
OPLA-Tool-ASP: a Tool to Prevent Architectural Smells in Search-based Product Line Architecture Design

Tiago Tadeu Madrigar  [Universidade Estadual de Maringá | tiago@madrigar.com.br]

Thelma Elita Colanzi  [Universidade Estadual de Maringá | thelma@din.uem.br]

Willian Nalepa Oizumi  [Pontifícia Universidade Católica Rio de Janeiro | woizumi@inf.puc-rio.br]

Luiz Fernando Okada  [Universidade Estadual de Maringá | ra107247@uem.br]

Alessandro Garcia  [Pontifícia Universidade Católica do Rio de Janeiro | afgarcia@inf.puc-rio.br]

Abstract Search-based algorithms have been successfully employed in Product Line Architecture (PLA) design in the seminal approach named Multi-Objective Approach for Product-Line Architecture Design (MOA4PLA). This approach generates a set of alternative PLA designs, which optimize different architectural properties. In addition to these properties, the alternative PLA designs should have as few architectural smells as possible. Architectural smells can negatively impact PLA variability, PLA extensibility, SPL maintainability, and other non-functional attributes. However, one of the main findings of a previous study is that the tool that automates the application of MOA4PLA adversely introduces architectural smells in the automatically generated solutions. In this work, we present OPLA-Tool-ASP, which is a tool that implements guidelines to detect and prevent the architectural smells Unused Interface, Unused Brick, Concern Overload, and Link Overload in the context of MOA4PLA. An empirical study was carried out to assess the effectiveness of OPLA-Tool-ASP in preventing the aforementioned smells in the resulting PLA designs. The obtained results pointed out that the proposed tool is effective in both preventing the smells and improving the architectural properties selected for optimization.

Keywords: Software Product Lines, Architectural Smells, Search-based Software Engineering.

© Published under the Creative Commons Attribution 4.0 International Public License (CC BY 4.0)

1 Introduction

Search-based algorithms have achieved satisfactory results to solve Software Engineering complex problems, including some problems related to Software Product Line (SPL) [21, 22, 26]. SPL Engineering aims to develop a family of software products to a particular domain by means of reusable artifacts [25]. One of the main artifacts of a SPL is the Product Line Architecture (PLA), which defines a common design for all products derived from the SPL [25]. PLA design is a high-effort task since it is influenced by several factors, such as feature modularity, PLA variability and extensibility. A modular, extensible, legible PLA design eases the SPL maintainability and evolution.

Given the complexity and variety of such influential factors, the use of search-based algorithms have recently been explored to derive PLA designs in the seminal approach named Multi-Objective Approach for Product-Line Architecture Design (MOA4PLA) [9]. In this approach, software metrics are used in the objective functions to guide the search process. Metrics provide indicators to a wide range of architectural properties, such as feature modularization, variability, PLA extensibility, coupling, cohesion, design size and so on. Each solution is evaluated and classified by the objective functions. At the end of the search process, the search-based al-

gorithm returns a set of solutions for the PLA design given as input. This set contains the solutions with the best trade-off among the objectives optimized, which represent different possible designs for that PLA design given as input, hereafter called alternative design or design instance.

MOA4PLA concepts were automated by OPLA-Tool [17], which produces a set of alternative PLA designs that improve the different optimized factors from an initial PLA design, which depicts the module view [6]. Empirical results point out that OPLA-Tool successfully optimize the selected factors in the generated alternative designs [5, 32, 36]. Besides the optimization of these factors, the obtained PLA designs should ideally have no architectural smell. Architectural smells are problems due to architectural design decisions, intentional or not, which negatively impact the system quality [19]. An architectural smell (or architectural problems) may harm not only the PLA variability and extensibility, but also other non-functional attributes, such as understandability, testability, reusability, maintainability and performance [18, 20].

Perissato et al. [35] carried out an exploratory investigation about the presence of architectural smells in a set of PLA designs obtained using OPLA-Tool. The results revealed that OPLA-Tool adversely introduces some ar-

architectural smells in resulting designs. Furthermore, it does not detect and remove the architectural smells existing in the original design given as input for the tool. Perissato et al. [35] also identified the most common architectural smells detected in the alternative design generated by OPLA-Tool and proposed guidelines to detect and prevent those architectural smells. However, such guidelines were not applied and validated.

