Softw Syst Model
DOI 10.1007/s10270-017-0614-9

REGULAR PAPER

Applying design patterns in the search-based optimization of software product line architectures

Giovani Guizzo[1] · **Thelma Elita Colanzi**[2] · **Silvia Regina Vergilio**[1]

Received: 23 August 2016 / Revised: 15 May 2017 / Accepted: 23 July 2017

Abstract The design of the product line architecture (PLA) is a difficult activity that can benefit from the application of design patterns and from the use of a search-based optimization approach, which is generally guided by different objectives related, for instance, to cohesion, coupling and PLA extensibility. The use of design patterns for PLAs is a recent research field, not completely explored yet. Some works apply the patterns manually and for a specific domain. Approaches to search-based PLA design do not consider the usage of these patterns. To allow such use, this paper introduces a mutation operator named "Pattern-Driven Mutation Operator" that includes methods to automatically identify suitable scopes and apply the patterns Strategy, Bridge and Mediator with the search-based approach multi-objective optimization approach for PLA. A metamodel is proposed to represent and identify suitable scopes to receive each one of the patterns, avoiding the introduction of architectural anomalies. Empirical results are also presented, showing evidences that the use of the proposed operator produces a greater diversity of solutions and improves the quality of the PLAs obtained in the search-based optimization process, regarding the values of software metrics.

Keywords Design pattern · Search-based software engineering · Software product line architecture

Communicated by Dr. Jeff Gray.

✉ Giovani Guizzo
gguizzo@inf.ufpr.br

Thelma Elita Colanzi
thelma.lopes@din.uem.br

Silvia Regina Vergilio
silvia@inf.ufpr.br

[1] Department of Informatics, Federal University of Parana, Av. Cel. Francisco H. dos Santos, 210, Jardim das Amêricas, Curitiba, PR 81531-970, Brazil

[2] Department of Informatics, State University of Maringá, Av. Colombo, 5790, Bloco C56, Maringá, PR 87020-900, Brazil

1 Introduction

A software product line (SPL) is generally defined as a set of products, sharing common features [28], which correspond to important SPL functionalities, generally visible for the users. In SPL engineering, the product line architecture (PLA) is an important artifact because it contains all commonalities and variabilities of an SPL, and it is used to derive the architectures of the products. A commonality is related to features and elements that are present in all products of the SPL, whereas a variability represents features and elements that change from one product to another. Variabilities are represented by variation points (where the variation occurs) and variants (what varies) [28].

The PLA design can be a difficult activity, as the SPL must accommodate the increasing complexity of the software products. In general, *unified modeling language* (UML [39]) class diagrams are largely used to model architectures in a more detailed level. Due to this, the usage of design patterns, such as the ones from the GoF catalog [14], can help the architect to solve some problems found in the PLA design.

Design patterns are common solutions widely used by developers in several projects to solve common design problems. In a detailed level of the development process, one of their main goals is to obtain high cohesion and low coupling of the architectural elements, which can lead to a better software reusability. However, the design of PLAs and the application of design patterns in the PLA context are still difficult for some architects, specially to novices. This is because the architect needs to deal with different architectural attributes, such as modularity and extensibility. In addition to this, he/she needs to identify the best patterns to be applied in a given situation.

Search-based approaches, such as multi-objective optimization approach for PLA design (MOA4PLA) [6,31,32], can help the architect in the PLA design. MOA4PLA includes a representation for the architecture, a set of mutation operators, which are specific for PLAs by focusing on feature modularity. These operators are used by search-based algorithms, such as multi-objective evolutionary algorithms (MOEAs), to produce a set of solutions with the best trade-offs among different objectives, related to PLA extensibility, cohesion and coupling. However, despite the good MOA4PLA results, the approach does not address another task of the PLA design mentioned above: the application of design patterns.

We found in the literature some works that successfully use design patterns with search-based design approaches [36], which served as evidence and motivation for the idea of this work. Cinnéide and Nixon [3] used design patterns in a semi-automatic refactoring approach, whereas Räihä et al. [37,38] used design patterns in the synthesis of software architectures with genetic algorithms (GAs). Although these works can be applied to the SPL context, they do not take into account PLA characteristics, such as how to comply design patterns with variabilities, variation points and variants. Some other works that encompass SPLs, such as [27,46], do not use search-based techniques. In addition to these, most works are specific to certain domains and generally use manual or semi-automatic approaches.

This lack of work addressing both subjects, design patterns applied to PLA and search-based design, maybe is due to two main reasons. The first reason is that both subjects are current research topics, not completely explored yet. The second one is that there are some challenges to overcome for the application of design patterns in the SPL context. The architect must know the design patterns and their main characteristics, as well as he/she must recognize and determine domain-specific patterns based on SPL features [34]. Coplien [8] stated that design patterns are not meant to be executed by computers, but rather to be used by engineers with perception, taste, experience and aesthetics sense. The idea is to capture and encapsulate these virtues into algorithms allowing automatic application of design patterns in architectures.

With this objective in mind, we introduced in a preliminary work [19] a metamodel to represent scopes that are suitable to receive the Strategy and Bridge patterns [14], and a mutation operator ["Pattern-Driven Mutation Operator" (PDMO)] used with MOA4PLA. In this context, a scope is a set of class diagram elements of the PLA, such as classes, interfaces, methods, variabilities and others. A suitable scope is a set of elements that can receive the application of a design pattern, i.e., the design pattern "fits" in the scope and can be suitably applied to these elements in order to improve the design. The proposed operator changes the PLA and helps to improve the design, considering cohesion, coupling and extensibility. The promising results, obtained in a preliminary evaluation with PDMO, serve as motivation to extend our previous work to allow the application of other patterns with MOA4PLA. In this paper, we describe the research results obtained with this extension.

The main contribution of this work to the application of design patterns in the search-based design of PLA is the addition of the *Mediator* design pattern to the proposed approach. To enable this, some secondary contributions are presented. We introduce models to represent the suitable scopes and detail the verification and application methods for such pattern. Because the suite of design patterns is different from our preliminary work, we detail in this paper a set of experiments and we present new findings regarding the addition of this new pattern. The overall results are satisfactory, and *Mediator* instances are found in the generated architectures with coherent and decoupled structures. Furthermore, we identify how the *Mediator* pattern had its elements abstracted by the Bridge and Strategy patterns.

Additionally, we can identify the following secondary contributions of this work: (i) an updated description of the results of the feasibility analysis conducted toward the GoF design patterns and (ii) an updated mutation operator that includes the complete description of the methods to automatically identify suitable scopes and apply design patterns. These updates were performed to increase the operator performance and to include *Mediator* in the approach.

The rest of this paper is organized as follows: Sect. 2 gives a brief background on multi-objective optimization. Section 3 reviews the search-based approach MOA4PLA [6]. Section 4 presents results of the conducted feasibility analysis. Section 5 describes the concept and metamodel to define suitable scopes for design patterns. Section 6 introduces the Pattern-Driven Mutation Operator. Section 7 presents the description of the empirical evaluation using the proposed operator. Section 8 shows and analyzes the experimental results. Section 9 contains related work. Finally, Sect. 10 concludes this paper and discusses future work.

2 Multi-objective optimization

Multi-objective problems are impacted by many conflicting factors, objectives to be optimized. If an objective is optimized, then the other ones will be probably degraded [4]. For instance, when buying a car, the buyer may pay more for a faster car, but if he/she wants a cheaper car, then probably he/she will lose some power in exchange of cost. The idea is to find solutions with a certain compromise between these objectives; thus, several non-dominated solutions may exist. A solution x is said to dominate another solution y ($x \prec y$) if it is better or equal to y in all objectives and is better in at least one objective. If this is not true, then x and y are non-dominated.

All the non-dominated solutions of a problem compose the true Pareto front, i.e., there are no solutions in the search space that are better than the ones in this front. However, these solutions are usually very hard to find, and the multi-objective algorithms can only find an approximation for this front (PF_{known}) [4].

There are several metaheuristics that work with multi-objective optimization [15]. Among them the multi-objective evolutionary algorithms (MOEAs) stand out as one of the most used [4], specially in the search-based software engineering (SBSE) literature [21]. Evolutionary algorithms are based on the theory of evolution [9], where the fittest individuals (solutions) survive and generate more offspring, thus spreading their genes (decision variables of a solution) to more new individuals. The idea behind these algorithms is to evolve a set of solutions through a number of generations by means of crossover and mutation of their chromosomes [4].

In this paper we use the *Non-dominated Sorting Genetic Algorithm II* (NSGA-II) [10]. NSGA-II is a strong elitist algorithm that in each generation creates a new set of solutions (the offspring) based on their parents, joins the offspring with the parents and sorts them according to their fitness. Therefore, at each generation, the solutions that are non-dominated when compared to the other ones survive to the next generation. When there are several non-dominated solutions and the next generation cannot receive all of them, the solutions with the greatest crowding distance (more scattered in the objective space) are selected to survive. Hence, this algorithm favors both the solutions that have the best fitness values (convergence) and the solutions that are more different (diversity). At the end of its execution, the algorithm returns the best non-dominated solutions so that the user can select the one that best fulfills his/her needs.

In software engineering, the fitness functions are related to software metrics [20]. In software testing, a good solution can be a set of test cases that maximizes the branch coverage of a software, while also generating little computational cost. For software design problems, a solution can be an architecture that maximizes some software principles, such as cohesion and coupling. In addition, these solutions can be represented in several ways and be hard to optimize. Therefore, MOEAs are efficient options for this kind of optimization.

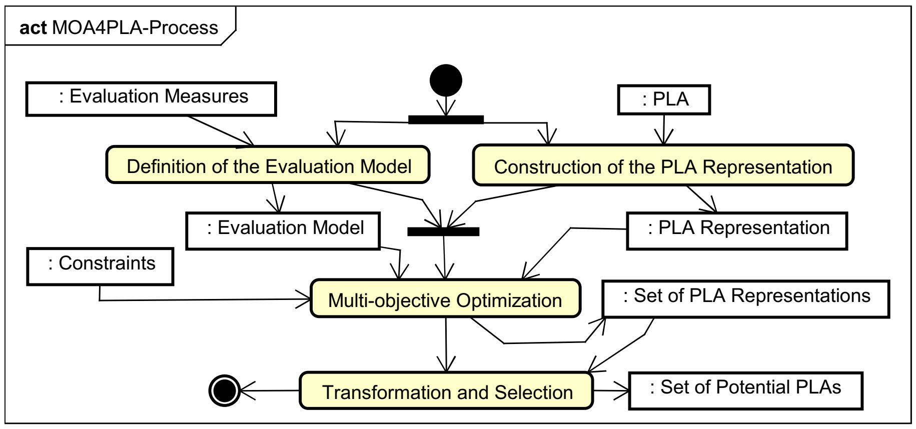
3 Software product line architecture optimization

We can observe that the optimization of a PLA is a multi-objective problem, and to deal with this difficult task the approach MOA4PLA [6] was proposed. MOA4PLA applies MOEAs to optimize PLA designs, using software metrics as fitness functions. At the end of its execution, MOA4PLA generates PLA designs with the best trade-off between the selected metrics. MOA4PLA activities are shown in Fig. 1 and detailed next.

One of the first activities is the *Construction of the PLA Representation*. This activity has as input the PLA, modeled as a class diagram. Hence, the goal is to optimize an existing PLA, but without changing its external behavior. The class diagram must be modeled with all the elements of the PLA, including the features, variabilities, variants and variation points using *SMartyProfile* [26], a variability management profile for UML-based PLAs. From this input, a representation is generated according to the metamodel presented in Fig. 2 [5].

The metamodel has all conventional elements of a class diagram (class, interface, method, attributes, etc.) and also SPL-specific elements (variability, variation point, variant, etc.). Each element is associated with the SPL features that it realizes. A variable element may be associated with a vari-

Fig. 1 MOA4PLA process


act MOA4PLA-Process
: Evaluation Measures
: PLA
Definition of the Evaluation Model
Construction of the PLA Representation
: Evaluation Model
: PLA Representation
: Constraints
Multi-objective Optimization
: Set of PLA Representations
Transformation and Selection
: Set of Potential PLAs

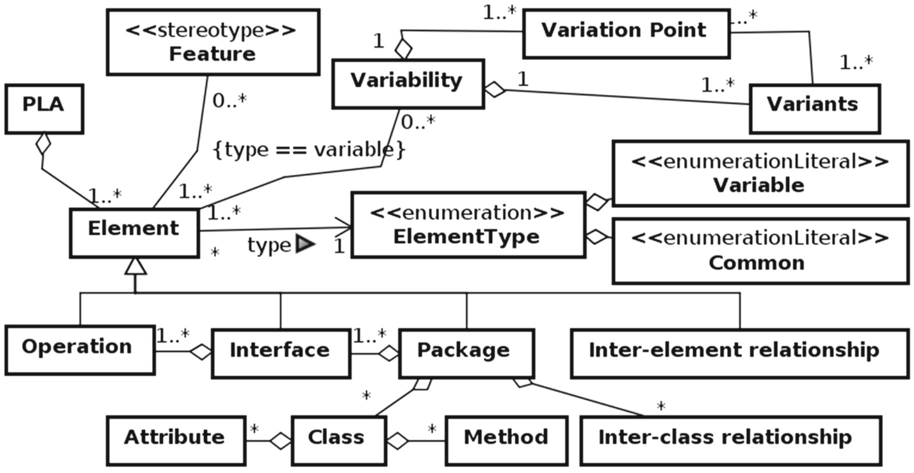


Fig. 2 PLA representation metamodel (extracted from [5])

ability that has a variation point and variants. As mentioned before, a feature is an important functionality that might be common or variable for each product. Thus, a feature may not be associated with any variability (e.g., mandatory feature for all products). Conversely, a feature can have several variabilities to represent all its possible variations.

It is important to emphasize the separation of the UML class diagram and the metamodel of Fig. 2. The UML class diagram is a graphic representation (phenotype) of the architecture where the architect graphically depicts each element. In this graphical representation, the architect is aided by the SMartyProfile notation for designing SPL elements. On the other hand, the instantiation of the metamodel is an object-oriented representation (genotype/chromosome) where each element is an object loaded in memory and can be easily manipulated by the algorithms. In other words, the metamodel defines a general object-oriented structure for instantiating the PLA in memory, which in turn is directly manipulated by the MOEAs. This is in fact the actual chromosome representation (encoding) of the solution for our problem, in contrast to other kinds of problems where the representation is usually an array of integers, doubles or bits [4]. At the end of the MOA4PLA execution, this metamodel instantiations loaded in memory (genotype) are converted back to a corresponding graphical representation (class diagram/phenotype) (Table 1).

