UML) software modeling tool which includes reverse engineering utilities. Although AltovaUModel can generate SDs from source code, the generated SDs are based on methods behavior at run time. The source code of real software systems has many statements representing the run time context such as control and loop statements . After generating the SDs, all runtime variables and messages shown in UML-SDs are eliminated via a filtering process. This makes the generated "runtime" SDs mimic the "functional" SDs that describe the "functional" behavior of UCs through objects and messages starting from the pre-condition to the post-condition. Details of the overall process are not included here; interested readers should consult [8][10]. Moreover, we used the Ebad and Ahmed's tool (XMI2UC) to extract UCs from the XMI documents generated by AltovaUModel. A detailed discussion on XMI2UC can be found in their work [31]. The above reverse engineering process could be modeled through data flow diagram (DFD) shown in Figure 1 (borrowed from [8][10])

[1]http://www.altova.com/umodel.html

4. REAL-WORLD CASE STUDIES

4.1 JHotDraw

JHotDraw[2] is an open source Java GUI framework meant for developing custom-made drawing editor applications. Because a wide amount of historical data of JHotDraw is available in its CVS repository, several researchers used JHotDraw as a case study in their research [7][8][31].

4.1.1 JHotDraw-Experiment Material

In our experiments we considered only these four packages [7]: *contrib*, *figures*, *standard* and *util* as they make up most of the application's core. Table 1 shows a brief description of each package.

Table 2 shows the statistics produced by the reverse engineering process described in Figure 1 for the three releases of JHotDraw project 5.1, 5.2, and 5.3. It also shows the OverallPackaging and ASM values. Interested readers can consult [31] for looking at a sample of UCs generated by XMI2UC tool.

4.1.2 JHotDraw-Experiment Settings

We used a Grouping Genetic Algorithm (GGA), which is particularly well suited for grouping problems [29]. We conducted the experiment using the Evolver software tool, version 6.0 2012. Evolver[3] is a GA solver offered as a plug-in for Microsoft Excel. As it is the practice when using GA, we used trial and error to set the GA parameters such as population size, crossover rate, and mutation rate. Table 3 shows the settings for the GA parameters. It is worth noting here that

[2]www.jhotdraw.org
[3]www.palisade.com/evolver

Table 1 A brief description of each package in JHotDraw.

Package name	Description
Contrib	Classes that where contributed by others.
Figures	A kit of figures together with their associated support classes (tools, handles).
Standard	The standard package provides standard implementations of the classes defined in the framework package.
Util	This package provides generally useful utilities that can be used independent of JHotDraw.

Table 2 Raw data of three JHotDraw releases.

	r 5.1	r 5.2	r5.3
No. of packages	4	4	4
No. of classes after filtering	62	68	82
No. of UCs generated by XMI2UC tool	130	131	165
OverallPackaging (original)	0.25	0.27	0.34
ASM (original)	N/A	0.76	0.55

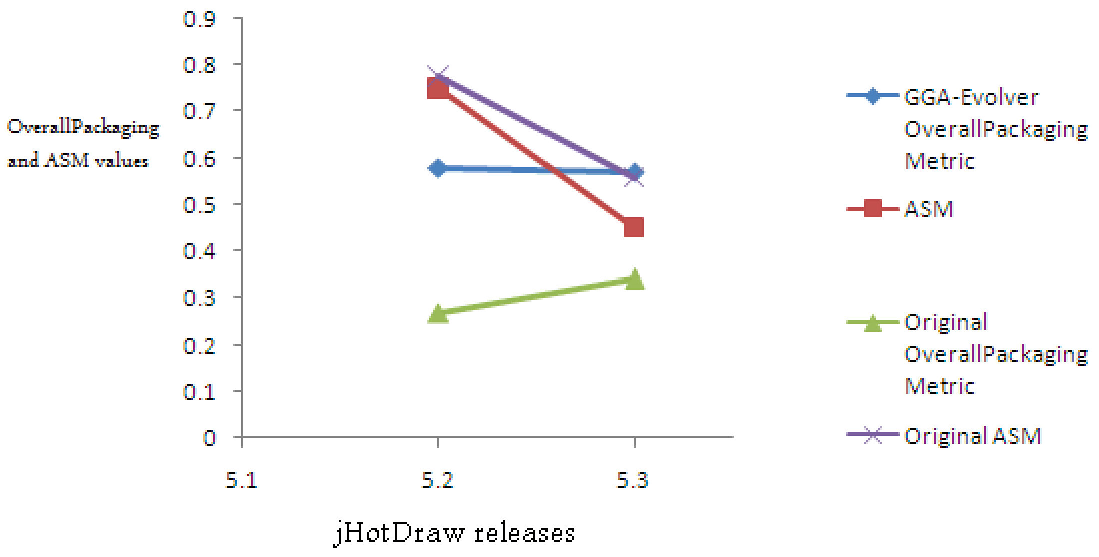


Figure 2 OverallPackaging and ASM values for JHotDraw releases where the initial population is the '*from scratch*' configuration.

in Evolver, a "trial" means a call to the function evaluator i.e., OverallPackaging metric. During an optimization, Evolver generates several trial solutions and uses GA to continually improve outcomes of each trial.

4.1.3 JHotDraw-Experiment Results

Three experiments were conducted. The first experiment was performed using '*from scratch*' initial population. Figure 2 shows the corresponding results. It particularly compares the original packaging of the JHotDraw releases (i.e., the system structure before using the considered packaging) against the suggested packaging using our packaging approach; the comparison was in terms of OverallPackaging and ASM values. The figure shows that GGA-Evolver suggested a better packaging than the original one in terms of the OverallPackaging value that was higher than that of the original packaging with the releases (0.58 vs. 0.27 and 0.57 vs. 0.34 in r5.2 and r5.3, respectively). However, the stability values (i.e., ASM) were not much better; they were slightly less than those of the original packaging (0.75 vs. 0.78 and 0.45 vs. 0.56 in r5.2 and r5.3, respectively).

Intuitively, we expect to obtain better stability if original JHotDraw packaging was used in the initial population. Accordingly, we conducted another experiment using the same parameter settings of the first experiment shown in Table 3 except that we GGA did not start '*from scratch*'; rather, the original packaging was seeded in the initial population (i.e., using the existing configuration). Figure 3 describes this process for the three releases.