Existing approaches to identify architectural smells usually are dependent on source code [24, 29, 34, 41]. However, the identification of these problems during the PLA design would: (i) avoid that congenital smells are propagated to the SPL implementation, and (ii) maximize non-functional quality attributes since the PLA conception. Particularly, some quality attributes – e.g., reusability and maintainability – are extremely important to SPL Engineering [25]. However, there is not any approach to neither detect architectural smells nor prevent their introduction in PLA design optimized by search-based approaches [35]. In this sense, it is worth investigating the presence of architectural smells in automatically generated PLA designs in order to avoid severe smells are present in the initial design of a SPL. Perissato et al.'s guidelines are a starting point to achieve this goal.

In this context, the main objective of this work is contributing to the evolution of the PLA design optimization by means of a tool to detect and prevent architectural smells during the search-based PLA design. For doing so, in this work we present a tool named OPLA-Tool-ASP (Optimization for PLA Tool - Architectural Smells Prevention). This tool is an evolution of OPLA-Tool that incorporates Perissato et al.'s guidelines [35] to prevent architectural smells in PLA design automatically generated, particularly those ones that arise in the module view of the architectural design. OPLA-Tool-ASP detects and prevents four smells that are the most frequent in the design instances investigated in [35]: Unused Interface [18], Unused Brick [18], Concern Overload [18] and Link Overload [18]. It is important to prevent Unused Interface and Unused Brick smells because they are the architectural smells adversely introduced by OPLA-Tool in alternative designs. Concern Overload and Link Overload negatively impacts on modularity, extensibility and reusability, which are architectural properties optimized by MOA4PLA. A preliminary version of OPLA-Tool-ASP was introduced in [28] encompassing the guidelines for the three first architectural smells. In this work we extend our tool including the guideline to prevent Link Overload and carry out another empirical evaluation of OPLA-Tool-ASP.

To evaluate the results obtained by OPLA-Tool-ASP, we performed an empirical study to compare OPLA-Tool-ASP and OPLA-Tool [17] aiming at answering the following research question RQ - Is OPLA-Tool-ASP effective to detect and prevent Unused Interface, Unused Brick, Concern Overload and Link Overload?. Our study involved the PLA design of three SPLs, one more than in our previ-

ous study [28]. The study encompassed 81 solutions (alternative designs), being 23 solutions generated by OPLA-Tool-ASP and 58 solutions obtained by OPLA-Tool. Each solution was inspected searching for the occurrence of the 4 architectural smells aforementioned. A total of 1450 occurrences of smells were detected in the alternative designs analyzed during the study.

The main contribution of this work is the increment of state of the art by providing a tool to detect and prevent the architectural smells Unused Interface, Unused Brick, Concern Overload and Link Overload in alternative PLA designs. Such a tool is useful for: (i) improving the resulting search-based PLA designs, or (ii) preventing upfront the emergence of architectural smells in PLA designs automatically obtained with search-based approaches. Other contribution is the implementation and application of the Perissato et al.'s guidelines in the alternative design generated from the original PLA design of three SPLs.

The remaining of this paper is organized in sections. Section 2 presents the main concepts related to our work as well as related work. Section 3 introduces OPLA-Tool-ASP, a tool that apply guidelines to detect and prevent architectural smells during the optimization of PLA design using search algorithms. The design of the empirical study is addressed in Section 4. Section 5 presents the results, highlights our findings and answers the research questions. Section 6 addresses the main threats to the validity of our study. Finally, Section 7 concludes the paper and points to future work.

2 Background

In this section, we present the main concepts related to this work.

2.1 Search-based Software Engineering

In the Search-based Software Engineering (SBSE) field [22], Software Engineering problems are formulated as optimization problems, with the goal of minimizing or maximizing a function or a group of factors through search techniques. As a result, it is expected to reach (quasi-)optimal solutions. In a SBSE approach, there are usually two main aspects: a search space that contains all possible solutions for the problem; and a fitness function, which is responsible for evaluating the quality of the solutions [22].

Genetic algorithms are among the most used SBSE techniques. A Genetic Algorithm (GA) [7] is a meta-heuristic inspired by natural selection and genetic evolution theory. Starting from an initial population – which in our case are alternatives for the PLA design – search operators are applied to evolve the population throughout multiple generations. The selection operator is responsible for selecting solutions that present the best fitness values to survive as parents for the next generation. The crossover operator combines parts of two

parent solutions to create a new one. Finally, the mutation operator randomly changes a solution. The offspring population created from the selection, crossover, and mutation replaces the parent population.