In the activity *Definition of the Evaluation Model*, the architect selects the metrics to be used in the optimization process in order to fulfill his/her purposes. These metrics are used as fitness functions to guide the multi-objective optimization. In this paper we used two predefined fitness functions presented in Table 2 [6]: (i) CM (Conventional Metrics) [43]—this function aggregates the metrics which aim at evaluating some basic design principles/attributes such as cohesion, coupling and size of architectural elements and (ii) FM (Feature-driven Metrics)—related to the Feature-driven Metrics [40] (specific for SPL), this function is an aggregation of metrics that measure the attributes of feature scattering, feature interaction and feature-based cohesion, measuring the architectural feature modularization. The metrics used to compose both fitness functions are briefly presented in Table 1.

In the next activity, *Multi-objective Optimization*, the PLA representation and the evaluation model obtained in the previous activities are used to optimize the PLA design using a multi-objective algorithm selected by the architect. In this activity the architect may inform if there is any constraint, such as discarding solutions containing interfaces without operations or relationships. During the optimization process the search operators are used to generate PLA alternatives.

In [6] the authors implemented the following operators, which we call here *PLA Operators*: (i) Move Method—moves a method from one class to another and creates a relationship between these classes; (ii) Move Attribute—moves an attribute from one class to another and creates a relationship between these classes; (iii) Add Class—creates a new class, moves a method of an existing class to it and creates a relationship between these classes; (iv) Move Operation—moves an operation from one interface to another and makes the implementors of the former to implement the latter; (v) Add Package—creates a new package and an interface inside it and moves an operation to this interface; and (vi) Feature-driven Operator—aims at modularizing a random scattered feature into a new modularization package, that is, it moves the elements associated with the feature to this new package. Moreover, all the model modifications ensure that the structure of the PLA remains equivalent to the one given as input, i.e., the operators do not change the external behavior/meaning of the structure. This is done by adding some minor operations to, for instance, add relationships between classes that exchanged methods so that they can still use their previously owned methods. After the PLA alternatives are generated by those search operators, they are evaluated by the fitness functions of the evaluation model.

Table 1 Metrics suites available in MOA4PLA for the definition of fitness functions (table extracted from [6])

Attribute	Metric	Definition
Conventional Metrics (CM) suite [43]		
Cohesion	Relational Cohesion (H)	Average number of internal relationships per class in a component
Coupling	Dependency of Packages (DepPack)	Number of packages on which classes and interfaces of this component depend
	ClassDependencyIn (CDepIn)	Number of elements that depend on this class
	ClassDependencyOut (CDepOut)	Number of elements on which this class depends
	DependencyIn (DepIn)	Number of UML dependencies where the package is the supplier
	DependencyOut (DepOut)	Number of UML dependencies where the package is the client
Size	Number of Operations by Interface (NumOps)	Number of operations in the interface
Feature-driven Metrics (FM) suite [40]		
Feature scattering	Feature Diffusion over Architectural Components (CDAC)	Number of architectural components which contribute to the realization of a certain feature
	Feature Diffusion over Architectural Interfaces (CDAI)	Number of interfaces in the system architecture which contribute to the realization of a certain feature
	Feature Diffusion over Architectural Operations (CDAO)	Number of operations in the system architecture which contribute to the realization of a certain feature
Feature interaction	Component-level Interlacing Between Features (CIBC)	Number of features with which the assessed feature shares at least a component
	Interface-level Interlacing Between Features (IIBC)	Number of features with which the assessed feature shares at least an interface
	Operation-level Overlapping Between Features (OOBC)	Number of features with which the assessed feature shares at least an operation
Feature-based cohesion	Lack of Feature-based Cohesion (LCC)	Number of features addressed by the assessed component

Table 2 CM and FM calculation [6]

$FM(pla) =$	$\sum_{i=1}^{p} LCC + \sum_{i=1}^{f} CDAC + \sum_{i=1}^{f} CDAI + \sum_{i=1}^{f} CDAO + \sum_{i=1}^{f} CIBC + \sum_{i=1}^{f} IIBC + \sum_{i=1}^{f} OOBC$
$CM(pla) =$	$\sum_{i=1}^{p} DepIn + \sum_{i=1}^{p} DepOut + \sum_{i=1}^{cl} CDepIn + \sum_{i=1}^{cl} CDepOut + \frac{\sum_{i=1}^{p} DepPack}{p} + \frac{\sum_{i=1}^{itf} NumOps}{itf} + \frac{1}{\sum_{i=1}^{p} H}$
where	p is the number of packages, itf is the number of interfaces, cl is the number of classes, and f is the number of features of a design pla

The output is a set containing the best PLA representations (solutions), i.e., solutions with the best trade-off between the objectives (fitness functions).

Finally, in the last activity, *Transformation and Selection*, the solutions are converted to UML class diagrams in order to be readable by the architect. Then, a set of potential PLA designs with the best trade-off between the objectives is given as output. After this, the architect can select the one that better fits his/her organizational goals and use it as the architecture of the SPL.

Experiments reported in the literature [6,31] show that MOA4PLA contributes to improving the original PLA, considering the objectives CM and FM. Works on search-based design of traditional architectures also show improvements with the application of design patterns [3,36,38,45]. However, MOA4PLA does not offer a support to the application of design patterns. Both facts serve as motivation to the present work that has as goal the application of such patterns in the MOA4PLA context, in order to obtain better results. To this end, it is necessary first to determine which design patterns can be applied with the MOA4PLA representation. This is the focus of a feasibility analysis, described in the next section.

4 Feasibility analysis

The feasibility analysis was conducted by the first author considering the main characteristics of MOA4PLA and the requirements to automatically apply a pattern. First of all, the approach is focused on class diagrams. Then the GoF patterns were chosen, because they are easily applied to class diagrams by having well-defined and generic structures for object-oriented architectures [14]. Moreover, these patterns appear to be the most well-known design patterns and the most commonly used in the search-based literature [3,36,38, 45]. All GoF patterns were evaluated alone, one after another.

Secondly, the design patterns' application needs to be automatic for a pattern to be considered feasible, since MOA4PLA is fully automated. Our main concern here is that, if we let the algorithm select random elements and apply a design pattern considering these elements, this would probably result in an unreasonable and indiscriminate application. Design patterns are not solutions that can be used in any scope of the architecture and should not be applied just for the sake of using design patterns [8,14]. Rather, a design pattern is a well-defined solution used for a common problem and applied in architectural contexts/structures containing elements that can accommodate their application. These structures in this work are called "suitable scopes" and are basically a group of classes, interfaces, methods, attributes and relationships that form a suitable place for a design pattern application. In order to prevent the indiscriminate application of design patterns and potentially the introduction of design anomalies, in this work a design pattern is only applied in suitable scopes.

Considering both characteristics, we can summarize the main factors that guided our analysis as follows. More details can be found in [18]:

– Structure of the design pattern: analyzed to determine possible scopes for the pattern application, considering the PLA representation given by the MOA4PLA metamodel;
– Consequences: we analyzed the main consequences resulting from the pattern application, for instance impact on cohesion and coupling;
– Applicability in PLAs: the pattern may have a specific application for conventional software architectures, but not necessarily for PLAs;
– Flexibility and implementation aspects: different implementation strategies or other aspects may prevent the automated application of design patterns. For instance, a design pattern may demand behavioral information (about the scope in which it will be applied) that is not representable in class diagrams.

We consider a pattern as feasible if it is possible to automatically identify a suitable scope for its application and the pattern benefits the PLAs in terms of coupling [44], cohesion [44], PLA extensibility [33] and Feature-driven Metrics [40]. In some cases, a pattern, even with an identifiable suitable scope, is considered infeasible if its application does not bring any benefit for the architecture considering the impact in some quality measures used in the evolution process of MOA4PLA. In other cases, some design patterns that have suitable scopes for their application in conventional architectures do not have specific application scopes for PLA. However, the pattern that does not have a particular PLA context to be applied is not necessarily infeasible because it can still be applied to a PLA in scopes without SPL elements.

As a result of the analysis conducted, we concluded that all creational patterns are infeasible. This is because they are appropriate to create objects and are applied to a low range of scopes. It is also difficult to automatically identify suitable scopes for behavioral patterns using only class diagrams. Five of them could be feasible if interaction or other behavioral diagrams were considered (*template method*, *chain of responsibility*, *observer*, *command* and *visitor*). Structural patterns are more suitable in the MOA4PLA context, since they are more compatible with the architecture representation used and have a structure that impacts on coupling and cohesion metrics.

Memento is an example of infeasible pattern. To identify its scope, it is necessary to automatically identify which types of objects can and should have their states stored for further use. The proper diagram for this type of identification is the state machine diagram. In addition to this, its application without such behavioral information would not bring significant influence to the architecture evolution process by means of improvement on the considered metric values, since it has a very specific purpose of solving a state-storing problem. At the end, we have found four feasible patterns: *Bridge*, *Strategy*, *Facade* and *Mediator*. Suitable PLA scopes could be identified only for *Bridge* and *Strategy*. A brief description of the feasible patterns are presented next.

– *Strategy* It aims at making a specific algorithm to vary independently of the clients that use it. It defines a *strategy* interface or abstract class for an algorithm family, in order to make these algorithms interchangeable [14]. In PLAs, the *Strategy* pattern can be used to allow an easy interchange of variants in a variation point.
– *Bridge* This pattern is used to detach the *abstraction* from its *implementation*. An *implementation* element determines what functionality must be executed, whereas an *abstraction* element determines how it must be executed. By using it the architect can vary both independently without using class inheritance, and consequently decreasing the coupling level between the elements [14]. In PLAs, the *Bridge* pattern can be used to extract implementation interfaces of features from the variants of a variation point and to allow a dynamic execution of these features.
– *Mediator* It defines an intermediary class (*mediator* class) in order to encapsulate how a set of elements (*colleagues*) communicate. It promotes a weak coupling between *colleagues* that have strong or complex relationships [14].

– *Facade* It defines a unified interface (*facade* class) for a set of interfaces of a subsystem. It makes the subsystem easier to use by encapsulating its main operations in a higher-layer interface [14].

5 Defining suitable scopes: PS and PS-PLA concepts

To define a mutation operator that is capable of successfully applying design patterns in the context of MOA4PLA, some challenges must be overcome. The following requirements must be ensured for a consistent design pattern application:

1. The pattern is applied into a suitable scope of the architecture;
2. The applied pattern is coherent and does not bring any anomaly to the architecture;
3. The pattern is effectively applied as a mutation in the evolutionary process; and
4. The identification of the suitable scopes and the application of the design patterns are totally automated, i.e., the application does not require user participation.

In order to satisfy these requirements, it is first necessary to provide an easy way to visualize and deal with the suitable scopes for each pattern during the mutation process. To this end, we define a generic metamodel (Fig. 3) that can structurally represent suitable scopes and the other elements related to them. This metamodel complements the MOA4PLA metamodel (Fig. 2).

If a scope is suitable for the application of a design pattern, it is called "Pattern application Scope" (PS). The notation "PS<Pattern Name>" is used to designate a scope for a specific design pattern, for example PS<Strategy>. It is important to highlight that a scope may be a PS for more than one design pattern. In this case, any of the feasible patterns for this scope can be applied.

In addition to the PS specification, there is another category of PS, specific for SPL scopes: "Pattern application Scope in Product Line Architecture" (PS-PLA). It is named in the same way as a PS: "PS-PLA<Pattern Name>".

A PS/PS-PLA is a scope composed by at least one architectural element, which in turn may be present in multiple PSs/PSs-PLA. In fact, for a scope to be considered a PS to a particular design pattern, it needs to meet all PS requirements that this particular design pattern requires. In addition, for a scope to be considered a PS-PLA, besides meeting all PS requirements, it needs to meet also the PS-PLA requirements of the design pattern. These requirements were extracted in the feasibility analysis and are incorporated into verification methods called "verify" (presented in Sect. 6) used by the proposed mutation operator. Moreover, when a pattern is applied using its "apply" method, it may influence some software metrics that are used by the evolutionary algorithms to evaluate the fitness of the achieved solutions.

Regarding the relation between PS and PS-PLA, next we present some points about the application of a given design pattern X:

1. A PS-PLA<X> is obligatorily a PS<X>. If by any reason a scope is not a PS<X>, it cannot be in any circumstance a PS-PLA<X>;
2. A PS<X> is not necessarily a PS-PLA<X>. If by any reason a scope is not a PS-PLA<X>, it can still be a PS<X>;
3. A given pattern X can be applied to any PS<X>, regardless of the type of the architecture of the scope (conventional or PLA);

Each feasible pattern has its PS and PS-PLA represented by an instantiation of the metamodel depicted in Fig. 3. In Fig. 3 the elements "PS Requirement" and "PS-PLA Requirements" are abstractly depicted as a single element, but they are actually composed by several elements that define what a scope must have to be suitable. We present next the PS and PS-PLA representation of all feasible patterns, except *Facade*. Even though we considered *Facade* feasible, we did not implement this pattern in this work due to limitations of the framework used to create the diagrams (Papyrus). For instance, *Facade* would require a subsystem in the diagram, but Papyrus does not have the option to include a subsystem as defined in the UML manual [39] (using the «subsystem» stereotype). Another reason to let *Facade* out of this work is

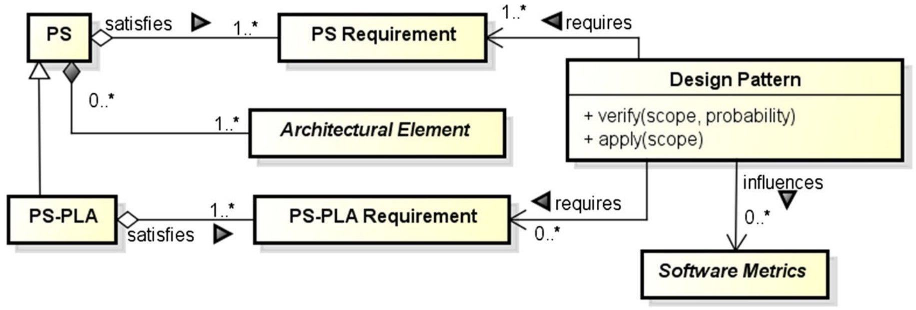


Fig. 3 Metamodel representing PS and PS-PLA

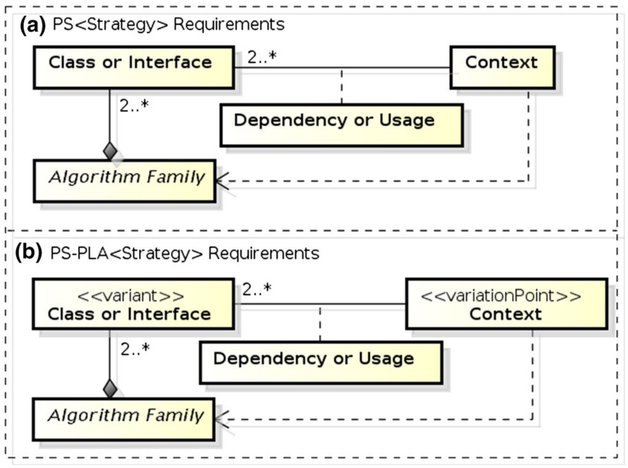

(a) PS<Strategy> Requirements
Class or Interface
2..*
Context
2..*
Dependency or Usage
Algorithm Family
(b) PS-PLA<Strategy> Requirements
<<variant>> Class or Interface
2..*
<<variationPoint>> Context
2..*
Dependency or Usage
Algorithm Family