The result here did not significantly differ from the result of first experiment. The stability (ASM) value of the suggested packaging by GGA-Evolver was slightly less than that of the original JHotDraw packaging especially with r5.3 release (i.e., 0.51 vs. 0.56). However, as depicted in Figure 4, the stability of the suggested packaging and the original one were the same (i.e., 0.78) with case of r5.2 release.

The third experiment showed improvement in both OverallPackaging and stability. In this experiment, we changed the weights of the main two components of the OverallPackaging,

Table 3 Parameter settings for the GGA-Evolver experiment using JHotDraw.

Parameter	Setting
Crossover	0.5
Mutation	0.06
Population Size	100
Number of Trials/Generations	2000 with progress i.e., it stops when no improvement within the last 250 trials. These values are chosen because of the limitation in run time.
Weights	Weights of the OverallPackaging's components are all set to 0.5
Initial Population	Two options: (1) The initial configuration used is based on placing each class in one package; this is referred to as ' *from scratch*' packaging, and (2) An existing JHotDraw configuration was used as the highly fit initial population
Machine features	PC with Intel®Core™i3-530 Processor,2.93 GHz, 4.00 GB RAM (3.43 usable). Windows 7 Enterprise.

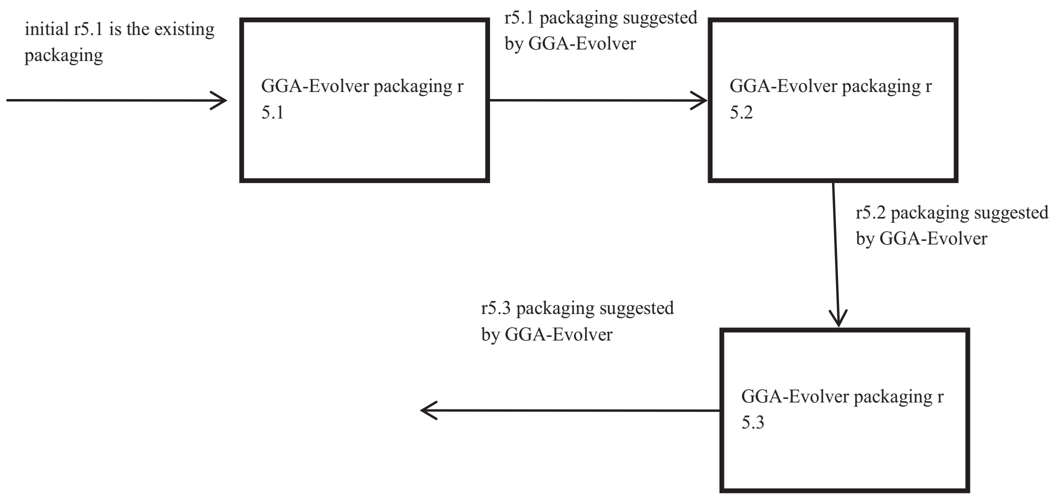

initial r5.1 is the existing packaging
GGA-Evolver packaging r 5.1
r5.1 packaging suggested by GGA-Evolver
GGA-Evolver packaging r 5.2
r5.2 packaging suggested by GGA-Evolver
r5.3 packaging suggested by GGA-Evolver
GGA-Evolver packaging r 5.3

Figure 3 Initial packaging used in JHotDraw packaging.

UC coverage (w_U) and class relevancy (w_C). The rationale of such change came from the functionality perspective of OverallPackaging in which packaging classes in a small number of packages means that the UC coverage aspect has a greater contribution to the fitness function than class relevancy. Therefore, we carried out the third experiment using the original configuration of JHotDraw as the initial population and with weights 0.55 for UC coverage and 0.45 for class relevancy; that is, $w_U = 0.55$ and $w_C = 0.45$. The option of using the existing JHotDraw configuration as initial population is good starting population in packaging the system, the strategy for finding highly fit individuals may vary depending on the availability of existing packaging: If a suitable packaging is given (e.g. by the package structure of a Java system as in JHotDraw case), we use it as the highly fit initial population. Actually, if appropriate initial packaging is not available, GGA-Evolver will just require more time to be able to propose the same quality packaging.

With the above settings, the GGA-Evolver achieved better results than the previous two experiments; in terms of both OverallPackaging and ASM values. Table 4 shows the results of five different runs with these settings. In these runs, the best suggested packaging for release 5.2 (at Run 3 where ASM is 0.808) is used as initial population for release 5.3. After repeating that run three times, the OverallPackaging value was not changed i.e., the suggested packaging on release 5.3 is best possible one.

Table 5 summarizes the best, worst, and average values extracted from the five runs shown in Table 4.

The best result achieved by of GGA-Evolver is summarized in Figure 5 that states that the OverallPackaging values are 0.56 and 0.54 for$r_{5.2}$ and $r_{5.3}$ respectively compared to the original values 0.26 and 0.27 for$r_{5.2}$ and $r_{5.3}$ respectively. The stability is 0.81 and 0.52 for$r_{5.2}$ and $r_{5.3}$ respectively.

The above experiments indicate that the more the system is functionally packaged (i.e., the OverallPackaging value increases), the more stable its modular structure is (i.e., the ASM value also increases). This result was achieved using these parameters: (1) using the original configuration of JHotDraw as the initial population, and (2) weights of the two main components of OverallPackaging, UC coverage (w_U) and class relevancy (w_C), are set to 0.55 and 0.45, respectively.

4.2 AWT

The Java AWT (Abstract Windowing Toolkit) Library is a collection of classes for creating lightweight user interfaces and for painting graphics and images. It is part of the standard Java platform.