Some GAs are adapted for optimizing multi-objective problems. Such GAs are called Multi-Objective Evolutionary Algorithms (MOEAs). The most popular and largely applied MOEA is the Non-dominated Sorting Genetic Algorithm (NSGA-II) [11]. A multi-objective problem depends on multiple factors (objectives) that may be conflicting and, therefore, there is not a single possible solution. Therefore, there may be multiple (quasi-)optimal solutions that represent the trade-off between the different objectives. Such solutions are called non-dominated solutions and form the Pareto front [7].

2.2 PLA Design Optimization

Colanzi et al. [9] proposed the MOA4PLA approach, which automates the search for the best PLA designs using MOEAs. MOA4PLA receives as input a PLA design modeled on a UML class diagram containing all the common and variable architectural elements. Such an input contains a detailed design, equivalent to the module view [6], including the classes and interfaces structure. Stereotypes are used to associate each element with the features they perform. The SPL architect must then choose which MOEA to use in the optimization process as well as the execution parameters. The design received as input is optimized through the search operators defined by MOA4PLA. These operators perform basic movements of the GAs which are the following: crossover operators for existing individuals and mutation operators for newly generated individuals. The approach includes specific mutation operators for PLA designs, which are: Move Method, Move Attribute, Add Class, Move Operation, Add Component, and Feature-driven Operator [9]. The latter aims to improve the modularization of interlaced and scattered features in architectural elements. MOA4PLA also has crossover operators specific for PLA design optimization [36].

The MOA4PLA approach employs an evaluation model proposed for PLA designs [38]. Such a model provides a set of objective (fitness) functions based on software metrics. The metrics are related to multiple architectural properties, such as feature modularization, variability, cohesion, and coupling, which are related to structural quality attributes, such as modularity, extensibility, understandability and changeability [38]. Other quality attributes, such as availability, security or scalability can only be addressed using the component view, which is not addressed by MOA4PLA.

MOA4PLA requires the architects to select the subset of objective functions that they want to optimize. The value of the objective functions defines the fitness of each design alternative obtained during the optimization process. The quality of each alternative design is evaluated according to its fitness, which is directly related to the objectives selected by the architects from the evaluation model. In this study, three objective

functions were selected, namely: COE, ACLASS and FM [36, 38]. In what follows, we present the equations of the three objective functions used in our experiments.

The COE objective function evaluates the cohesion of PLA design in terms of the internal relationship of the classes of the PLA design, measured by the H metric (Equation 1). ACLASS measures class coupling by the number of architectural elements that depend on other classes of the design ($CDepIn$), added to the number of elements on which each class depends ($CDepOut$) according to Equation 2. In the case of Equations 1 and 2, c is the number of classes.

$$COE(pla) = \frac{1}{\sum_{i=1}^c H} \quad (1)$$

$$ACLASS(pla) = \sum_{i=1}^c CDepIn + \sum_{i=1}^c CDepOut \quad (2)$$

FM deals with the modularization of a PLA in terms of its features. It evaluates the feature modularization of the PLA design, which is formed by metrics specific to SPL features [33] according to Equation 3, where given a PLA design pla , c is the number of components and f is the number of its features. FM provides an indicator for feature modularization based on the sum of the metrics for feature scattering ($CDAC, CDAO, CDAI$), feature interlacing ($CIBC, IIBC, OIBC$) and feature-driven cohesion (LCC) [38].

$$FM(pla) = \sum_{i=1}^c LCC + \sum_{i=1}^f CDAC + \sum_{i=1}^f CDAI \\ + \sum_{i=1}^f CDAO + \sum_{i=1}^f CIBC + \sum_{i=1}^f IIBC \quad (3) \\ + \sum_{i=1}^f OIBC$$

All objective functions were conceived to be minimized during the search process. This means that the algorithms employed should search to maximize each function's minimization as an objective.

After the search process, MOA4PLA returns a set with the best trade-off alternative designs among the optimized objectives. The architects must choose one of the solutions obtained according to their priorities to adopt as a PLA design.

OPLA-Tool [13, 17] is a tool that automates the application of the MOA4PLA approach. Such a tool allows the selection of objective functions, search operators, and MOEAs. NSGA-II is among the MOEAs provided by the OPLA-Tool. Besides that, the OPLA-Tool also provides a visualization mechanism for PLA alternative designs.

As the main objective of MOA4PLA is the optimization of PLA designs, aiming to maximize quality attributes such as reusability, extensibility and maintainability, it is desirable that the generated design solutions

are free from architectural smells – which can directly affect the aforementioned attributes. However, existing versions of the OPLA-Tool do not prevent architectural smells.