Fig. 4 Metamodel representing PS<Strategy> and PS-PLA<Strategy>

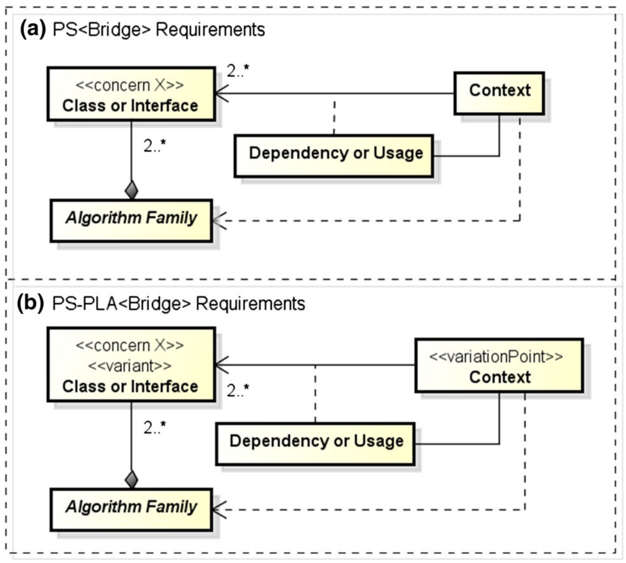

(a) PS<Bridge> Requirements
<<concern X>> Class or Interface
2..*
Context
2..*
Dependency or Usage
Algorithm Family
(b) PS-PLA<Bridge> Requirements
<<concern X>> <<variant>> Class or Interface
<<variationPoint>> Context
2..*
2..*
Dependency or Usage
Algorithm Family

Fig. 5 Metamodel representing PS<Bridge> and PS-PLA<Bridge>

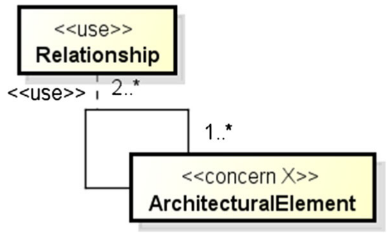


Fig. 6 Metamodel representing PS<Mediator>

that we do not have any PLA class diagram with subsystems in it available for experimentation, so we would not be able to evaluate such a pattern application. It should be addressed in future work by using other frameworks and SPLs.

Figure 4 presents the representations of PS<Strategy> and PS-PLA<Strategy>.

The architectural elements encompassed by PS<Strategy> and PS-PLA<Strategy> are a ‘Context’, which uses at least one ‘Class or Interface’. These classes/interfaces should be part of an ‘Algorithm Family’, so that *Strategy* can abstract such a family from the ‘Context’ point of view. For PS-PLA<Strategy>, ‘Context’ must be a variation point and the classes/interfaces from the ‘Algorithm Family’ must be variants. By abstracting a PS-PLA<Strategy> with the application of a pattern, we can decouple the variation point from its variants.

Figure 5 presents the representations of PS<Bridge> and PS-PLA<Bridge>.

The requirements of PS<Bridge> and PS-PLA<Bridge> are very similar to the requirements of PS<Strategy> and PS-PLA<Strategy>, except that *Bridge* requires at least one common concern assigned to the classes and interfaces of the algorithm family. It is important to note that in SPL engineering a “concern” can be considered a feature. Thus, an element is associated with a concern when it fully or partially realizes a feature (SPL) or a functionality (non-SPL software). In this work we identify concerns in a class diagram by using stereotypes in each associated element. An element is said to be associated with a concern if it is annotated with the respective stereotype, such as «concern X» in the example of the figure. For classes, interfaces and packages, they are also said to be associated with the concerns of their inner elements, such as methods, attributes and contained classes/interfaces (for packages). These stereotypes must be set by the user in the input PLA. The concept of concerns is used by the *Bridge* and *Mediator* methods in their scope identification and in their application. By identifying a common concern, we can identify a common functionality to abstract using *Bridge*.

Figure 6 presents the representations of PS<Mediator>. Even though *Mediator* does not have PS-PLA, the *Mediator* pattern can still be used in PLAs.

The elements encompassed by PS<Mediator> are at least one “ArchitecturalElement” (class/interface) with at least two usage relationships (“Relationship” element with a «use» stereotype). Another requirement here is that the classes/interfaces represented by “ArchitecturalElement” must have at least one concern in common («concern X» stereotype) and at least two usages between such elements.

By instantiating the PS and PS-PLA metamodels, we are able to extract information, for example, about relationship rules (such as multiplicity), element roles in the design pattern structure and type of elements to receive the mutation. This aided the development of methods for identification of suitable scopes and application of the patterns, used by the mutation operator and presented in next section.

6 The Pattern-Driven Mutation Operator

Algorithm 1 presents the *Pattern-Driven Mutation Operator* (PDMO), first introduced in [19]. It has as input the architecture to be mutated (A) and a mutation probability ($\rho_{mutation}$). In the first step, a design pattern DP is randomly selected from the set of feasible patterns (line 2). After this, the mutation operator obtains a scope and stores it in S by randomly selecting a set of architectural elements (classes, interfaces, packages and their inner elements) from A (line 3).

Algorithm 1 Pseudocode for the mutation operator

```
Ensure: $A$ - Architecture to be mutated; $\rho_{mutation}$ - Mutation probability.
  $DP \leftarrow$ randomly select a feasible design pattern;
  $S \leftarrow$ randomly select a set of architectural elements from $A$;
  if $DP.verify(S, \rho_{pla})$ and $\rho_{mutation}$ is achieved then
    $DP.apply(S)$;
  end if
  return $A$
```

The verification method *verify* of DP checks if the scope S is a PS/PS-PLA for the design pattern DP (line 4). The parameter ρ_{pla} is a value used to determine if the verification method of DP will consider the PS or the PS-PLA requirements. If ρ_{pla} is not achieved, then the PS requirements are used; otherwise, the PS-PLA requirements are used. This value is randomized every verification by default, or it can be predefined by the user. Either way, if the verification method returns *true* and the mutation probability $\rho_{mutation}$ is achieved, then the design pattern DP is applied to S (line 5) using its *apply* method. At the end, the architecture A is returned (line 7), whether it was mutated or not.

Each feasible design pattern has its own verification and application methods. These methods are the key components of the proposed mutation operator, since they are the ones that verify the suitability of a scope and actually apply the mutation. We present, respectively, the verification and application methods of *Strategy*, *Bridge* and *Mediator* in the next subsections.

6.1 Verification methods

Each feasible design pattern has a *verify* method for the PS or PS-PLA verification. This method receives a scope S as parameter and does several verifications on its architectural elements to check if S is suitable to receive the application of DP. In other words, the *verify* methods scan the scope S to check if the elements comply with the PS or PS-PLA requirements of DP. Hence, it is divided into two main procedures: the PS verification and the PS-PLA verification. The PS-PLA verification is only used when the ρ_{pla} probability is achieved and if the design pattern can be applied to PS-PLA scopes. In this work only *Mediator* does not have PSs-PLA; therefore, it is always applied to PSs.

The PS<Strategy> verification checks if the S scope has an algorithm family in its structure and if the algorithms from this algorithm family are being used by at least one external element (*context* role). Furthermore, *context* must use these algorithms with a dependency or a usage relationship. In this work an algorithm family is characterized by at least two classes and/or interfaces with: (i) a same suffix in their names; (ii) a same prefix in their names; or (i) at least one method in common, i.e., a method with a same name and return type. We used names and methods as source of similarity identification because of the limitations of class diagrams in denoting behavior. A class diagram is a type of static diagram [39] that mainly displays structural information. Hence, we believe that names and methods are some of the most reliable ways to identify an algorithm family. For example, an algorithm family for sorting algorithm classes (e.g., *BubbleSort*, *QuickSort* and *ShellSort*) can easily be identified by the suffix “Sort” in the name of the classes, or by a common “sort” method. The definitions of what is an algorithm family may vary from one author to another depending mainly on their aesthetic comprehension and design preferences.

The PS<Bridge> verification is similar to the PS<Strategy> verification, with the addition of one rule: at least two algorithms of the algorithm family must have at least one common concern associated with them. By adding this rule we can identify elements with similar functionalities and create an *abstraction* for them. This common concern does not need to be a SPL feature but merely a functionality annotated by the user in the design time; thus, a non-SPL software is also included in this verification as the PS concept suggests.

The PS<Mediator> verification also uses the concern concept. A scope is a PS<Mediator> if it has at least two classes and/or interfaces, with at least two dependencies or usages between them and with at least one concern associated with them. If a scope meets these criteria, then it has a set of *colleagues* with similar functionalities and a potential high dependency among them. If that is the case, then the *Mediator* pattern can be used to mediate their communication. A lower minimum number of relationships between the *colleagues* may cause the addition of unnecessary complexity by the *Mediator* pattern.

Finally, for both PS-PLA<Strategy> and PS-PLA<Bridge> verifications, there must be at least one *context* that is a variation point and the algorithms used by this *context* must be variants.

These criteria do not ensure that a design pattern will only be applied in suitable scopes, since there can be scopes that satisfy all rules and yet would not be considered suitable by some designers. Instead, we designed and implemented these verification methods in order to decrease the number of indiscriminate applications of design patterns throughout the optimization process and consequently to provide a good design quality for the PLAs. If S is evaluated as a PS<*DP*> or a PS-PLA<*DP*>, S can receive the pattern DP. This is done by the *apply* methods, presented next.

6.2 Application methods

The *apply* method is the one that actually mutates the architecture by adding, removing or changing architectural elements. Each application method first verifies if the design pattern is already applied to the scope. If its design pattern is not yet applied in the scope, then a new instance is created. On the other hand, if there is a design pattern instance in the scope, then this instance receives minor adjustments in order to be reused by the operator. For example, if the scope has a *Strategy* interface and the *Strategy* design pattern was selected to be applied to that scope, then the application method will make any new element to implement the existing interface instead of creating an unnecessary interface for the algorithm family. In addition, after a design pattern is applied, all elements that take roles in the pattern structure receive a stereotype annotation for that design pattern. This annotation helps a further identification of applied patterns.

Strategy application method
Figures 7 and 8 present, respectively, an example of a PS-PLA<Strategy> to be mutated and the scope mutated with the *Strategy apply* method. Besides the conventional elements of a UML diagram [39], the diagrams also show some specific SPL elements using the SMarty notation [33]. Briefly, using SMarty the architect describes a variability in an UML comment, indicates a variation point using the «variationPoint» stereotype and depicts several types of variants using other stereotypes.

As shown in Fig. 7 the scope is a PS-PLA<Strategy> because there is a *context* element ("Client") that uses two algorithms ("ClassA" and "ClassB") of an algorithm family. In fact, "Client" is considered a *context* element just because it uses elements of the algorithm family and thus can have its internal behavior changed by changing the concrete algorithm it uses. Furthermore, these are algorithms of a same algorithm family because they have a common prefix ("Class") and a common method ("method"). Also, the *context* element is a variation point (annotated with «variationPoint») and the algorithms are variants (annotated with «variant»). Figure 8 shows the resulting scope.

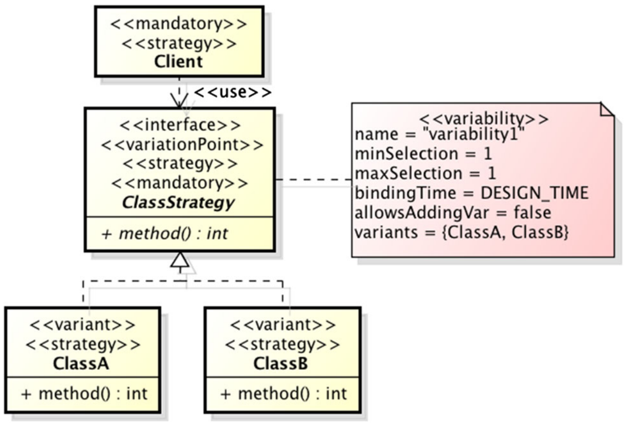


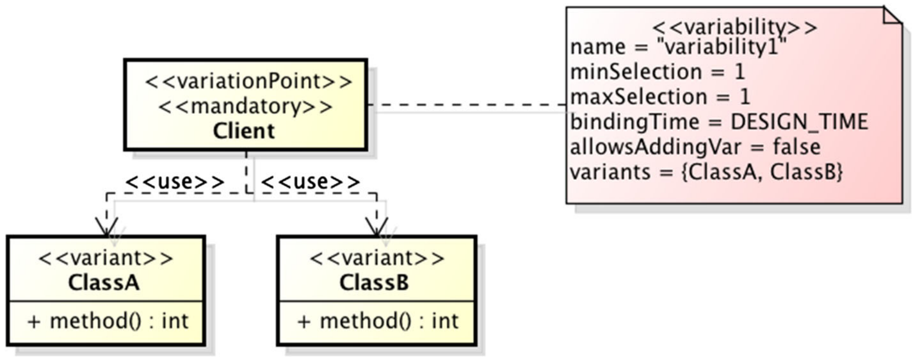
Fig. 8 *Strategy* application example (after mutation)

In order to achieve this result, the following steps are performed by the *Strategy apply* method:

1. Create a *Strategy* interface for the algorithm family;
2. Declare in the *Strategy* interface all methods of all algorithm elements;
3. Make the algorithm elements to implement the interface;
4. Make all *context* and other elements that use the algorithms to use the interface instead of using the algorithms;
5. If the architecture being mutated is a PLA and there is a variability whose variants are all algorithms of the algorithm family, then move the variability to the *Strategy* interface and define it as the variation point of the variability;
6. Annotate the *context* element, *Strategy* interface and all algorithms, with the «strategy» stereotype.

The first step is to create a *Strategy* interface if there is no such interface yet. In the example the "ClassStrategy"

Fig. 7 *Strategy* application example (before mutation)



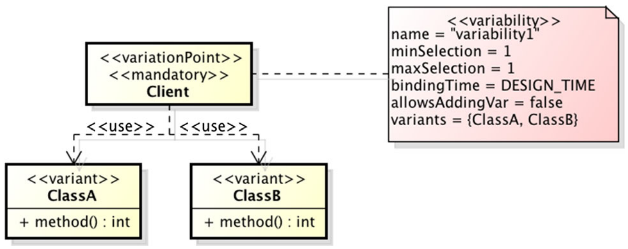


Fig. 9 *Bridge* application example (before mutation)

was created to abstract all algorithms of the algorithm family. After that, all methods of the algorithms are declared in the interface and the algorithms implement the interface. As explained in [14], the *Strategy* interface must declare all methods of all algorithms, even though some of these methods will never be implemented by all of them. In that case, when a class has to declare an unused method from its *Strategy* interface, it does not need to really implement the method, but rather leave the method in blank.