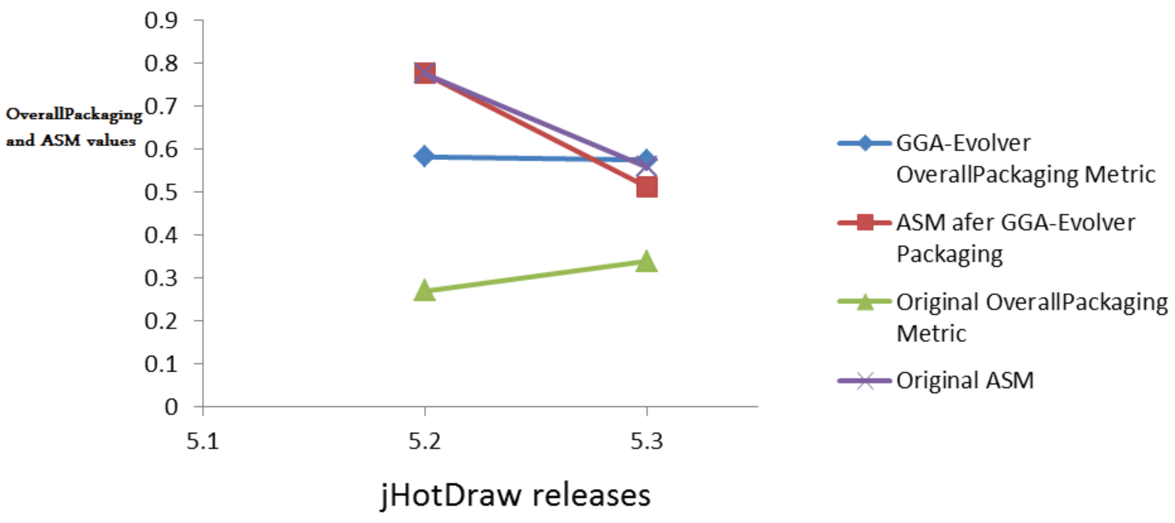
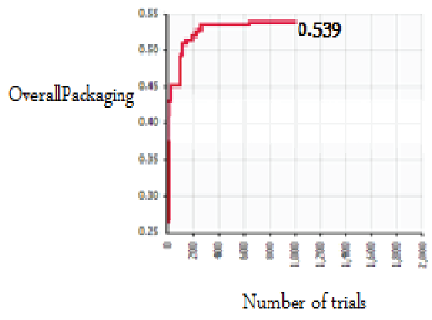
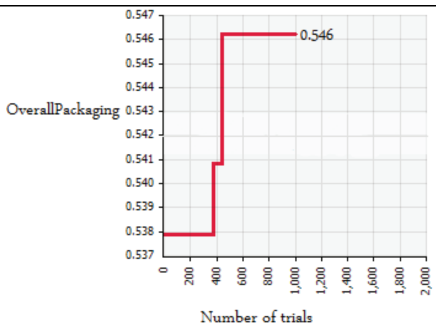
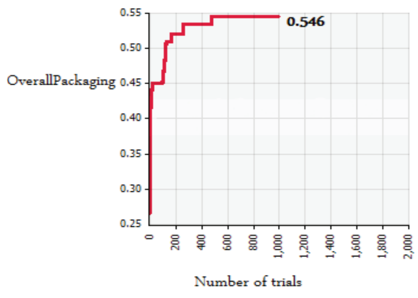
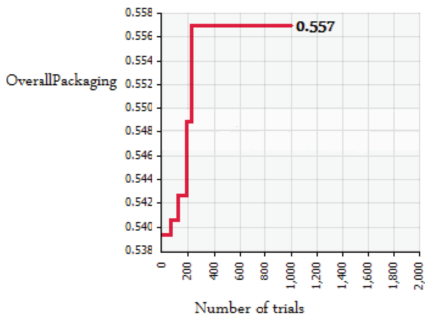
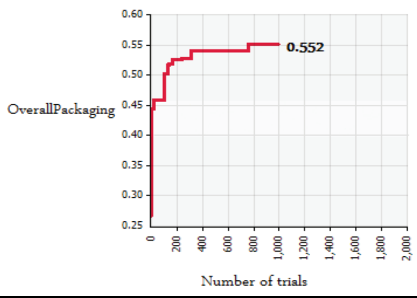
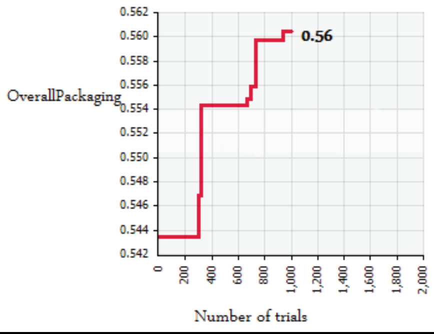


Figure 4 OverallPackging and ASM values for JHotDraw releases where the initial population is the existing packaging configuration.

Table 4 GGA-Evolver Results for the three JHotDraw releases.

Run	OverallPackaging metric r 5.1	OverallPackaging metric r 5.2	ASM (r5.1,r5.2)
R1			0.785
R 2			0.779
R 3			0.808

4.2.1 AWT-Experiment Material

The *java.awt* is the main package of the AWT Library. This package is structured into relatively large packages, we concentrated on the nine main packages that make up the library's core: *color, datatransfer, event, font, geom, im, image, image.renderable, and print*[4]. Table 6 shows a brief description of each package. For simplicity, we refer to *java.awt* as AWT.

[4]AWT Documentation. available https://docs.oracle.com/javase/8/docs/api/index.html?java/awt

Table 4 Continued.