2.3 Identification of Architectural Smells in PLA Design

Architectural smells are problems arising from architectural design decisions that negatively affect the quality of the system [19]. The existence of architectural smells can negatively impact quality attributes, such as comprehension, testability, extensibility and reusability [16, 18]. After performing a systematic mapping of the literature, we identified that there are no tools for the prevention of architectural smells in PLA designs optimized by SBSE techniques.

Four architectural smells were initially selected to be investigated in this work, namely: Unused Interface, Unused Brick, Concern Overload and Link Overload. We selected such smells because they were identified as the most frequent in the 24 PLA design instances generated by OPLA-Tool analyzed in [35]. See the description of each smell in Table 1.

Table 1 presents the identification strategies used in the previous study [35] as well as the proposed guidelines for preventing such smells in the solutions generated by OPLA-Tool. However, none of the proposed guidelines has been previously implemented.

The proposal of Perissato et al. [35] for preventing Unused Interface and Unused Brick was to discard solutions that contained such smells, avoiding their propagation between the solutions generated during the optimization process. For Concern Overload, we proposed the application of the Feature-driven Operator, which provides the modularization of some of the features associated with the smelly class/interface to solve or mitigate the manifestation of the smell.

Regarding the Link Overload smell, in our previous work we identified several false positives when using the detection strategy proposed by Garcia [18]. That happened because inheritance relationships, which are commonly used to resolve SPL variabilities, are considered as regular dependencies for detecting Link Overload [35]. Thus, we suggested to changing the identification strategy to not consider inheritance relationships when detecting Link Overload in SPL. Moreover, we pointed out the application of a penalty to the solution fitness as a guideline for preventing such a smell [35].

2.4 Related Work

In this section we present and discuss the papers that are related to this work. We organized the discussed papers into the following categories: (1) identification of architectural smells and (2) architectural smells in SPLs, and (3) SBSE approaches for identifying and removing smells.

Identification of architectural smells. In order to find studies that propose strategies for the detection and

identification of architectural smells, we conducted a systematic literature mapping. After applying inclusion and exclusion criteria, 30 papers were approved. Below, present and discuss the most relevant ones.

Fowler [15] and Garcia [18] provided catalogs of code smells and architectural smells, respectively. Azadi et al. [3] also proposed a catalog of architectural smells. Their catalog was based on the types of smells that are automatically detected by existing tools. Finally, Mo et al. [31] proposed and evaluated a suite of automatically detectable architectural smells that occur in large-scale systems. Their results provided evidence that the proposed smells significantly impact files bug-proneness and change-proneness. However, none of the aforementioned catalogs were designed and evaluated in the context of PLA design.

Macia et al. [27] investigated how code smells could be related to architectural smells. The authors presented a set of detection strategies based on metrics focused on architecture. Vidal et al. [39] also addressed how code smells can affect the system's architecture. The authors implemented a set of three scoring criteria to prioritize code smells, reducing the number of locations that possibly contained problems, helping developers in the identification process. The authors' strategies were based on the identification of multiple correlated code smells. However, none of the aforementioned studies focused on the identification of architectural smells in PLA design.

Architectural smells in SPLs. Perissato et al. [35] investigated which architectural smells are more frequent in PLA design solutions generated by OPLA-Tool. Based on this investigation, they proposed guidelines for the identification and prevention of the most frequent architectural smells. Although their work focused on PLA design, none of their proposed guidelines was evaluated.

SBSE approaches for identifying and removing smells. The work of Mansoor et al. [29] addressed the detection of code smells using multi-objective evolutionary algorithms. The authors applied a set of metrics using the NSGA-II algorithm. They followed the guidelines of Harman et al. [22], which recommend the use of genetic operators focused on the problem to obtain the best performance. However, the authors did not provide an automated tool for the proposed strategies. Besides, their approach is neither specific for SPL nor architectural smells. Mariani and Vergilio [30] conducted a systematic review to provide an overview on existing search-based refactoring approaches. Their results show that most approaches are focused on removing code smells through source code refactoring. Our work, on the other hand, is focused in preventing the occurrence of architectural smells during the optimization of PLA design.