Instead of using each algorithm directly, the *context* element uses the interface after the mutation, thus reducing the coupling between it and the algorithms. Because there is a variability linked to the context element and all algorithms are variants, then this variability must be moved to the interface in order to preserve the direct link between variabilities, variants and variation points. Finally, these elements are all annotated with the *Strategy* design pattern-specific stereotype («strategy»).

Bridge application method

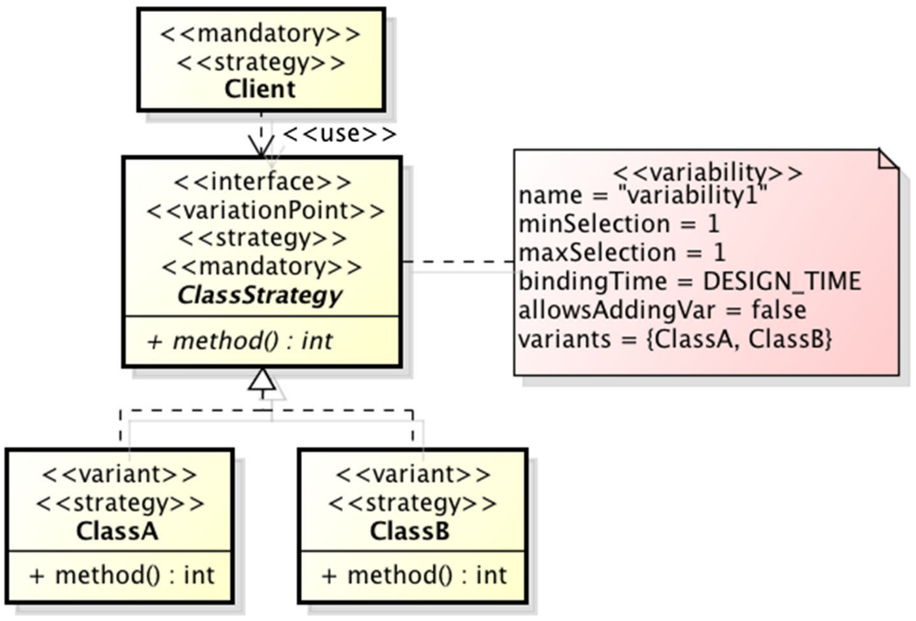
Similarly to the *Strategy* mutation, Figs. 9 and 10 show, respectively, an example before and after the *Bridge* mutation using the *apply* method. Figure 9 shows the PS-PLA<Bridge> to be mutated.

As shown in the diagram, “Client” is the context element that uses both classes “ClassA” and “ClassB” with a usage relationship. These classes are algorithms of an algorithm family, and they are variants, same as in the *Strategy* example. The difference here is the common concern “concern X” given for both algorithms, which enables the characterization of the scope as a PS-PLA<Bridge>. Figure 10 presents the expected result of the *Bridge* application.

To obtain this result, the *Bridge apply* method follows the steps below:

1. Create an abstract class *abstraction* for the elements of the algorithm family and declare in this class all methods of these elements;
2. For each concern associated with at least two algorithms of the algorithm family, create an *implementation* interface declaring all methods (of the algorithms) that are associated with the concern;
3. Make each algorithm to implement the *implementation* interfaces created for its concerns;
4. Make *abstraction* to aggregate all *implementation* interfaces;
5. Create a default concrete class and make it to extend *abstraction*;
6. Make all *context* and other elements that use the algorithms to use *abstraction* instead of using the algorithms;
7. If the architecture being mutated is a PLA and there is a variability whose variants are all algorithms of the algorithm family, move the variability to the *implementation* interfaces and set these interfaces as variation points;

Fig. 10 *Bridge* application example (after mutation)



8. Annotate with «bridge» all algorithms, all *implementation* interfaces, the *abstraction* class and the default concrete class.

The first step is to create the "ClassAbstraction" abstract class (*abstraction* role) for the algorithm family. This abstract class must declare all methods of the algorithm family, as a *Strategy* interface does. In addition, for each concern associated with at least two algorithms, then an *implementation* interface is created for this concern and the methods that are associated with this concern must be declared in the interface. In the example there is only one *implementation* interface ("XImplementation"), which was created for "concern X." This interface is implemented by both algorithms, because they share "concern X." Furthermore, "ClassAbstraction" must aggregate all *implementation* interfaces and the concrete class "ClassAbstractionImpl" is created to extend the *abstraction* class. This concrete class must have a default behavior to define how each algorithm is executed. The *context* elements should only use the *abstraction* class, so that they can be decoupled from the concrete classes.

By doing this mutation, the *abstraction* class can vary each concrete object aggregated through the interfaces. This implies in a free and runtime variation of which class realizes each concern, since a concern is abstracted by its own *implementation* interface. For instance, if there was another common concern for both algorithms in the example, there would be another *implementation* interface and the *abstraction* class would aggregate two interfaces. Therefore, a single object of a class that implements both interfaces could be used to realize both concerns, or two objects of different classes could be used to realize one concern each.

Even though some methods are repeated in the *abstraction* class and in their respective *implementation* interface, there is a crucial difference between each declaration. While a given method declared in an *implementation* interface determines what functionality should be implemented by the implementors of that interface, the same method declared in the *abstraction* class determines how this functionality must be used. Therefore, the latter can have a template body in the abstract class and a more specialized behavior for each concrete class that extends *abstraction*.

Mediator application method

For the *Mediator* example, Fig. 11 presents a PS<Mediator> to be mutated. We used in this example a PS instead of a PS-PLA, because *Mediator* does not have PS-PLA scopes in our work.

This scope contains some *colleagues* ("ClassA", "ClassB", "ClassC" and "ClassD") with usage relationships in several directions. In addition, all *colleagues* are associated with a common concern ("concern X"). Therefore, this is a PS<Mediator>. The main purpose of applying the *Mediator* pattern here is to decrease the coupling between the *colleagues* by delegating the communication responsibility to a new class. Figure 12 shows the mutated scope.

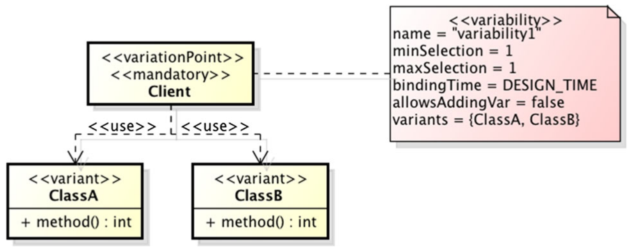


Fig. 11 *Mediator* application example (before mutation)

For the *Mediator* pattern application we used the *Event of Interest* implementation. This implementation defines a class to encapsulate the details of a communication event between *colleagues* and to be used by the *Mediator* class. The *Mediator apply* method performs the main steps as follows:

1. Create the "EventOfInterest" class;
2. For each common concern associated with at least two elements with at least two dependencies and/or usage between them, create:
   (a) A *Mediator* interface;
   (b) A default *Mediator* class;
   (c) A *Colleague* interface;
3. Make each *colleague* to implement the *Colleague* interfaces created for its concerns;
4. Create an action method for the *Mediator* interface and class. This method has as parameter an object of the "EventOfInterest" (identifies the event) class;
5. Add to the *Colleague* interfaces a method to attach and another to detach a *Mediator* object. This method defines which object must be used to intermediate the communication between the colleagues;
6. Make the *Colleague* interfaces to use their respective *Mediator* interface;
7. Make the *Mediator* class to directly use each one of its respective *colleague*;
8. If a *colleague* has only concerns that have *Mediators*, then make this *colleague* to stop being used by any other element that is not a *Mediator* class and make the elements to use the *Mediator* interfaces instead of using the algorithms;
9. If the architecture being mutated is a PLA, then move all variabilities that have only *colleagues* as variants to the respective *Colleague* interface and set this interface as a variation point;
10. Annotate all *colleagues*, the *Mediator* class, the *Mediator* interface and the *Colleague* interface with the «mediator» stereotype.

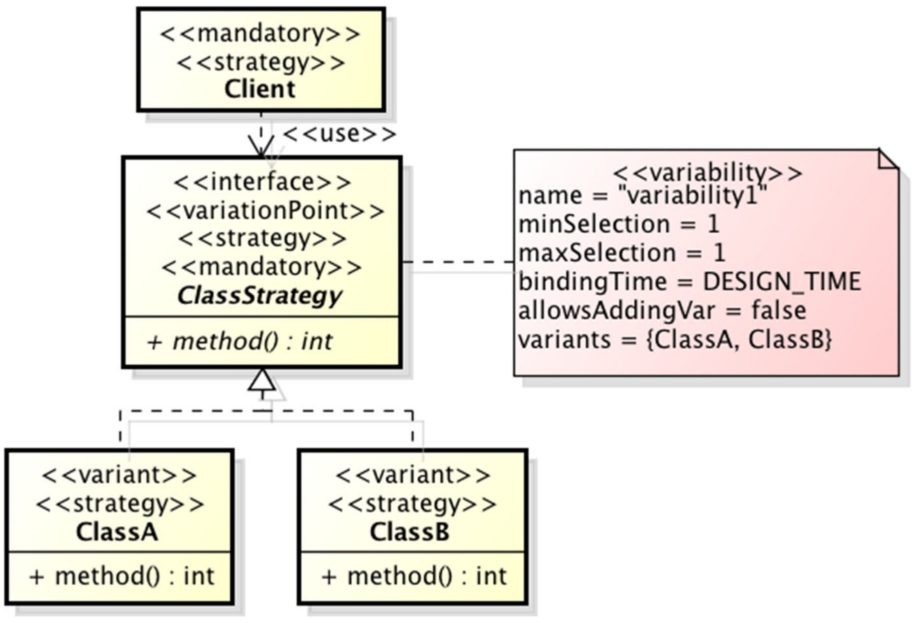


Fig. 12 *Mediator* application example (after mutation)

In the example only one *Mediator* interface ("XMediator"), one *Mediator* class ("XMediatorImpl") and one *Colleague* interface ("XColleague") were created because there is only one concern for the *colleagues*. The *Mediator* interface has a single method named "concernXAction" with a single parameter of the "EventOfInterest" class. This method is responsible for receiving the "EventOfInterest" object that identifies the communication between *colleagues*. Therefore, when a *colleague* needs to send a message to another, it creates the "EventOfInterest" object, fills its properties with useful data and invokes the action method of the "XMediator" interface. Then the *Mediator* object receives the event object, identifies the receiver and proceeds to conclude the interaction. Optionally, the architect may define communication rules in the *Mediator* class, or even create several classes with different communication handling. This allows a runtime changing of communication behavior.

One of the rules of the *Mediator* pattern is that the *Mediator* class must use each *colleague* directly. In addition, as Gamma et al. defined in [14], the *Mediator* interface can be used as a *Facade*. Therefore, instead of using the *colleagues* directly, other elements will now use the *Mediator* interface.

The *Mediator* pattern may seem to bring more coupling to the structure by adding some relationships between the *Mediator* elements and the *colleagues*. However, these relationships introduce an overall weaker coupling between these elements than relationships between *colleagues*. On the other hand, the *Mediator* application may cause the architecture to be less understandable, given the introduced complexity.

Each design pattern is applied to a scope disregarding the existence of other design patterns in that same scope. Therefore, the application of one design pattern can disrupt the structure of other design patterns instances. To evaluate if cases like this really occur and if the proposed operator is beneficial for the search-based optimization of PLAs design, we conducted an empirical evaluation with some SPLs. This empirical evaluation is described in the next section.

7 Empirical evaluation

The empirical evaluation presented in this section is concerned in answering the following general research question: "Does the automatic application of design patterns using PDMO contribute to improve the PLA design?" To answer this question, we conducted quantitative and qualitative analyses. We are also interested in observing Mediator applications, since this is a new design pattern application introduced in this paper.

In the quantitative analysis, we compared the fitness values obtained by the design pattern application with the values of the original PLAs and with the values of the solutions found by MOA4PLA. In these comparisons we evaluated the generated Pareto fronts and used the *hypervolume* indicator [47] and the Euclidean distance to the ideal solution (ED) [23]. The main purpose of these comparisons is to determine the capability of *Pattern-Driven Mutation Operator* to improve the PLAs and the results of MOA4PLA in terms of software metrics and solutions diversity. We chose the hypervolume indicator because we do not know the true Pareto front for the PLAs and because we want to evaluate the results regarding both fitness and diversity. Furthermore, this indicator is Pareto compliant [48] and can be used to differentiate the generated fronts when they overlap. The ED indicator was

Table 3 PLAs details

PLA	Fitness (FM, CM)	Packages	Interfaces	Classes	Variabilities
AGM	(789.0, 6.1)	9	15	31	5
MOS	(567.0, 0.1)	14	15	26	10
MM	(221.0, 0.3)	8	15	10	7
ETC	(742.0, 0.02)	56	30	115	8

used as a tie-breaking technique for identifying the best non-dominated solution of a Pareto front. Finally, the Friedman test was used to determine if there is statistical difference between the results of each set of results.

In the qualitative analysis, we analyzed one architecture among the generated, since a manual analysis of all solutions would be infeasible. The objective is to assess for that solution: (i) its quality; (ii) if the solution is useful for the architect; (iii) if the design patterns were correctly applied in suitable scopes; and (iv) if any inconsistency (anomaly) was observed in the PLA design after the optimization.

7.1 PLAs used in the evaluation

We used four PLAs: (i) AGM (*Arcade Game Maker*) [24] is an SPL created for supporting the learning and experimentation of product lines. This PLA encompasses three arcade games: *Brickles*, *Bowling* and *Pong*; (ii) *Microwave Oven Software* (MOS) [17] is an SPL that offers options for basic to top of the line microwaves. It has some features such as display language selection and predefined recipes for fast cooking; (iii) *Mobile Media* (MM) [7] is an SPL for the media management of mobile devices, supporting music, video and photograph; and (iv) *Electronic Transport Cards* (ETC) [12] is used for the management of electronic transport cards in urban transportation, mainly commercial, having several features such as payment, itinerary and ticket management.

We used two objectives associated with the CM and FM functions, described in Sect. 3. Table 3 presents some details about each PLA, such as the original FM and CM values, number of packages, classes, interfaces and variabilities.

Among all used PLAs, AGM has the greatest value of feature scattering and interaction, while MM has the lowest. MOS has the lowest CM value, and AGM has the greatest. The optimization focuses on minimizing the fitness values of the PLAs.

7.2 Experiment description

To answer the research question we used NSGA-II [10] as MOEA for MOA4PLA and conducted three experiments: (i) *Product Line Architecture Mutation* (PLAM) encompassing all MOA4PLA Operators (called here *PLA Operators*); (ii) *Design Pattern Mutation* (DPM) encompassing only the *Pattern-Driven Mutation Operator*; and (iii) *Product Line Architecture and Design Pattern Mutation* (PLADPM) encompassing all MOA4PLA Operators and the *Pattern-Driven Mutation Operator*. Ultimately, the objective of comparing such different experiments is to asses if the usage of design patterns can contribute to the MOA4PLA approach (PLAM experiment).