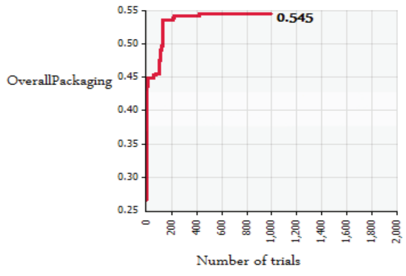
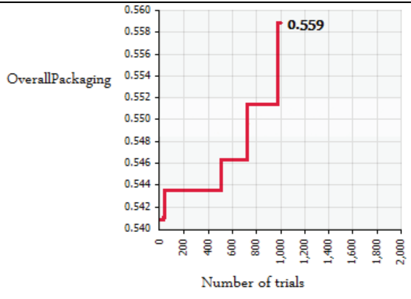
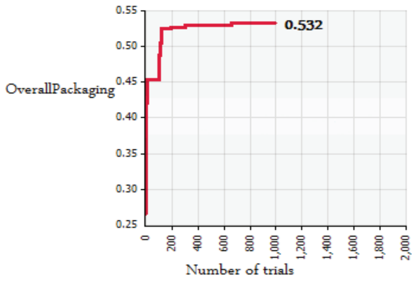
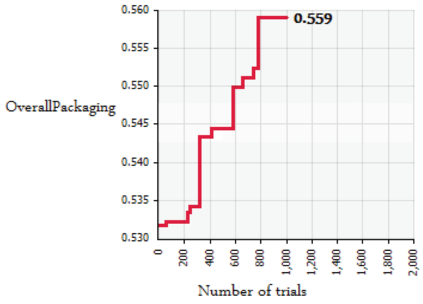
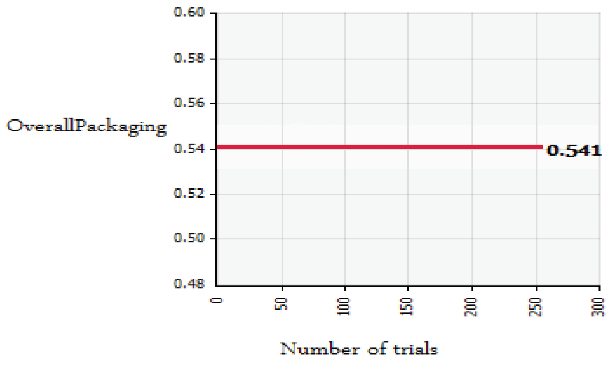
R 4			0.776
R 5			0.74
Run	OverallPackaging metric r 5.3		ASM (r5.2,r5.3)
R1			0.491
R 2	0.541, As the above plot		0.506
R 3	0.541, As the above plot		0.516
R 4	0.541, As the above plot		0.502
R 5	0.541, As the above plot		0.513

Table 5 Reports the best, the worst, and the average values of OverallPackaging matric for five runs shown in Table 4.

Statistic	OverallPackaging r 5.1	OverallPackaging r 5.2	ASM (r5.1, r5.2)	OverallPackaging r 5.3	ASM (r5.2, r5.3)
Best	0.552	0.559	0.808	0.541	0.516
Worst	0.532	0.546	0.74	0.541	0.491
Avg.	0.543	0.556	0.778	0.541	0.506

Compared to JHotDraw case study, AWT library is larger in size. The size does not only depend on the number of classes but also on the number of UCs generated by XMI2UC tool which in turn depends on the number of methods in the whole system. For instance, some classes in AWT have tens of methods e.g., BufferedImage and ColorModel; such cases would generate tens of UCs/SDs by Altova. We filter these UCs/SDs by XMI2UC tool to avoid any overlapping or duplication (Ebad & Ahmed 2012). Table 7 shows the raw data of AWT library of JDK 1.4 and 1.5 which are used in our experiment. It indicates to higher number of UCs in AWT compared to JHotDraw. Therefore, we reduced the number of trials/generations in GGA-Evolver experiment to 300 due to the experiment run time.

4.2.2 AWT-Experiment Settings

It is worth noting here that, due to the large number of classes, we restricted the maximum number of packages to 20 to avoid having large number of packages with very few number of classes each, which is not practically acceptable. Figure 6 shows the initial population used in the GGA-Evolver experiment for packaging AWT Library.

4.2.3 AWT-Experiment Results

Contrary to the original AWT packaging having 0.44 and 0.47 OverallPackaging values in release 1.4 and 1.5 respectively (see Table 8), the suggested packaging generated by GGA-Evolver is

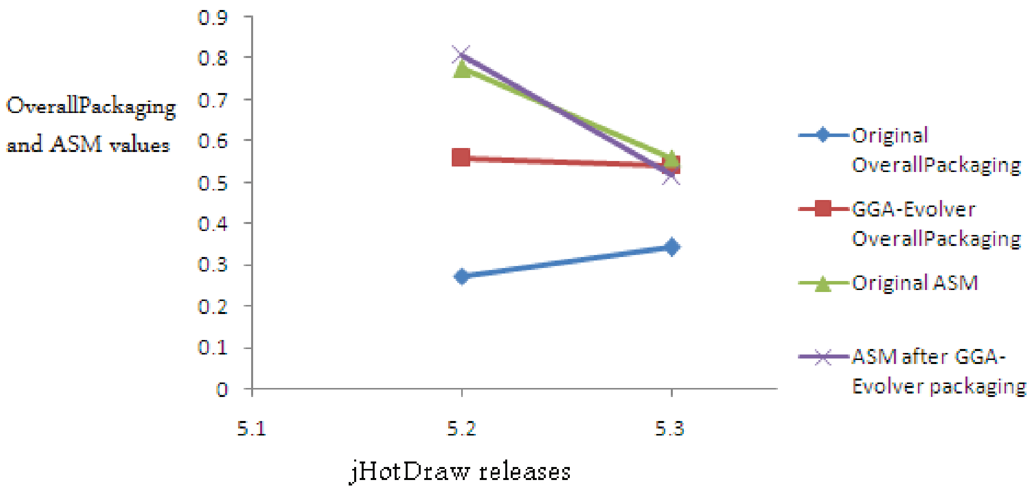


Figure 5 The GGA-Evolver best result in terms of OevrallPackaging metric and ASM values. This result was achieved using (1) original configuration as the initial population, and weights of the UC coverage (w_U) and class relevancy (w_C) were set to 0.55 and 0.45, respectively.

Table 6 A brief description of each package in java.awt

Package name	Description
Color	provides classes for color spaces
Datatransfer	provides interfaces and classes for transferring data between and within applications
Event	provides interfaces and classes for dealing with different types of events fired by AWT components
Font	provides classes and interface relating to fonts.
Geom	provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry
Im	provides classes and interfaces for the input method framework
Image	provides classes for creating and modifying images
image.renderable	provides classes and interfaces for producing rendering-independent images
Print	provides classes and interfaces for a general printing API.

Table 7 Raw data of AWT Library.

	r 1.4	r 1.5
No. of packages	9	9
No. of classes after filtering	81	82
No. of UCs generated by XMI2UC tool	196	271
OverallPackaging (original)	0.44	0.47
ASM (original)	N/A	0.31

Table 8 Summarizes the settings of GGA-Evolver experiment.