As discussed above, there is no tool that is able to detect and remove architectural smells from optimized PLA design solutions. Thus, in our previous work [28], we addressed this gap through the proposal and evaluation of a tool called OPLA-Tool-ASP. The first version of our tool applied the guidelines proposed in [35] to prevent the Unused Interface, Unused Brick and Concern

Overload smells. In comparison with the original version of OPLA-Tool [13], OPLA-Tool-ASP presented promising results for the three smells during the optimization of the PLA designs of two academic SPLs. Furthermore, another finding of our previous study is that the strategy to detect the Link Overload smell proposed in [18] need to be adapted to the SPL context. The threshold to detect Link Overload proposed by Garcia [18] count the inheritance relationships. However, inheritance is highly used to design variabilities in SPL Engineering, i.e. the variation point is modeled in a super class connected to the variants modeled in subclasses. Thus, counting inheritance implies in several false positives.

In this work, we extend the original version of OPLA-Tool-ASP to include the prevention of the Link Overload smell using the findings of our previous work [28]. Furthermore, we performed a new empirical study to evaluate the effectiveness of OPLA-Tool-ASP in preventing architectural smells during the PLA design optimization. In addition to the two academic SPL, we include a real-world SPL in the scope of the empirical study. We present details about the design and implementation of OPLA-Tool-ASP in the next section.

3 OPLA-Tool-ASP

This section presents OPLA-Tool-ASP – a tool proposed as an evolution of the OPLA-Tool with the addition of functionalities related to the detection and prevention of the introduction of architectural smells during the optimization of PLA design. Besides the original functionalities of OPLA-Tool, OPLA-Tool-ASP implements the guidelines, proposed by Perissato et al. [35], for the detection and prevention of the most frequent smells in optimized PLA designs, which are: Unused Interface, Unused Brick, Concern Overload, and Link Overload. Table 1 presents the definition of each of the aforementioned architectural smells.

To automate each of the smell prevention guidelines, we created a new version of the NSGA-II algorithm – called NSGAIIASP. This new implementation addresses the rules and constraints related to architectural smells. We also created a new fitness evaluation method to deal with the prevention of Link Overload smell. The next subsections present details about the prevention of each smell type in the OPLA-Tool-ASP ¹.

3.1 Prevention of Unused Interface and Unused Brick

Since an Unused Brick smell is a consequence of an Unused Interface smell, by preventing the latter we will also prevent the former. Following the strategy to identify both smells (Table 1), we included a constraint to consider invalid any solution that contains interfaces or classes without any relationship with other elements. For implementing such a constraint, we created a

method called `isValidSolution`, which is executed whenever a new solution is created and returns false when an isolated interface is identified.

The `isValidSolution` method is called in the loop in which the crossover and mutation operators are applied to generate offspring populations, as shown in lines 8 and 11 of Algorithm 1.

This method is also called during the creation of the initial population. Thus, none of the generated populations will contain solutions with the occurrence of Unused Interface and Unused Brick, as invalid solutions are not added to the new populations. Discarding invalid solutions is justified by the difficulty in correctly repairing a solution, as it is not straightforward to discover to which elements an isolated interface should be linked.

3.2 Prevention of Concern Overload

For detecting the Concern Overload smell, we implemented a method called `detectCO`, which is partially presented in Algorithm 2. As suggested in the detection strategy for Concern Overload (Table 1), all features associated with PLA classes and interfaces are considered as concerns in our implementation.

Lines 6 to 11 of Algorithm 2 are responsible for the detection of Concern Overload in the PLA classes. To achieve this goal, there is a loop that verifies whether the number of features associated with a given class is higher than the threshold, returning true when the condition is met. The same procedure is performed for the PLA interfaces between lines 12 and 17.

The `detectCO` method also calculates the threshold that is used to identify whether a class/interface is smelly or not. The threshold is the average number of concerns in elements of the PLA plus the standard deviation [18]. We omitted this part of the implementation from Algorithm 2 for the sake of simplicity.

The `detectCO` method was implemented in the `PLAFeatureMutation` class, which is responsible for implementing the MOA4PLA mutation operators. Among the implemented mutation operators there is the Feature-driven Operator, which provides the modularization of features and, as a consequence, can help to decrease the occurrence of Concern Overload.

As suggested in [35], the `detectCO` method is executed before selecting which mutation operator to apply. If the presence of Concern Overload is detected, the Feature-driven Operator is selected for application. Otherwise, the optimization returns to its normal flow, randomly selecting any mutation operator.

In the case of Concern Overload, we decided to try to repair the smelly solution instead of discarding it. We took this decision because there is a mutation operator that aims to improve the feature modularization, which can directly impact the occurrence of Concern Overload.

¹The experimental package is available at <https://github.com/optimizes/opla-tool-asp>.