If the experiment being executed is PLAM, then a random mutation operator (out of six available ones) is applied in each recombination (if the mutation probability is achieved). For DPM, only the *Pattern-Driven Mutation Operator* is available; thus, it is the only mutation possibility. For PLADPM, a random operator is used (out seven, six MOA4PLA Operators plus the *Pattern-Driven Mutation Operator*). In order to avoid the disruption of design pattern instances and the introduction of design anomalies, we added a restriction to forbid the application of any *PLA Operators* in elements that have roles in a design pattern structure. The identification of a design pattern structure is aided by the stereotypes created by the application methods (Sect. 6.2).

The initial population is composed by one solution with the original PLA design and by other solutions that are derived from the application of random mutation operators in the original PLA design, as done by Colanzi et al. [6]. Thus, each experiment uses its available operators to generate the initial population.

Each experiment had 30 independent runs executed for each problem (in this work a problem is a PLA design). At the end of the runs, each experiment obtained for each PLA a known Pareto front (PF_{known}) composed by all non-dominated solutions found during its 30 runs for that PLA. Because we do not know the real true Pareto fronts of the PLAs, we composed an approximation of the true Pareto front (as recommended in [48]) for each PLA by joining all non-dominated and non-repeated solutions from each PF_{known} found for the PLA. This approximation is called here simply PF_{true}.

7.3 Parameter tuning

In our parameter tuning, we adjusted the following parameters: Population Size and Max Evaluation Number. Each parameter was adjusted for each PLA and each experiment; since it is common, values do differ depending on the problem [2].

According to the experiments of Colanzi et al. [6], 90% is the most adequate value for the mutation probability for the type of problem of this work. This value is the most suitable because if a lower value is used, too many solutions will not be modified throughout the generations and the quantitative results would be spoiled, because the mutation is the only source of design evolution. On the other hand, a greater value would deteriorate the quality of the design and make the

Table 4 Best configurations obtained with the tuning

Experiment	PLA	Population size	Max fitness evaluations
PLAM	AGM	200	30.000
	MOS	–	–
	MM	50	30.000
	ETC	50	30.000
DPM	AGM	50	30.000
	MOS	50	300.000
	MM	50	30.000
	ETC	100	30.000
PLADPM	AGM	200	30.000
	MOS	50	30.000
	MM	50	30.000
	ETC	200	30.000

Row with a dash symbol represents the inability of PLA optimization by the referred experiment

evolutionary process to be like a random one. Therefore, we fixed the mutation probability as 90%.

For the population size, we adjusted the values 50, 100 and 200, which are the most used values in the literature [2]. In this work, the stop criterion is the maximum number of fitness evaluations. According to Arcuri and Fraser [2], this parameter must be adjusted when it is the stop criterion. The first step is to obtain a default value and to generate a value that is one-tenth and another that is ten times this default value. Based on the work of Colanzi et al. [6], the maximum numbers of fitness evaluations for this tuning are: 3000, 30,000 and 300,000.

That way, using the values defined, we created 108 combinations of parameters/PLA/experiment (4 PLAs $\times$ 3 experiments $\times$ 3 population size values $\times$ 3 max fitness evaluations values) and we executed each combination 30 times. We then chose the best configuration for each experiment for each PLA regarding the *hypervolume* indicator [47] and the Friedman test with 95% significance [11]. For each experiment and each PLA, we chose the configuration that was the fastest to be executed and that yielded statistically better or equivalent results to the best *hypervolume*. By doing this we ensured the usage of configurations that generated good results and that are fast to be executed. Table 4 presents the best configurations obtained.

The first column of Table 4 shows the experiments, followed by each available PLA in the second column. The columns three and four show, respectively, the population size and the maximum number of fitness evaluations obtained in the tuning.

There is a peculiarity regarding the MOS PLA: the PLAM experiment could not optimize its design. In this case, the original PLA dominated all the solutions generated by the PLAM experiment for all configurations. Therefore, in the comparisons regarding the MOS PLA, the *hypervolume* value of the PLAM experiment is the *hypervolume* of the original PLA.

7.4 Threats to validity

We identified some threats that can invalidate the obtained results:

Size of the problems The PLAs used in this study have a relatively small number of elements. However, the majority of PLAs do not have class diagrams as main model for their representation, or their class diagrams could not have been provided by their developers due to copyright reasons. This hindered the search for other problems. The usage of bigger problems could provide a more significant sample of real-world problems and present different experimental results.

Initial population The population initialization is a factor that must be evaluated. A well-designed initialization can lead to better results and a greater solution diversity. In this work the initial population is created using only the mutation operators available for each experiment. Other strategies could be analyzed, such as giving to all experiments the same initial population. However, this is a subject of discussion for future work with a deeper and careful analysis focused exclusively on it. We adopted the same approach from related work [6], where the population initialization is based on the input PLA. This generates an initial population that is not very different from the original PLA, in order to ensure the SPL functionality. We did this to enable comparison and to ensure that we are evaluating only the performance of each set of operators independently, i.e., not letting the results be modified by external factors such as an initial population that favors one kind of operator over another.

Software metrics and objective functions We observed a certain inability from the metrics and objective functions for capturing some design problems in the architecture. For instance, a *Strategy* interface may declare several methods that are implemented by all its implementors. Despite that being foreseen and justified in the GoF catalog [14], this can lead to a creation of interfaces overloaded with operations and an indiscriminate responsibility assignment for the elements of the algorithm family. Perhaps other software metrics or even the same metrics used in different objective functions could capture this and other design problems, and thus the optimization process could discard/correct the solutions during its execution. The usage of other metrics and the composition of other objective functions are subjects that might be evaluated and treated in future work. Finally, this issue is not exclusive to our work, but rather a recurrent problem in the search-based software engineering field, specially for the design and refactoring tasks [42].

Table 5 Pareto fronts details

PLA	$\boldsymbol{PF_{\text{true}}}$	$\boldsymbol{PF_{\text{known}}}$		
		PLAM	DPM	PLADPM
AGM	53	30 (22)	32 (0)	**46 (32)**
MOS	37	1 (1)	**37 (36)**	26 (3)
MM	46	6 (6)	31 (0)	**45 (42)**
ETC	102	6 (6)	92 **(82)**	**94** (18)

8 Results and discussion

This section presents the results and the quantitative and qualitative analyses performed.

8.1 Quantitative analysis

The quantitative analysis takes into account the fitness values of the solutions of each experiment. First we plotted the obtained fronts, and then we used the *hypervolume* [47] and the Euclidean distance to the ideal solution (ED) [23] indicators to evaluate the quality of these fronts. After this, we present the execution time of each algorithm. Finally, we discuss what we observed with these experiments in a discussion section.

8.1.1 Pareto fronts results

Table 5 presents some details about the obtained Pareto fronts. The second column presents the number of solutions in its PF_{true}. Columns three, four and five show, respectively, the number of solutions in the PF_{known} found by the experiments PLAM, DPM and PLADPM. The number in parentheses shows the quantity of solutions from PF_{known} that are also in PF_{true}. The best values are highlighted in bold.

As listed in Table 5, using the *Pattern-Driven Mutation Operator* (both DPM and PLADPM), it is possible to obtain a greater number of non-dominated solutions. The number of solutions found by DPM and PLADPM is always greater when compared to PLAM. Figure 13a–d presents, respectively, the PF_{known} found by each experiment for AGM, MOS, MM and ETC.

Despite some PF_{known} fronts being close, it is possible to note that in every subfigure of Fig. 13 and in the values of Table 5, the fronts obtained by the PLAM experiment (only the MOA4PLA Operators) are always less diversified. In addition, for the PLAs AGM, MM and ETC the original architecture is dominated by at least one solution with design patterns (experiments DPM and PLADPM). For the MOS PLA, the original design was kept as a non-dominated solution, but it did not prevent the discovery of new non-dominated solutions using design patterns. In this case, the optimization process at least gave to the architect other options equally good (non-dominated). Another interesting point here is that, as mentioned in the last section, PLAM was not able to optimize MOS. However, as listed in Table 5, PLAM obtained one non-dominated solution and this solution is present in PF_{true}. In fact, that specific solution obtained by PLAM is the original PLA itself, which was given as input, added as an initial solution in the initialization process [6] and survived during the whole evolutionary process until it was returned as the only non-dominated solution by that experiment.

8.1.2 Hypervolume results

Since the PF_{known} fronts overlap on some points, we provide another way to analyze the results by using the hypervolume indicator. Table 6 presents the *hypervolume* averages achieved by each experiment from its 30 executions. The best averages are highlighted in bold, and the values with an equal ("=") symbol represent values with statistical equivalence in their *hypervolumes* (all 30, not average) for that problem using the Friedman test with 95% significance [11]. We also normalized the fitness values before executing the hypervolume calculation. Some *hypervolume* values are over 1 because we fixed the reference point as "1.1, 1.1" (worse than the worst normalized values). Figure 14a–d presents the *boxplot* graphs for the obtained *hypervolumes*, respectively, for the PLAs AGM, MOS, MM and ETC. These graphs support the results listed in Table 6.

By analyzing Table 6 and Fig. 14, it is possible to note that the PLAM experiment was better only for AGM, with results that are statistically equivalent to the PLADPM experiment. In the other cases, the use of the *Pattern-Driven Mutation Operator* alone or in combination with the *PLA Operators* obtained the best (or statistically equivalent) results. It is interesting to note that for AGM, PLADPM obtained more solutions in PF_{true} than PLAM, yet PLAM obtained a greater hypervolume average. In reality, PLAM obtained a greater "raw" hypervolume average, but it is statistically equivalent to PLADPM. So, it means that the apparent "inconsistency" happened by chance, and this could have been the other way around: we could have observed a greater number of solutions in PF_{true} for PLAM and had PLADPM with a greater hypervolume average, yet these results would probably be statistically equivalent. Furthermore, the "raw" hypervolume difference can be explained by how close the solutions are scattered in the fronts. Basically, the solutions obtained by PLADPM that are in PF_{true}, even though in greater number, produce less hypervolume contribution by being too close from one another (even though not entirely visible in the plots).

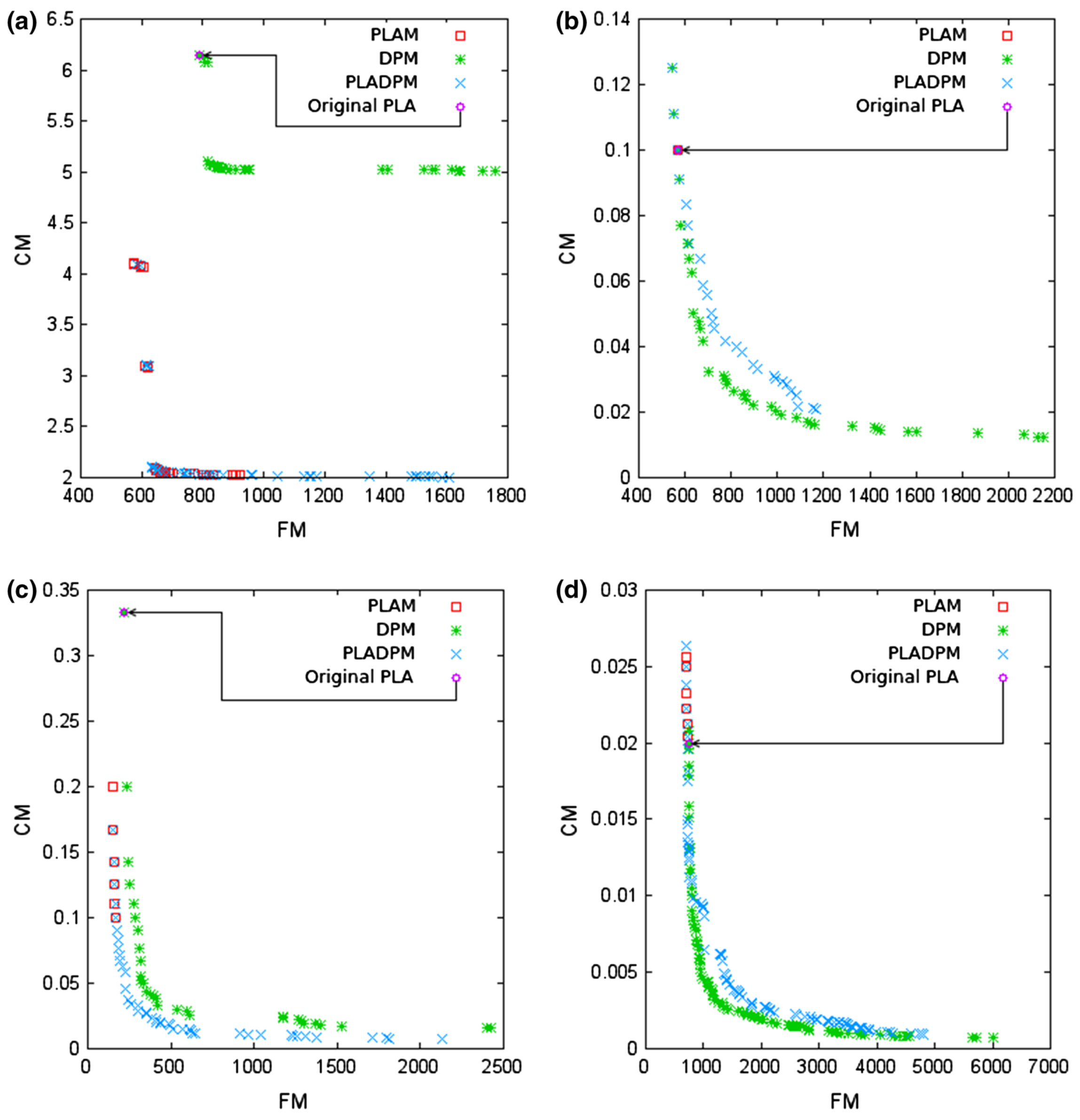


Fig. 13 PF_{known} found by the experiments. **a** AGM, **b** MOS, **c** MM and **d** ETC

Table 6 Hypervolume average obtained by the experiments

PLA	Hypervolume		
	PLAM	DPM	PLADPM
AGM	=**0.9914** (0.1770)	0.3481 (0.0019)	=0.9586 (0.2098)
MOS	0.3647 (0.0)	**1.0876** (0.0172)	0.9518 (0.0552)
MM	0.8392 (0.0522)	1.0939 (0.0148)	**1.1849** **(0.0075)**
ETC	0.4013 (5.76E-5)	**1.1795** (0.0041)	1.1394 (0.0102)

8.1.3 Euclidean distance results

Another indicator that we used is ED. This indicator computes the phenotype distance between two solutions, i.e., the distance between two solutions in the objective space. By calculating the distance between the solutions and an ideal one (the best objective values found for all objectives during the experiment), we can assess how close each solution is from the ideal one. Table 7 shows the solutions with the best and the worst ED values found by each experiment. The first and second columns present, respectively, the PLA and the fitness value of its ideal solution. Columns three and four, five and six, and seven and eight present, respectively, the solutions with the lowest ED (−ED) and greatest ED (+ED) for the experiments PLAM, DPM and PLADPM. Each cell contains the ED value and the fitness value (FM, CM) of the respective solution. The best (lowest) values are highlighted in bold for each PLA.