Parameter	Setting
Crossover	0.5
Mutation	0.06
Population Size	100
No. of Trials	300
Initial Population	Using the existing packaging
Constraint	Maximum number of generated packages is 20
Machine Features	PC with Intel®Core™i3-530 Processor,2.93 GHz, 4.00 GB RAM (3.43 usable). Windows 7 Enterprise

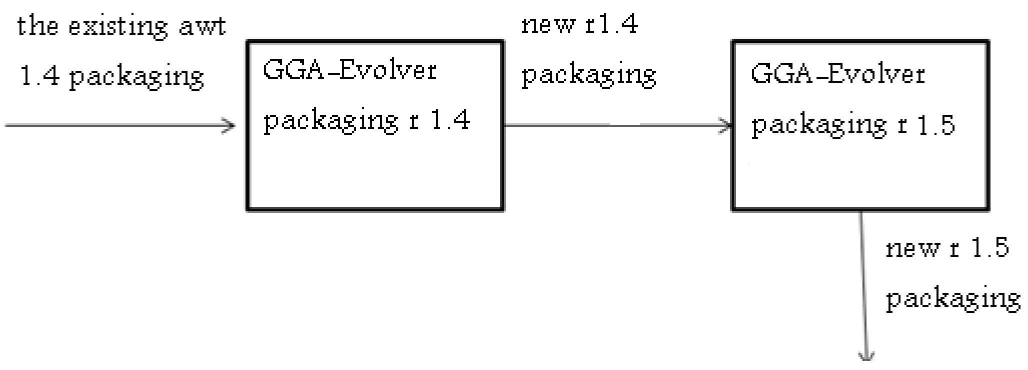


Figure 6 Initial packaging used in AWT packaging.

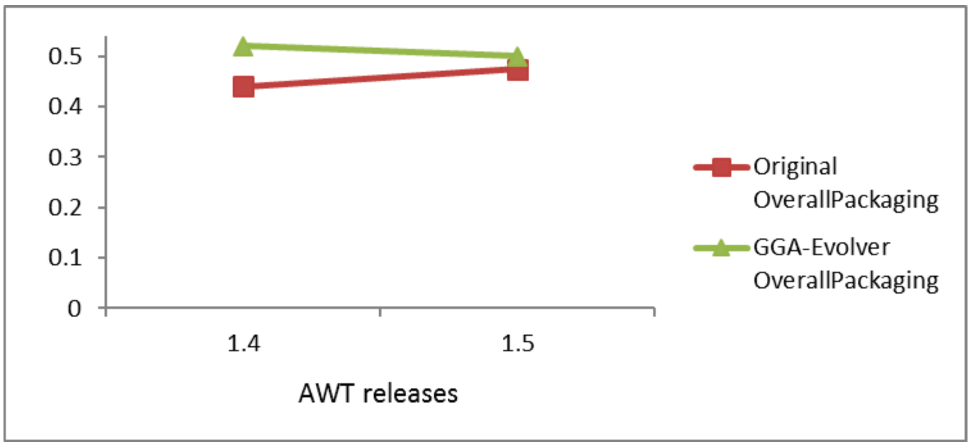


Figure 7 OverallPackaging metric for AWT releases

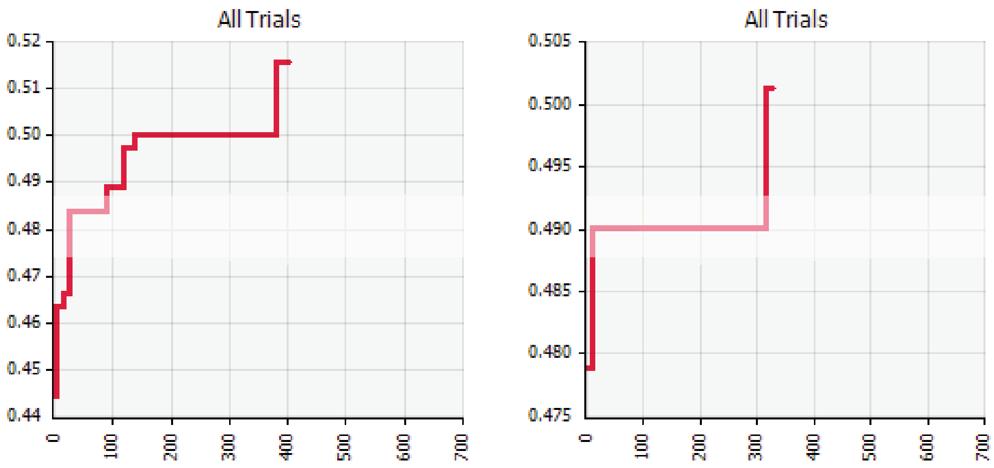


Figure 8 Trials course in AWT packaging.

better; it has 0.52 and 0.50 values for both releases respectively. This result is shown in Figure 7.

Figure 8 illustrates the trials' course during the GGA-Evolver execution for packaging AWT 1.4 and 1.5.

Figure 9 illustrates the stability values. Despite the stability of AWT releases automatically packaged using the considered approach was not better than that of the corresponding original releases, it was at least comparable (0.28 versus 0.31). We think there are several "better" configurations to improve the stability value. More investigation is then needed to find the optimal values for all considered parameters: those related to the GA and those related to the weights.

4.3 A General Summary

As you have seen, we applied Ebad and Ahmed's packaging approach to two open-source projects (JHotDraw with three releases and AWT with two releases). The measurements achieved from our structures are, if not better, close to the measurements archived from the original structure. It is worth noting here that this observation is two-fold: 1) shows that automatic packaging can get results that are as good as or even better than experts'; and 2) confirms that different structuring in effect affects the system stability. The former is interesting to the software industry as it would save time and money during

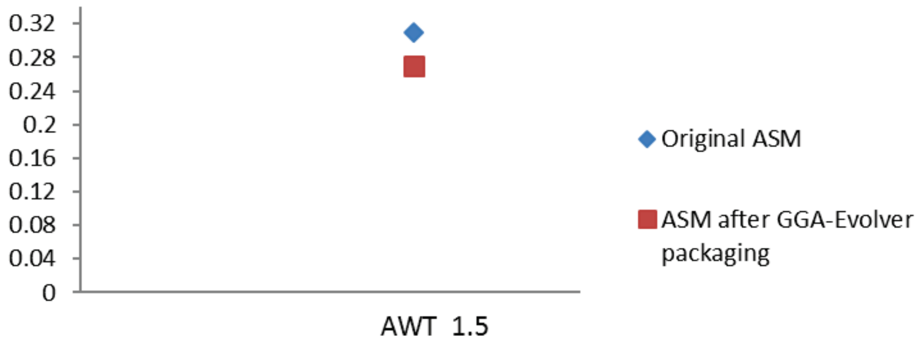


Figure 9 ASM value of the original and GGA-Evolver packaging for AWT.

the architecture design effort. The latter is also interesting since it offers an approach to improve future stability of the system.