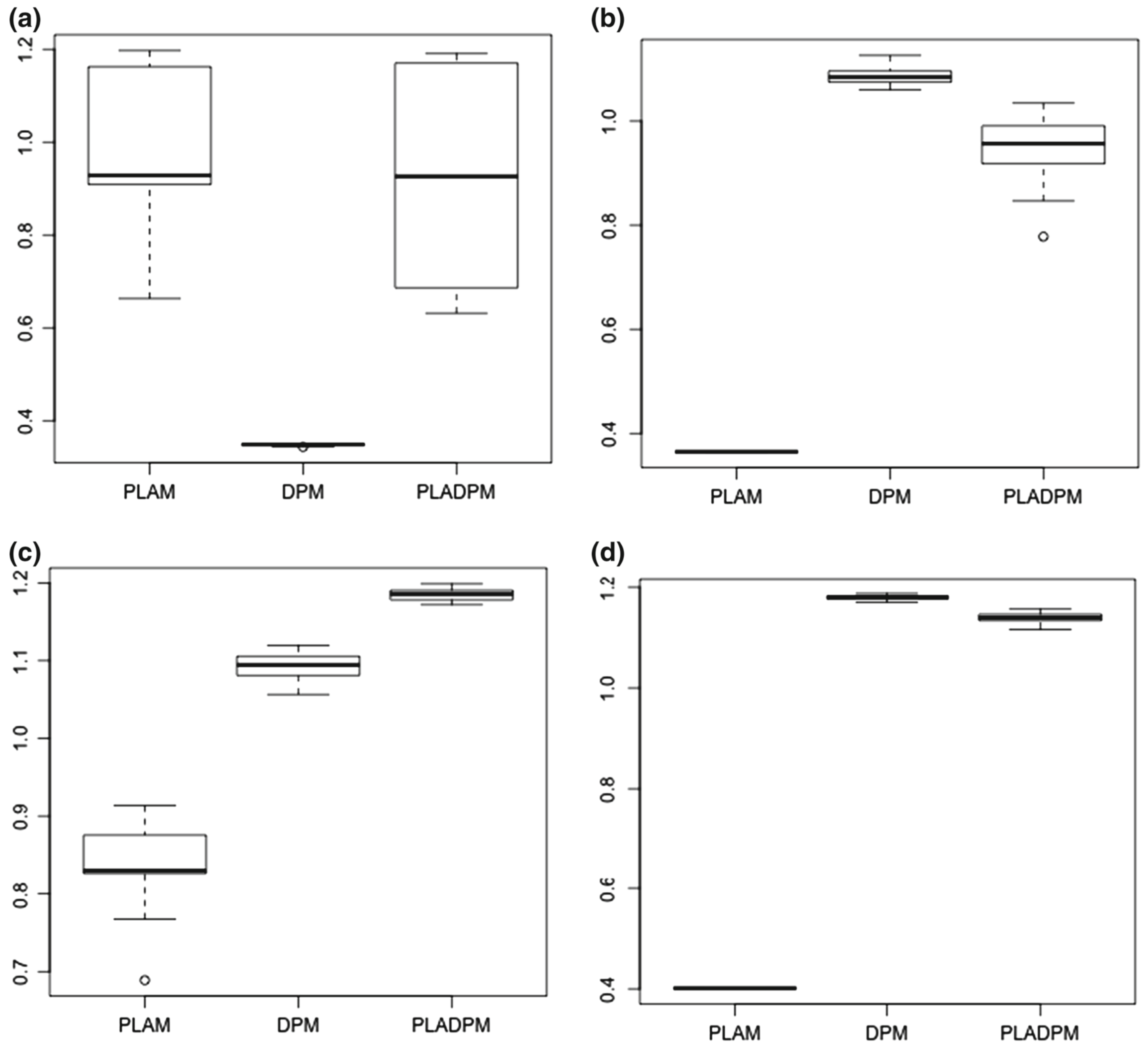


Fig. 14 Boxplot graphs for the obtained *hypervolume* values. **a** AGM, **b** MOS, **c** MM and **d** ETC

Table 7 Solutions with the lowest and greatest ED values by experiment

PLA	Ideal solution	PLAM		DPM		PLADPM	
		−ED	+ED	−ED	+ED	−ED	+ED
AGM	(573.0, 2.005)	0.058 (641, 2.091)	0.509 (573, 4.111)	0.766 (853, 5.037)	1.19 (1758, 5.012)	**0.054 (634, 2.1)**	0.824 (1609, 2.006)
MM	(152.0, 0.006)	0.285 (169, 0.1)	0.591 (152, 0.2)	0.142 (351, 0.043)	1 (221, 0.333)	**0.093 (302, 0.029)**	0.869 (2128, 0.007)
ETC	(705.0, 0.007)	0.748 (720, 0.02)	0.947 (705, 0.026)	**0.119 (1188, 0.003)**	0.815 (6003, 0.001)	0.171 (1437, 0.004)	0.973 (706, 0.026)
MOS	(554.0, 0.012)	0.779 (567, 0.1)	0.779 (567, 0.1)	**0.204 (703, 0.032)**	1 (543, 0.125)	0.295 (896, 0.034)	1 (543, 0.125)

The best ED values were obtained by DPM or PLADPM. Therefore, applying design patterns with *PLA Operators* or even alone can provide solutions with the best *trade-off* between the considered objectives. However, the solutions with the worst ED were also found by DPM or PLADPM for all PLAs. This points out that not every solution with design patterns has the best *trade-off* and that sometimes a slight improvement of one objective can significantly worsen the other one.

8.1.4 Execution time

Table 8 presents information about execution time. The first column presents the PLAs, and the columns two, three and four present, respectively, the average time in milliseconds for each fitness evaluation by the experiments PLAM, DPM and PLADPM (standard deviation in parentheses). The lowest values are highlighted in bold.

Table 8 Average execution time per fitness evaluation

PLA	Average in milliseconds		
	PLAM	DPM	PLADPM
AGM	16.00 (1.20)	**15.21** (2.35)	16.79 (0.86)
MOS	**4.10** (0.01)	19.68 (4.95)	7.14 (0.78)
MM	**9.18** (2.20)	93.08 (15.97)	41.42 (6.26)
ETC	**67.17** (1.06)	1,001.75 (97.78)	247.17 (40.59)

Considering the execution time, we observe that for MM, MOS and ETC, the PLAM experiment was faster than PLADPM, which in turn was faster than DPM. For AGM, PLADPM was the slowest. This can be explained by the several PS and PS-PLA verifications done by the *Pattern-Driven Mutation Operator* before applying design patterns, besides verifying if the given design pattern is already applied in the selected scope. The verifications and pattern application are more expensive than the mutations done by the *PLA Operators*. Only for AGM the DPM experiment was faster. In most cases applying design patterns demanded more time as expected.

8.1.5 Discussion

We observed a tendency for strong optimization of one objective in each experiment. As shown in Fig. 13, the PLAM experiment concentrated the majority of its solutions on the left side of the objective space, minimizing more the FM objective. On the other hand, the *Pattern-Driven Mutation Operator* tends to spread its solutions on the bottom part of the objective space, minimizing more the CM objective. We expected this correlation, since the *PLA Operators* focus mainly on the feature modularization (measured by the FM function), whereas design patterns focus on the cohesion and coupling (measured by the CM function). This tendency of *PLA Operators* in optimizing FM can be supported by the fact that one of them, the *Feature-driven Mutation Operator* [6], was designed specially to modularize features and by the fact that the experiments of Colanzi et al. [6] also point to a similar conclusion. Furthermore, the tendency of the *Pattern-Driven Mutation Operator* in optimizing CM can be explained by the initial motivation of the design patterns and by the purpose for which they were designed [14]: to decrease coupling and to increase cohesion.

The fact that the operators are associated with the optimization of a single objective can also be supported by their functionalities. The *PLA Operators* rarely remove relationships between elements, whereas all operators move methods, attributes and operations from one class or interface to others, or move classes or interfaces from one package to another. Usually, each one of these movements creates new relationships between elements, and in some cases, these relationships are bidirectional, which increases the coupling between the elements. Moreover, moving elements allows the modularization of these elements and of their features, which affects the FM objective directly.

The *Pattern-Driven Mutation Operator* never moves a class, interface, method or attribute, whereas in almost every design pattern application at least one relationship is removed from the elements. This relationship removal usually comes with an interface or an abstract class to abstract the communication between classes and interfaces, which decreases coupling. On the other hand, the interfaces and classes created by the operator usually are associated with all concerns/features of the interfaces and classes that they abstract, which in turn negatively affects the Feature-driven Metrics [40]. Adding to the fact that this operator does not move elements, a good feature modularization may not be achieved. This can be observed in the Pareto fronts shown in Fig. 13. The majority of solutions obtained by the experiment DPM are concentrated to the right of the original PLA in the objective space, which indicates that usually the original PLA has a better feature modularization than the solutions obtained by the *Pattern-Driven Mutation Operator*. In some cases, such as for the PLAs MOS and ETC, the *Pattern-Driven Mutation Operator* was able to yield solutions with better FM values compared to the original PLA.

Therefore, when a PLA has tangled features, which is the case of AGM, the usage of *PLA Operators* is more efficient. However, when the features of a PLA are well modularized, which is the case of MOS, then design patterns seem to be the best option. Nevertheless, the usage of both kinds of operators in combination (such as in the experiment PLADPM) appears to obtain the best *trade-off* and consequently can be considered most appropriate. Despite this combination showing the best *hypervolume* results only for two of the four PLAs of this study, in most cases this combination yielded a greater diversity of solutions and never stood too far from the best results (as shown in Fig. 14). In addition, by using all operators the optimization can benefit from both kinds of functionalities. Furthermore, as we will show further in this section, applying both kinds of operators is not much more expensive (execution time) than applying only MOA4PLA Operators.

Another observed point was the correlation between the size of the PLAs and the population size. For the PLAs MOS and MM (smaller in number of classes and interfaces), for all experiments, a population size of 50 was enough to obtain the best *hypervolume* results. Using a lower population size for small problems is more favorable, because with a smaller population the number of generations increases (max fitness evaluation divided by the population size), and increasing the number of generations the solutions have more time to converge.

For bigger PLAs with more scopes to receive the mutations, it is better to increase the population size. However, only experiments that can perform better for these problems can benefit from a bigger population. For instance, for ETC the experiments DPM and PLADPM obtained better *hypervolume* results than PLAM; thus, a bigger population for them (100 and 200) is favorable, because these experiments can better explore the search space and generate a greater number of solutions. For PLAM, a population of 50 is enough for the ETC optimization, since this experiment cannot perform as well as the other two for this PLA. In this case, it is better to decrease the population size and let the experiment evolve more the few solutions found. The same occurs for AGM. Because the PLAM and PLADPM experiments can perform better for AGM, then a population of 200 provides a better exploration of the search space. Moreover, a population size of 50 is enough for DPM for this PLA.

When a PLA can be optimized by an experiment, then a greater population size (without increasing the number of maximum fitness evaluations) can generate better results, because it allows a better exploration of the search space. When a PLA is less likely to be optimized by a certain experiment the best choice is to decrease the population size, maintaining the number of maximum evaluations and, as a consequence, increasing the number of generations. The problem here is that the architect generally does not know if the PLA is highly or less probable to be well optimized by an experiment. However, if he/she knows the PLA and is aware of the level of feature modularization, then it is possible to identify which kind of operator is most suitable for the optimization and consequently he/she can select the most suitable population size.

8.2 Qualitative analysis

The next subsections present, respectively, the goals of the qualitative analysis, the results and discussion and some final remarks.

8.2.1 Goals

Now for the qualitative analysis, we aim at verifying if the *Pattern-Driven Mutation Operator* was able to qualitatively improve the PLA design. Taking into account that the analysis of all solutions is infeasible, we selected the AGM solution with the lowest ED (Table 7) found by the experiment DPM. This solution was chosen because it represents the best *trade-off* found using only the *Pattern-Driven Mutation Operator*. Furthermore, we chose AGM because this PLA is the one with the worst metric values and the most problematic in terms of design structure.

The main objective of this qualitative analysis is to evaluate the quality of the solution by comparing it to the original PLA. The idea is to analyze the solution to asses: (i) its quality; (ii) if the solution is useful for the architect; (iii) if the design patterns were correctly applied in suitable scopes; and (iv) if any inconsistency (anomaly) was observed in the PLA design after the optimization. To this end, we conducted the qualitative analysis following some specific objectives:

1. To identify which design patterns were applied in the solution;
2. To identify in which scopes these patterns were applied and if they were combined;
3. To identify how many times each design pattern was applied;
4. To analyze the understandability of the solution;
5. To analyze any violation of design pattern structures;
6. To analyze the effects of the pattern application on the software metrics;
7. To identify potential introduction of anomalies.

We considered these objectives because we acknowledge that an indiscriminate design pattern application may overload the architecture and degrade its quality. On the other hand, the correct application of design patterns with structures not very complex/hard to understand can improve the PLA quality and increase the considered metrics values. Therefore, with these specific objectives we were able to analyze the architectures as a whole. The considered anomalies mentioned are the following [29]:

1. *Ambiguous Interface*: refers to interfaces that are ambiguous, i.e., do not reveal their exact purpose and the functionalities they provide;
2. *Scattered Functionality*: happens when a functionality or concern is scattered throughout the architecture in several components;
3. *Component Responsibility Overload*: occurs when a given component/package has several functionalities and is associated with several concerns;
4. *Bloated Interface*: refers to interfaces that are overloaded with many operations.

We considered these anomalies because, given the functionality of the *Pattern-Driven Mutation Operator*, it is likely that they are introduced into the architectures, because this operator creates or changes new interfaces, and their concerns can be interlaced.