5. THREATS TO VALIDITY

As with any empirical study, there are some threats to the validity of the results of this study. Here is a list of what we consider internal and external threats to our study:

- *Internal validity*: the most important concern is the use of different tools for data collection. This threat is not expected to be of serious concern because we used readily available commercial tools such as Evolver and Altova. Although the other tool, XMI2UC, is developed for experimentation purposes only, several previous studies relied on it and we reported sufficient information about it. Another factor that may lead to internal threat is the selection of parameter setting of the heuristic technique used in the packaging approach. We intend to carry out more experiments to obtain appropriate values for such settings especially weights of the different components of OveralPackaging.
- *External validity*: the systems tested in this study were open-source ones. A threat here is that we used Altova and XMI2UC tools to reverse engineer the considered open-source systems; which may not be good representative of actual requirements artifacts.

6. RELATED WORK

In this section, we briefly discuss the existing approaches on software packaging; we also discuss metrics to measure stability.

6.1 Packaging Approaches

Mancoridis et al. [5] developed a tool called Bunch [6] that uses HC and GA to aid its clustering algorithms. They defined the modularization quality (MQ) of a system as an objective function to express the trade-off between intra- and inter-connectivity attributes at the module level. Doval et al. [26] applied a GA in the Bunch tool to find the optimal grouping using MQ as the fitness function. Mitchell & Mancoridis [33] integrated a new version of MQ into Bunch; the new version supports weights. Liu et al [34] used GGA to present a method for decomposing a large number of objects into mutually exclusive groups. The min-cut algorithm was used in the clustering approach presented by Chiricota et al. [4]. The resultant software structure was evaluated by applying the original MQ metric proposed by [5]. Bauer and Trifu [3] proposed an approach that combines clustering with pattern-matching techniques to produce meaningful decomposition. Using GGA, Seng et al. [7] proposed a decomposition approach with a multi-modal fitness function that included cohesion, coupling, complexity, cycles, and bottlenecks. In their approach, Abdeen et al. [1] defined the fitness function using several measures inspired by the principle stating that "*packages are desired to be loosely coupled and cohesive to a certain extent*". Compared with previous approaches, this modularization approach allows maintainers to define certain constraints. Alkhalid et al. [35] packaged classes from the source code using two approaches: (1) fixed number of packages, and (2) variable number of packages. They evaluated their approaches using the similarity measure of Lung et al. [36]. Corazza et al. [37] worked on six parts of the source code: class, attribute, method and parameter names, comments, and source code statements. They grouped source files according to the lexical information using a hierarchical clustering algorithm. Beck & Diehl [24] found that none of the investigated forms of coupling (e.g., structural dependencies, evolutionary coupling, and conceptual similarity) reflects the modular structure of the studied systems. Risi et al. [38] automated the architecture recovery process of systems. They used (1) latent semantic indexing (LSI) to get similarities among software entities, (2) the k-means clustering algorithm to form groups of entities, and (3) fold-in and fold-out mechanisms to improve computational time. Bavota et al. [39] focused on a specific restructuring: given a package with poor cohesion, decompose it into smaller and meaningful packages that have higher cohesion. To measure package cohesion, they used information-flow-based coupling (ICP) [40] and conceptual coupling between classes (CCBC) [41]. The measures capture structural and semantic relationships between classes, respectively. They then used an aggregated measure to determine classes that should belong together in a package. Ebad and Ahmed [8] proposed a new packaging approach that was based on UCs which in turn are realized by sequence diagrams (SDs). This approach came to reflect the functionality packaging classes of the conceptual class model developed during the requirements engineering phase. Besides theoretical validation, the approach was validated empirically using different heuristic algorithms. Paixao et al. [28] discusses developers' perception on coupling and cohesion based on different software systems. The results reveal that developers agreed with the fitness functions measured to calculate coupling and cohesion.

6.2 Measures of Stability at Architecture Level

Martin [42] hinges the stability of a system on the dependency between the different packages in the system. However, it was not clear how the metric assesses the impact of evolution on the software architecture. Bansiya [19] proposed an approach to evaluate the architectural stability of frameworks by means of a suite of OO design metrics. These metrics are computed from the class model of successive releases of frameworks; corresponding measurements are compared to determine the extent-of-change in the structural characteristics of the different releases. Alshayeb [16] used Bansiya's approach [19] to assess the effect of software refactoring on architecture stability. He recommended the designers interested to optimize their design for architecture stability to avoid using refactoring methods that affect the class hierarchy. Instead, the designers can use those methods that affect field/method levels. Jazayeri [14] has applied retrospective analysis to twenty releases of a large telecommunication software system. The analysis uses simple metrics such as software size metrics (e.g. module size, number of modules changed, and the number of modules added in the different releases); coupling metrics; and color visualization to outline the evolution pattern of a software system across releases. Bahsoon and Emmerich [11] used a predictive approach for measuring architectural stability. They present an architectural review approach, called ArchOptions, with stability as the primary review objective. Tonu et al. [43] have identified complexity, cohesion, and coupling as the factors that contribute to the architectural stability. They presented a metric-based approach to assess architecture stability. Both retrospective and predictive analyses are applied in the approach. By using the metric-based approach, they identified where the architecture of the systems used in the experiment become stable. Ahmed et al. [20] introduced an approach to measure architectural stability of an OO system by using similarity metrics. These metrics compare the base version of a system with the next versions. Then a regression line is generated from these similarity values. A higher value represents a stable architecture. Raemaekers et al. [44] introduced a way to measure interface and implementation stability of a library. To that end, they proposed four metrics which provide different insights in both implementation and interface stability. Aversano et al. [45] presents an empirical study aimed at assessing software architecture stability and its evolution along the software project history. Two metrics were defined based on a previous work [21]. Ebad and Ahmed [10] proposed architectural stability metric (ASM) based on the inter-package connections (IPCs); such connections were represented by message passing messages conduct at architectural design phase.