8.2.2 Results

Figure 15 depicts a scope of the AGM PLA before the optimization process. Due to space restrictions, the image depicts

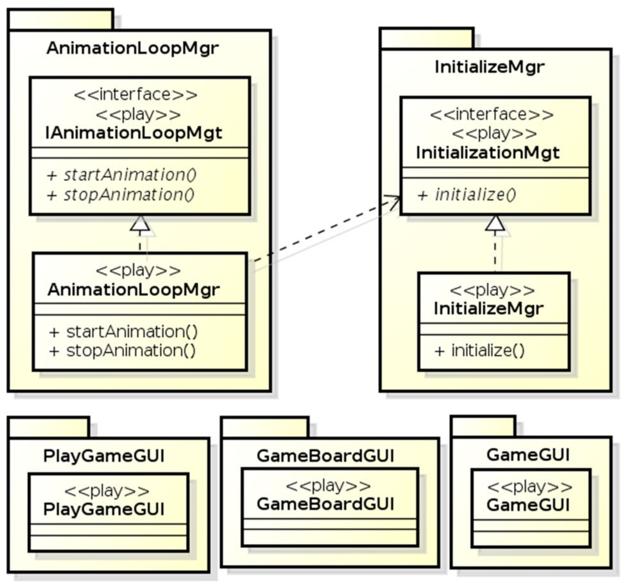


Fig. 15 AGM before the optimization process

only the relevant excerpt of the architecture diagram. Also, the relationships between packages included and the ones not included in the figure were omitted.

After the optimization process, the architecture depicted in Fig. 16 was obtained. The mutated scope of Fig. 16 has an instance of *Mediator*, two instances of *Bridge* and one instance of *Strategy*. These were the only design pattern applications identified in the whole architecture; therefore, as we did for Fig. 15, only a relevant excerpt of the diagram is illustrated.

The first impression when looking into both figures is that the architecture became overloaded. The original AGM design has 46 classes and interfaces, and the mutated one has 55, which is an increase of approximately 19% in this kind of elements. This was expected, since the application of design patterns in this work never deletes classes and interfaces. However, this solution had its CM value decreased when compared to the original one (as shown in Fig. 13a). This objective function, among other things, measures the number of elements in the architecture; hence, the cohesion and coupling gained surpassed the addition of these 9 new classes, interfaces and their new methods.

This increase in the number of elements resulted in a hard understanding of the new structure. In fact, this hard understanding was the major drawback resulting from the pattern application. However, if each design pattern is analyzed separately considering their purposes and applicabilities, it is possible to understand what is the meaning of this scope. The *Strategy* instance has a *Strategy* interface "GUIStrategy" to abstract all elements with the "GUI" suffix, i.e., elements that control the graphic interface. The *Mediator* instance was created to mediate the communication between elements associated with the *play* concern through the "PlayMediator" interface and "PlayMediatorImpl" class. Finally, the *Bridge* instances with the *abstraction* classes "DettachMediatorAbstraction" and "InitializeAbstraction" were created to abstract, respectively, how *Mediator* objects are attached and detached from *colleagues* and how elements are initialized. At the end of the evolutionary process, all elements from the scope were considered *colleagues* of the *Mediator* instance. Despite the original structure having only one relationship, the application of each design pattern in this scope generated new relationships between the elements, which explains the application of *Mediator*. Evaluating other scenarios and other possibilities of output, maybe only the application of *Mediator* would be enough to improve the quality of the architecture. Nevertheless, the combination of such patterns was coherent with their functionalities and we considered as "successful" the application of *Mediator* within our approach.

To overcome the limitation related to the hard understanding, we can use restrictions for combining design patterns in order to increase the understandability of their combination. In other words, we can design the application methods to consider the existence of other design patterns in the scopes and apply procedures for specific patterns combination. These restrictions are not treated in this work, but can be considered in the future.

With the presented result, the original architecture was improved considering the PLA extensibility and coupling metrics. The PLA extensibility improvement is due to the easier inclusion and removal of elements in the scope, since the architect now only needs to identify the purpose of the new element and to include it in the instance of the design pattern that adequately abstracts it. The coupling was decreased because all interactions are now managed by the *Mediator* pattern and the *play* concern is realized through this pattern.

Another observed peculiarity of the scope was that all elements that took part in the mutations are all exclusively associated with the *play* concern. Therefore, the anomaly *Component Responsibility Overload* was not introduced in this solution, because, besides staying in their original packages, only elements that are associated with a single concern were mutated. In addition, the created and changed elements do not present the *Bloated Interface* and *Scattered Functionality* anomalies, since they have at maximum four methods, the elements were created in the same packages/components of the existing elements and the name of the new elements are straightforward and reflect their purposes: "PlayColleague," "DettachMediator" and "InitializationAbstraction."

Furthermore, as shown in Fig. 16 an instance of *Strategy* was applied for elements with the "GUI" suffix. Other possible suffixes are "Mgr", "Mgt" and "Ctrl". These suffixes identify the layers [16] in which the elements are located in the architecture. Elements with a same suffix or prefix are

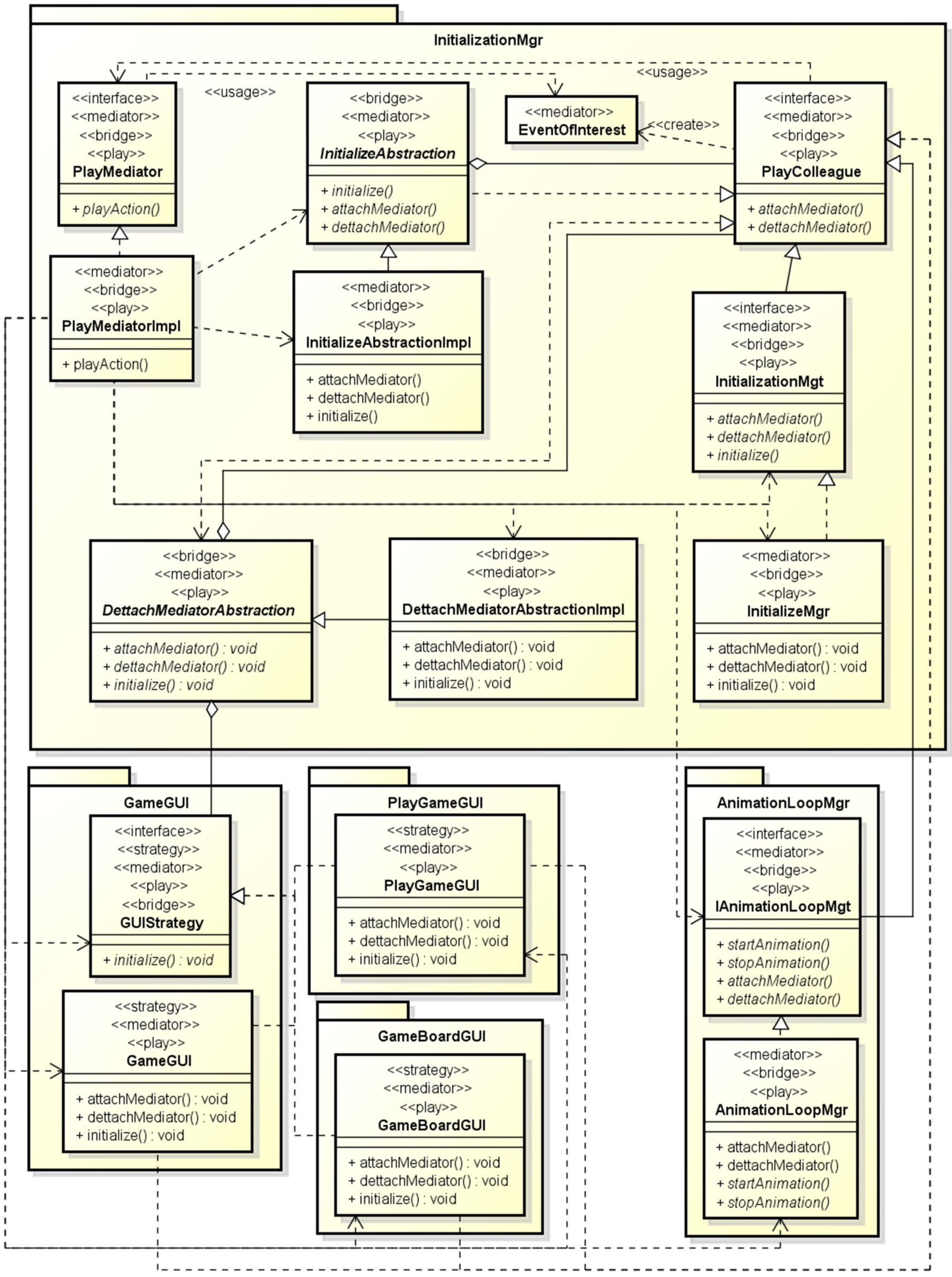

InitializationMgr
<<interface>> <<mediator>> <<bridge>> <<play>> PlayMediator
+ playAction()
<<usage>>
<<bridge>> <<mediator>> <<play>> InitializeAbstraction
+ initialize()
+ attachMediator()
+ dettachMediator()
<<mediator>> EventOfInterest
<<create>>
<<usage>>
<<interface>> <<mediator>> <<bridge>> <<play>> PlayColleague
+ attachMediator()
+ dettachMediator()
<<mediator>> <<bridge>> <<play>> PlayMediatorImpl
+ playAction()
<<mediator>> <<bridge>> <<play>> InitializeAbstractionImpl
+ attachMediator()
+ dettachMediator()
+ initialize()
<<interface>> <<mediator>> <<bridge>> <<play>> InitializationMgt
+ attachMediator()
+ dettachMediator()
+ initialize()
<<bridge>> <<mediator>> <<play>> DettachMediatorAbstraction
+ attachMediator() : void
+ dettachMediator() : void
+ initialize() : void
<<bridge>> <<mediator>> <<play>> DettachMediatorAbstractionImpl
+ attachMediator() : void
+ dettachMediator() : void
+ initialize() : void
<<mediator>> <<bridge>> <<play>> InitializeMgr
+ attachMediator() : void
+ dettachMediator() : void
+ initialize() : void
GameGUI
<<interface>> <<strategy>> <<mediator>> <<play>> <<bridge>> GUIStrategy
+ initialize() : void
<<strategy>> <<mediator>> <<play>> GameGUI
+ attachMediator() : void
+ dettachMediator() : void
+ initialize() : void
PlayGameGUI
<<strategy>> <<mediator>> <<play>> PlayGameGUI
+ attachMediator() : void
+ dettachMediator() : void
+ initialize() : void
GameBoardGUI
<<strategy>> <<mediator>> <<play>> GameBoardGUI
+ attachMediator() : void
+ dettachMediator() : void
+ initialize() : void
AnimationLoopMgr
<<interface>> <<mediator>> <<bridge>> <<play>> IAnimationLoopMgt
+ startAnimation()
+ stopAnimation()
+ attachMediator()
+ dettachMediator()
<<mediator>> <<bridge>> <<play>> AnimationLoopMgr
+ attachMediator()
+ dettachMediator()
+ startAnimation()
+ stopAnimation()

Fig. 16 AGM after the optimization process

present in a same layer; however, they are not necessarily part of a same algorithm family, which can generate pattern applications in scopes that were verified as suitable, when actually they are not. In the example it is justified because the intention is to abstract the elements that manage the graphic interface.

In the original design, the PLA AGM has an interface named "IGameBoardMgt" that declares 13 operations and can be considered an example of *Bloated Interface*. However, this interface was not modified by the proposed operator and the anomaly was not corrected. Even though the *play* concern had been abstracted by the pattern application, this concern stood scattered throughout the architecture as it was in the original design; thus, the *Scattered Functionality* anomaly was not corrected. In the original PLA there is a package named "GameBoardCtrl" that has elements associated with 8 different concerns and can be considered an example of *Component Responsibility Overload*. This anomaly was also not corrected by the operator. Although these anomalies have not been corrected, new anomalies were not introduced in the architecture by the *Pattern-Driven Mutation Operator*, which points to a capability of the operator in optimizing architectures without degrading its quality. It is possible that the PLADPM experiment corrected such anomalies because the Feature-driven Operator of MOA4PLA is able to improve feature modularization, which impacts directly on the referred anomalies.

In the solution, all elements that received the mutation are associated with the *play* concern. Moreover, the *Mediator* pattern was applied just to intermediate the communication of elements associated with this concern. This indicates a potential design problem in the modularization of the *play* concern in the original PLA. This problem was not corrected by the operator, but was identified by it, since this feature received several design pattern applications.

8.2.3 Final remarks

Summarizing the results, we can note that no quality decrement was observed by the application of a specific pattern or by the application of a pattern in a specific scope, but rather when a design pattern encompassed several elements or when several design patterns were applied in a single scope. Nevertheless, this quality degradation is focused on the understandability of the architecture, and when a design pattern was applied in a small scope the obtained architecture quality was good.

We did not observe the application of design patterns in PSs-PLA in the evaluated solution, only in PSs. A factor that prevented this to happen may be the way in which the variation points and variants are modeled in the diagrams by the architect, which is not compliant with the PS-PLA requirements defined. Moreover, the relationships are often modeled as associations, aggregations and compositions, and not as usage or dependency, which are required by the verification methods. However, other qualitative analysis, PLAs designed by other architects and a deeper study of the solutions can lead to identification of mutated PSs-PLA.

The design pattern application of this work does not concern with the consistency of architectural styles in the PLA, which in some cases can be violated. This problem had been already identified by Colanzi et al. [6] in their study, but only using their operators. Mariani et al. [31,32] proposed mutation operators aiming at preserving the architectural styles of PLAs being optimized by search-based approaches. Some restrictions or other operators can be similarly proposed to preserve architectural styles when applying design patterns. However, it can be complex to do this, since the mutations performed by the *Pattern-Driven Mutation Operator* usually encompass several elements, but they are necessary to ensure the organization and quality of the architecture.

It is difficult to determine if the flexibility provided by the application of a design pattern is really needed. Furthermore, the application of each design pattern or their combination can present other characteristics, such as a high complexity. The choice of which architecture is the best and which design pattern is the best depends on the needs and preferences of the architect. The pattern application allows MOA4PLA to obtain a greater diversity of solutions with good metric values; however, the architect needs to decide which solution (architecture) will be used as PLA for the SPL.

8.3 Answering the research question

Lastly, taking into account the quantitative and qualitative results presented in this section, we can positively answer the research question considered for this study: automatically applying design patterns using the *Pattern-Driven Mutation Operator* contributes to the improvement of the PLA design being optimized. Moreover, we observed that the proposed operator provided a greater diversity of solutions for the architect to make his/her decisions.

9 Related work

The application of design patterns in PLAs is not a completely established research field. Works on this subject usually offer manual or semi-automatic approaches in specific domains [27,45]. Keepence and Mannion [27] proposed a method to use design patterns to model variabilities in class diagrams. They defined three patterns that aim at modeling specific variant types: (i) *Single Adapter*—for mutually exclusive variants; (ii) Multiple Adapter—for mutually inclusive variants; and (iii) *Optional*—for optional variants. These patterns have some characteristics that cannot be identified in suitable scopes of class diagrams that prevent their automatic application by a search-based approach, such as MOA4PLA.