6.3 Gaps in the Prior Work

In conclusion, most of the researchers evaluated the effect of their packaging approaches on software using internal quality attributes such as cohesion and coupling [1][7][24][28][34][35]. However, some researchers considered somehow external quality attributes such as stability but their packaging (refactoring) was not at architectural design level but at source code level with different granularity levels [16][24]. The literature survey revealed there is a lack of studies to assess the effect of packaging on software stability at architecture level. In this paper, we assess the impact of the functionality-based packaging on the software modular structure stability. Key in the packaging approach proposed by Ebad and Ahmed (2015a), chosen in our work herein, is that decomposing should be based on functionality view. Effective functionality based packaging allows independent offerings and reuse. To the best of our knowledge, that approach is the first functionality-based modularization approach that can be used during the architecture design phase using conceptual models. It is based on the system UCs and their corresponding SDs, which offer the functional view of the system.

7. CONCLUSION AND FUTURE WORK

Effective software packaging is meant to group OO classes into packages so that each package performs a single task that is entirely carried out in that package. Because this activity is concerned with the structure without affecting its internal behavior, packaging aims at improving the software quality manifested in improving its maintainability, and reducing the impact of future changes. This is achieved via improving the software architecture stability. In this paper we assess the impact of system packaging on the modular structure stability. This assessment is done through conducting different experiments on two real-world case studies (JHoDraw or AWT) using a functionality-based packaging approach and architectural stability metric proposed recently by Ebad and Ahmed. While it might be a common knowledge that functionally cohesive and loosely coupled structures are expected to be stable over time, we actually use this common knowledge to validate and confirm that using the *OverallPackaging* measure/fitness is meaningful and indeed can lead to functionally modular structures. Experiments show that using *OverallPackaging* can guide the structure design to be more stable. To the best of our knowledge, the approach presented is the first one that uses UCs to represent the functionality during the structure design. We used GA as a heuristic technique to allow evolving optimal package structures. Experiments show that GA parameter settings (for example, the initial population and weights) have a significant impact on the GA solution quality.

It is worth noting here that each run of each experiment on JHoDraw or AWT take considerable amount computational time (around 20 hours). However, we do not consider this to be an issue since, in practice, the packaging activity spans over a period of time. Actually, for large systems, it could typically take designers days and even weeks to structure the system. The proposed automated approach is expected to be used offline, and not in real-time. Having the program running overnight would be very practical. In general, all optimization parameters are not easy to address in this research especially that the number of parameters in our case is not small. Additionally, we plan to apply other heuristic techniques (e.g., simulated annealing and particle swarm optimization) for comparison with the results of GGA-Evolver reported in this work. The last point to further research includes working on multi-level packaging. Recall that the packaging process may continue recursively. An architect may decide to package OO packages rather than OO classes. In other words, packaging the packages into a higher level (second level); and probably packaging the resultant packages (third level), and so forth. In such a context, the package is considered to be a big logical class.

ACKNOWLEDGMENT

The authors wish to acknowledgeKing Fahd University of Petroleum and Minerals (KFUPM) for providing the facilities to carry out this research.

REFERENCES

1. Abdeen H, Ducasse S, Sahraouiy H, Alloui I. (2009) Automatic package coupling and cycle minimization. In: Proceedings of the 16^{th} Working Conference on Reverse Engineering (WCRE). Lille, France, pp.103–122.
2. Abdeen H, Ducasse, S, Sahraouiy, H, Alloui, I. (2011) Modularization metrics: assessing package organization in legacy large object-oriented software. In: Proceeding of the 18^{th} Working Conference on Reverse Engineering (WCRE), USA, pp.394–398.
3. Bauer, M, Trifu, M. (2004) Architecture-aware adaptive clustering of OO systems. In: European Conference on Maintenance and Reengineering (CSMR 04), Karlsruhe, Germany, pp. 3–14.
4. Chiricota, Y, Jourdan, F, Melancon, G. (2003) Software components capture using graph clustering. In: the 11^{th} IEEE International Workshop on Program Comprehension (IWPC), USA.
5. Mancoridis S, Mitchell B, Rorres C, Chen Y, Gansner R. (1998) Using automatic clustering to produce high-level system organizations of source code. In Proceedings of the International Workshop on Program Comprehension (IWPC). USA, pp. 45–53.
6. Mancoridis S, Mitchell B, Chen Y, Gansner R. (1999) Bunch: a clustering tool for the recovery and maintenance of software system structures. In: Proceeding of the IEEE International Conference on Software Maintenance, USA, pp 50–59.
7. Seng O., Bauer M., Biehl M., Pache G. (2005) Search-based improvement of subsystem decompositions. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'05), USA.
8. Ebad S, Ahmed M. (2015) Functionality-based software packaging using sequence diagrams. Software Quality Journal, 23(3), 453–481.
9. Rozanski N. and Woods, E. (2012) Software systems architecture: working with stakeholders using viewpoints and perspectives, 2^{nd} edition, Addison Wesley.
10. Ebad S, Ahmed M. (2015) Measuring stability of object oriented software architectures, IET Software, 9(3), 76–82.
11. Bahsoon R, Emmerich W. (2004) Evaluating the stability of software architectures with real options theory. In Proceedings of the 20^{th} IEEE International Conference on Software Maintenance, Chicago Illinois, USA, pp. 11–17.
12. Grosser D, Sahraoui HA, Valtchev P. (2003) An analogy-based approach for predicting design stability of java classes. In: Proceedings of the 9th International Software Metrics Symposium, Sydney, Australia, pp. 252–262.
13. Alshayeb M, Li W. (2004) An empirical study of system design instability metric and design evolution in an agile software process. Journal of Systems and Software, 74(3), 269–274.
14. Jazayeri M. (2002) On architectural stability and evolution. In: Proceedings of the 7^{th} Ada-Europe International Conference on Reliable Software Technologies, Vienna, Austria, pp 13–23.
15. Le D.M., Behnamghader P., Garcia J., Link, D., Shahbazian, A., and Medvidovic, N. (2015) An empirical study of architectural change in open-source software systems. In Proceedings of the IEEE/ACM 12^{th} Working Conference on Mining Software Repositories, 235–245.
16. Alshayeb M. (2011) The impact of refactoring on class and architecture stability. Journal of Research and Practice in Information Technology, 43(4), 269–284.
17. Mattsson M, Bosch J. (2000) Stability assessment of evolving industrial object-oriented frameworks. Journal of Software Maintenance: Research and Practice, 12(2): 79–102.
18. Mattsson M, Bosch J. (1999) Characterizing stability in evolving frameworks. In: Proceedings of Technology of Object-Oriented Languages and Systems, Nancy, France, pp.118–130.
19. Bansiya J. (2000) Evaluating framework architecture structural stability. ACM Computing Surveys (CSUR), 32(1), doi: 10.1145/351936.351954.
20. Ahmed M, Rufai R, Alghamdi J, Khan S. (2003) Measuring architectural stability in object oriented software. In: Proceedings of 1^{st} Workshop on Stable Analysis Patterns: A True Problem Understanding with UML, San Francisco, CA, pp. 21–28.
21. Olague H, Etzkorn L, Li W, Cox G. (2006) Assessing design instability in iterative (agile) object-oriented projects. Journal of Software Maintenance and Evolution: Research and Practice, 18(4): 237–266.
22. Hassan YS. (2007) Measuring Software Architectural Stability Using Retrospective Analysis. Master Thesis, King Fahd University of Petroleum & Minerals, Saudi Arabia.
23. Ebad S, Ahmed M.(2011) Software packaging approaches – a comparison framework. In: Proceedings of the 5^{th}European Conference on Software Architecture (ECSA 2011), Essen, Germany, pp. 438–446.
24. Beck F, Diehl S. 2011. On the congruence of modularity and code coupling. In: Proceedings of the 19^{th}ACM SIGSOFT Symposium and the 13^{th}European Conference on Foundations of Software Engineering(ESEC/FSE'11). 354–364.
25. Jacobson I. (1992) *Object-oriented software engineering: a use case driven approach*. Addison-Wesley.
26. Doval D, Mancoridis S, Mitchell B. (1999) Automatic clustering of software systems using a genetic algorithm. In: Software Technology and Engineering Practice (STEP '99). Pittsburgh, PA, pp. 73–81.
27. Ebad, S. (2012). *Functionality-based software packaging approach for higher architecture stability*. Ph.D. Dissertation. Department of Computer Science and Engineering, King Fahd University of Petroleum and Minerals (KFUPM), Saudi Arabia.
28. Paixao, M., Harman, M., Zhang, Y., Yu, Y. (2018). An Empirical study of cohesion and coupling: balancing optimisation and disruption. IEEE Transactions on Evolutionary Computation, 22(3): 394–414.
29. Falkenauer E. (1998) Genetic Algorithms and Grouping Problems. Wiley, New York.
30. Basili, V.R. and Weiss, D.M. (1984) A method for collecting valid software engineering data, IEEE Transactions on Software Engineering, 10(6): 728–38.

31. Ebad S, Ahmed M. (2012) XMI2UC: an automatic tool to extract use cases from object-oriented source code. In Proceeding of the International Conference on Advancements in Information Technology (ICAIT 2012), Hong Kong.
32. Lu, G, Bahsoon, R., Yao, X. (2010) Applying elementary landscape analysis to search-based software engineering. In: Proceedings of the 2^{nd}International Symposium on Search Based Software Engineering (SSBSE '10), Benevento, Italy, pp. 3–8.
33. Mitchell B, Mancoridis S. (2002) Using heuristic search techniques to extract design abstractions from source code. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02), USA.
34. Liu X, Swift S, Tucker A. (2001) Using evolutionary algorithms to tackle large scale grouping problems. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'01), USA.
35. Alkhalid A, Alshayeb M, Mahmoud S. (2011) Software refactoring at the package level using clustering techniques. IET Software, 5(3), 274–286.
36. Lung C-H, Xu X, Zaman M, Srinivasan A. (2006) Program restructuring using clustering techniques. The Journal of Systems and Software, 79 (9), 1261–1279.
37. Corazza A, Martino SD, Maggio V, Scanniello G. (2011) Investigating the use of lexical information for software system clustering. In: the 15^{th} European Conference on Software Maintenance and Reengineering, Mar 1–4, Oldenburg, Germany.
38. Risi M, Scanniello G, Tortora1 G. (2012) Using fold-in and fold-out in the architecture recovery of software systems. Formal Aspects of Computing (24), pp. 307–330.
39. Bavota G, Lucia AD, Marcus A, Oliveto R. (2013) Using structural and semantic measures to improve software modularization. Empirical Software Engineering, 18(5), pp. 901–932.
40. Lee Y, Liang B, Wu S, Wang, F. (1995) Measuring the coupling and cohesion of an object-oriented program based on information flow. In: International conference on software quality, Maribor, Slovenia, pp.81–90.
41. Poshyvanyk D, Marcus A, Ferenc R, Gyimóthy T. (2009) Using information retrieval based coupling measures for impact analysis. Empirical Software Engineering, 14(1):5–32.
42. Martin, R. (1997) *Stability, C ++ report*, pp. 54–60.
43. Tonu S, Ashkan A, Tahvildari L. (2006) Evaluating architectural stability using a metric-based approach. In: Proceedings of the Conference on Software Maintenance and Reengineering, Bari, Italy, pp. 261–270.
44. Raemaekers S, Deursen A, Visser J. (2012) Measuring software library stability through historical version analysis. In: The IEEE 28^{th} International Conference on Software Maintenance, Trento, Italy, pp. 378–387.
45. Aversano L, Molfetta M, Tortorella M. (2013) Evaluating architecture stability of software projects. In: Procedings of the 20^{th} Working Conference on Reverse Engineering, Koblenz, Germany, pp.417–424.