Fant et al. [13] presented a pattern-based modeling approach for SPL. This approach uses architectural patterns to specify variabilities at a higher level of granularity. The authors stated that their approach is useful when the variabilities are not all known during the domain engineering (development of the PLA), which decreases the design effort during that phase. This approach was validated on a space flight software, and domain-specific patterns were created. Although the functional correctness of the evaluated architecture was ensured, the approach is still manual and is focused more on the variability modeling problem, rather than on the optimization of the architectures.

Ziadi et al. [45] proposed an approach based on the *Abstract Factory* pattern [14] structure to model and derive PLAs. Their approach includes the usage specification and an Object Constraint Language (OCL) algorithm for the model derivation. However, we did not use their approach in this work because it has some specific characteristics (such as structure and relationship rules between variants and variation points) that cannot be automatically identified in scopes of a class diagram.

We have not found in the literature works that automatically apply design patterns in search-based design of PLAs. The works most related to ours apply design patterns for search-based design of architectures in general, as described in the survey of search-based software design written by Räihä [36]. We give a brief overview of such works in the next paragraphs, but for a more detailed overview, we recommend the mentioned survey.

According to the systematic review of Mariani and Vergilio [30], some search-based algorithms have been used in order to find sequences of design patterns' applications to improve the quality of software architectures [1,22,25,35, 41]. Such works optimize the architectural design indirectly or only use semi-automatic refactoring. We present next the works most related to ours in terms of design pattern application in conventional architectures.

Räihä et al. [38] used genetic algorithms to apply design patterns in the synthesis of software architectures. The authors used five design patterns: *Adapter*, *Strategy*, *Mediator*, *Template Method* and *Facade*. However, they adopted a different problem representation from ours, which needs extension to represent PLAs. Besides, they did not perform an identification of suitable scopes before applying design patterns.

Cinnéide and Nixon [3] proposed a semi-automatic refactoring approach for applying design patterns in source code. They created *minipatterns*, small transformations that when put together can successfully apply design patterns in the source code. A limitation of this approach is that it is not completely automated. The architect must select which design pattern should be applied and where the refactoring must occur.

10 Concluding remarks

This work investigates a way to allow the application of design patterns in the search-based optimization approach MOA4PLA. However, some challenges had to be overcome before automatically applying design patterns in software architectures. First, we conducted a feasibility analysis to determine which GoF patterns could be applied in the MOA4PLA context, in order to avoid an indiscriminate application and introduction of anomalies. In the feasibility analysis we found four design patterns to be feasible: *Strategy*, *Bridge*, *Facade* and *Mediator*.

To represent suitable scopes to the application of the feasible patterns we introduced PS and PS-PLA concepts, which are metamodels to represent architectural elements that compose the adequate scope to receive each pattern.

Using the PS and PS-PLA concepts, we proposed and implemented a mutation operator named *Pattern-Driven Mutation Operator* that automatically applies different patterns. This operator uses some verification methods to check if a scope can receive the application of a given design pattern. The pattern application is done by the application methods that add, remove or change elements from the scope.

Our empirical evaluation used four PLAs and compared the *Pattern-Driven Mutation Operator* results with the *PLA Operators*, which are conventional operators of MOA4PLA [6]. Our analysis considered the Pareto dominance concepts and the *hypervolume* indicator. This evaluation showed the capability of the proposed mutation operator to find good solutions and with a favorable population diversity in all problems. However, despite some authors [38] claiming that a concurrent optimization using design patterns and conventional operators (such as the *PLA Operators* used by MOA4PLA) is too complex for the software design, in this work the results showed that the usage of both kinds of operators (*PLA Operators* and *Pattern-Driven Operator*) is a more reasonable option in most cases, since it presents a less unpredictable behavior and fronts with solutions, if not better than the ones obtained using only the *Pattern-Driven Operator*, at least close to the best solutions. Using only *Pattern-Driven Operator* showed some quantitative advantages, but for a PLA such as AGM, it was not such an advantageous option.

In addition to this, we also qualitatively evaluated the generated architectures in order to assess their utility. Design patterns were successfully applied in suitable scopes, and the most important improvements obtained are related to cohesion and coupling metrics. Therefore, the design pattern application is quantitatively and qualitatively beneficial for the PLA design in the context of this work.

It is difficult to determine if the flexibility provided by the design patterns is really necessary; hence, despite the good results obtained, the participation and experience of the architect are crucial to decide which is the best solution. The main advantage of this work is to reduce the architect's effort providing different good PLA alternatives.

As future work we intend to extend this approach to other design patterns and other UML diagrams, such as the interaction ones. Other possibility is the creation of *minipatterns* as in [3] to allow a better reusability of transformations. Furthermore, other researches may use an interactive approach to help the architect in guiding the evolution toward his/her goals and consequently improving the quality of the designs. We intend to create restrictions to allow the combination of design patterns, thus decreasing chances of violations in the design patterns structures.

New experiments with bigger PLAs can yield different results and should be conducted. We intend to formulate other objective functions with different software metrics in order to increase the accuracy of the results and to conduct other qualitative analyses with researchers/architects to provide a better evaluation of the solutions.

References

1. Amoui, M., Mirarab, S., Ansari, S., Lucas, C.: A genetic algorithm approach to design evolution using design pattern transformation. Int. J. Inf. Technol. Intell. Comput. **1**, 235–245 (2006)
2. Arcuri, A., Fraser, G.: On parameter tuning in search based software engineering. In: Proceedings of the 3rd Symposium on Search Based Software Engineering (SSBSE), pp. 33–47 (2011)
3. Cinnéide, M.O.: Automated software evolution towards design patterns. In: Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE), pp. 162–165. ACM, New York (2001)
4. Coello, C.A.C., Lamont, G.B., Veldhuizen, D.A.V.: Evolutionary Algorithms for Solving Multi-Objective Problems, 2nd edn. Springer, Berlin (2007)
5. Colanzi, T.E., Vergilio, S.R.: Representation of software product lines architectures for search-based design. In: CMSBSE Workshop of International Conference on Software Engineering (ICSE), pp. 28–33 (2013)
6. Colanzi, T.E., Vergilio, S.R., Gimenes, I.M.S., Oizumi, W.N.: A search-based approach for software product line design. In: Proceedings of the 18th Software Product Line Conference (SPLC), pp. 237–241 (2014)
7. Contieri Junior, A.C., Correia, G.G., Colanzi, T.E., Gimenes, I.M.S., Junior, E.A.O., Ferrari, S., Masiero, P.C., Garcia, A.F.: Extending UML components to develop software product-line architectures: lessons learned. In: Proceedings of the 5th European Conference on Software Architecture (ECSA), pp. 130–138 (2011)
8. Coplien, J.O.: Software design patterns: common questions and answers. In: Rising, L. (ed.) The Patterns Handbook: Techniques, Strategies, and Applications, pp. 311–320. Cambridge University Press, Cambridge (1998)
9. Darwin, C.: On the Origin of Species by Means of Natural Selection, or, The Preservation of Favoured Races in the Struggle for Life. J. Murray, London (1859)
10. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
11. Derrac, J., García, S., Molina, D., Herrera, F.: A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. Swarm Evol. Comput. **1**(1), 3–18 (2011)
12. Donegan, P.M., Masiero, P.C.: Design issues in a component-based software product line. In: Proceedings of Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), pp. 3–16 (2007)
13. Fant, J.S., Gomaa, H., Pettit, R.G.: A pattern-based modeling approach for software product line engineering. In: Proceedings of the 46th Hawaii International Conference on System Sciences (HICSS), pp. 4985–4994 (2013)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, vol. 206. Addison-Wesley Longman Publishing Co., Inc, Boston (1995)
15. Gandibleux, X., Sevaux, M., Sörensen, K., T'Kindt, V.: Metaheuristics for multiobjective optimisation. Lecture Notes in Economics and Mathematical Systems. Springer Berlin Heidelberg (2012)
16. Garlan, D., Shaw, M.: An introduction to software architecture. In: Ambriola, V., Tortora, G. (eds.) Advances in Software Engineering and Knowledge Engineering, vol. I, pp. 1–39. World Scientific, Singapore (1994)
17. Gomaa, H., Hussein, M.: Software reconfiguration patterns for dynamic evolution of software architectures. In: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 79–88. IEEE Computer Society, Washington (2004)
18. Guizzo, G., Colanzi, T.E., Vergilio, S.R.: Applying design patterns in product line search-based design: feasibility analysis and implementation aspects. In: Proceedings of the 32nd International Conference of the Chilean Computer Society (SCCC), Temuco, Chile, SCCC'13 (2013)
19. Guizzo, G., Colanzi, T.E., Vergilio, S.R.: A pattern-driven mutation operator for search-based product line architecture design. In: Proceedings of the 6th Symposium on Search Based Software Engineering (SSBSE), pp. 77–91. Springer, Fortaleza (2014)
20. Harman, M., Clark, J.: Metrics are fitness functions too. In: Proceedings of the 10th International Symposium on Software Metrics (ISSM), pp. 58–69 (2004)
21. Harman, M., Mansouri, A.: Search based software engineering: introduction to the special issue of the IEEE transactions on software engineering. IEEE Trans. Softw. Eng. **36**(6), 737–741 (2010)
22. Harman, M., Tratt, L.: Pareto optimal search based refactoring at the design level. In: Proceedings of the 9th Genetic and Evolutionary Computation Conference (GECCO), pp. 1106–1113. ACM (2007)
23. Hwang, C.L., Yoon, K.: Multiple attribute decision making: methods and applications : a state-of-the-art survey. Lecture notes in economics and mathematical systems. Springer (1981)
24. Institute SE (2014) Arcade Game Maker pedagogical product line. http://www.sei.cmu.edu/productlines/ppl/, Accessed on 8 Aug 2016
25. Jensen, A.C., Cheng, B.H.C.: On the use of genetic programming for automated refactoring and the introduction of design patterns. In: Proceedings of the 12th Genetic and Evolutionary Computation Conference (GECCO), pp. 1341–1348 (2010)
26. Junior, E.A.O., Gimenes, I.M.S., Maldonado, J.C.: Systematic management of variability in UML-based software product lines. J. Univers. Comput. Sci. **16**(17), 2374–2393 (2010)
27. Keepence, B., Mannion, M.: Using patterns to model variability in product families. IEEE Softw. **16**, 102–108 (1999)
28. van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer, Berlin (2007)
29. Macia, I.: On the detection of architecturally-relevant code anomalies in software systems. Ph.d. thesis, Pontifical Catholic University of Rio de Janeiro (2013)
30. Mariani, T., Vergilio, S.R.: A systematic review on search-based refactoring. Inf. Softw. Technol. **83**, 14–34 (2017)

31. Mariani, T., Colanzi, T.E., Vergilio, S.R.: Search based design of layered product line architectures. In: Proceedings of the 39th IEEE Computer Society International Conference on Computers, Software and Applications (COMPSAC). IEEE, Taichung (2015a)
32. Mariani, T., Vergilio, S.R., Colanzi, T.E.: Optimizing aspect-oriented product line architectures with search-based algorithms. In: Proceedings of the 7th Symposium on Search Based Software Engineering (SSBSE), pp. 173–187. Springer, Bergamo (2015b)
33. Oliveira Junior, E.A., Gimenes, I.M.S., Maldonado, J.C., Masiero, P.C., Barroca, L.: Systematic evaluation of software product line architectures. J. Univers. Comput. Sci. **19**, 25–52 (2013)
34. Philippow, I., Streitferdt, D., Riebisch, M.: Design pattern recovery in architectures for supporting product line development and application. In: Riebisch, M., Coplien, J., Streitferdt, D. (eds.) Modelling Variability for Object-Oriented Product Lines, pp. 42–57. Springer, Berlin (2003)
35. Qayum, F., Heckel, R.: Local search-based refactoring as graph transformation. In: Proceedings of the 1st Symposium on Search Based Software Engineering (SSBSE), pp. 43–46. IEEE (2009)
36. Räihä, O.: A survey on search-based software design. Comput. Sci. Rev. **4**(4), 203–249 (2010)
37. Räihä, O., Koskimies, K., Makinen, E.: Complementary crossover for genetic software architecture synthesis. In: 10th International Conference on Intelligent Systems Design and Applications (ISDA), pp. 266–271 (2010)
38. Räihä, O., Koskimies, K., Mäkinen, E.: Generating software architecture spectrum with multi-objective genetic algorithms. In: 2011 Third World Congress on Nature and Biologically Inspired Computing (NaBIC), pp. 29–36. IEEE (2011)
39. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley Professional, Boston (2004)
40. Sant'Anna, C., Figueiredo, E., Garcia, A., Lucena, C.J.P.: On the modularity of software architectures: a concern-driven measurement framework. In: Proceedings of the 1st European Conference on Software Architecture (ECSA), pp. 207–224 (2007)
41. Shimomura, T., Ikeda, K., Takahashi, M.: An approach to ga-driven automatic refactoring based on design patterns. In: Proceedings of the Fifth International Conference on Software Engineering Advances (ICSEA), pp. 213–218 (2010)
42. Simons, C., Singer, J., White, D.R.: Search-based refactoring: metrics are not enough. In: Search-Based Software Engineering, Lecture Notes in Computer Science, vol. 9275, pp. 47–61. Springer International Publishing (2015)
43. Wüst, J.: SDMetrics: the software design metrics tool for the UML. http://www.sdmetrics.com/index.html. Accessed on 8 Aug 2016
44. Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall Inc, Upper Saddle River (1979)
45. Ziadi, T., Jézéquel, J.M., Fondement, F.: Product line derivation with UML. In: Proceedings of Software Variability Management Workshop (2003)
46. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a UML profile for software product lines. In: Linden, F.J. (ed.) Software Product-Family Engineering, Lecture Notes in Computer Science, vol. 3014, pp. 129–139. Springer, Berlin Heidelberg (2004)
47. Zitzler, E., Thiele, L.: Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. Trans. Evol. Comput. **3**(4), 257–271 (1999)
48. Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C.M., da Fonseca, V.G.: Performance assessment of multiobjective optimizers: an analysis and review. IEEE Trans. Evol. Comput. **7**(2), 117–132 (2003)

Giovani Guizzo is a PhD student in Computer Science and is currently doing research on search-based software engineering (SBSE). He has worked with search-based software design of product line architectures in his Master. Giovani is currently doing research on search-based software testing, more specifically applying hyper-heuristics to reduce the cost of mutation testing.

Thelma Elita Colanzi received the DS degree (2014) from Federal University of Paraná (UFPR), Brazil. She holds the MSc degree in Computer Science and Computational Math from University of São Paulo (USP/São Carlos), Brazil. She has been a lecturer at State University of Maringá since 2006. Her areas of interest are software architecture, software product lines, software modelling, search-based software engineering and multi-objective evolutionary algorithms.

Silvia Regina Vergilio received the MS (1991) and DS (1997) degrees from University of Campinas, UNICAMP, Brazil. She is currently at the Computer Science Department at the Federal University of Paraná, Brazil, where she has been a faculty member since 1993. She has been involved in several projects, and her research interests are in the areas of software engineering, such as software testing, software architecture, software metrics and search-based software engineering.