Algorithm 1: Generation of the child population in NSGA-II

```

1 for i=0;i<(populationSize/2);i++ do
2   parentes[0]=(Solution)selectionOperator.execute (population);
3   parentes[1]=(Solution)selectionOperator.execute (population);
4   Object Execute=CrossoverOperator.execute (population);
5   if (execute instanceof Solution) then
6     Solution offspring=(Solution) crossoverOperator.execute(parents);
7     if (is ValidSolution((Architecture) offspring.getDecisionVariable()[0])) then
8       problem_.evaluateConstrains(offspring);
9       mutationOperator.execute(offspring);
10      if (is ValidSolution((Architecture) offspring.getDecisionVariable()[0])) then
11        problem_.evaluateConstrains(offspring);
12        problem_.evaluate(offspring);
13        offspringPopopulation.add(offspring);
14      end
15    end
16  end
17 end

```

Algorithm 2: Partial view of the implementation for the detectCO method

```

1 public boolean detectCO(Solution solution) throws JMException{
2   final Architecture arch = ((Architecture)solution.getDecisionVariables()[0]);
3   final List< Package >allPackage = new ArrayList< Package >(arch.getAllPackages());
4   if (!allPackage.isEmpty()) then
5     for Package selectedPackage: AllPackage do
6       List < Class >lstClass = new ArrayList<>(selectedPackage.getAllClasses());
7       for Class selectedClass: lstClass do
8         List< Concern >lstConcern = new ArrayList<>(selectedClass.getOwnConcerns());
9         if (lstConcern.size()>threshold) then
10          | return true;
11        end
12        List < Interface >lstInterface = new ArrayList<>(selectedPackage.
13        getAllInterfaces());
14        for Interface selectedInterface: lstInterface do
15          List< Concern >lstConcern = new ArrayList<>(selectedInterface.
16          getOwnConcerns());
17          if (lstConcern.size()>threshold) then
18            | return true;
19          end
20        end
21      end
22    end
23  end

```

3.3 Prevention of Link Overload

To detect Link Overload, Garcia [18] proposed an algorithm to define the threshold for each type of link. We considered as links the following relationships involving classes and interfaces: dependency, association, abstraction and realization. The definition of the threshold for each link type is based on the average number of links plus a standard deviation. The threshold is defined separately for each directionality (input, output or bidirectional).

Based on the results of our previous work [28] (see Section 2.4), we conducted a quantitative analysis to validate the guideline proposed by Perissato et al. [35] for Link Overload. We verified that the inheritance relationship between classes should not be considered in the detection of Link Overload, as this relationship is commonly used in PLA designs to model the variations of each variability. Thus, including such a relationship in the context of a SPL would result in false positives.

The first step for detecting Link Overload in our new implementation is early discovering and classifying relationships according to their directionality (input, output or bidirectional).

The dependency relationship is a relationship in which one element (the customer) depends on another element (the supplier). The realization relationship is a relationship between two elements, in which one element (the customer) performs the behavior that the other element (the supplier) specifies. The abstraction relationship is generally defined as a relationship between customer(s) and supplier(s) in which the customer (source subset) depends on the supplier (destination subset). The association relationship describes a link between classes. Through the multiplicity of the association, it is possible to determine that the instances of one class are linked to the instances of the other class.

For all types of relationships, the properties of the architectural elements were observed, since some elements had the same identification as the relationship with classes and interfaces. To determine the directionality of the relationship, first, the directionless relationships were considered to be bidirectional. Next, incoming links were considered for customers who had an identifier (id) equal to the relationship id. The remaining relationships that had a different id to that of the architectural element were considered outbound links. After counting all the relationships of each class and interface, we followed the formula indicated by Garcia [18] to specify the threshold for each directionality.

Once the thresholds has been calculated, the guideline proposed by Perissato et al. [35] recommends applying a penalty on the fitness of the solution, so that it is worse evaluated in the process of selecting the best designs to remain in the optimization process. Penalization is one of the strategies for treating invalid solutions in SBSE. In the case of Link Overload it is not feasible to change relationships to repair the smell. On the other hand, discarding such solutions could lead to loss of diversity in the population. Thus, the penalty makes

it difficult for the solution to survive, but even if it survives it can be repaired by search operators in future generations.

The next step to implement the Link Overload detection was to create a new fitness evaluation method. This new method is based on an existing method called evaluate, which already existed in the OPLA-Tool implementation. The evaluate method calculates the fitness of the solution and stores it in a list. The difference of our new implementation is that it also verifies which are the solutions that exceed the Link Overload threshold and applies a penalty for these solutions. Algorithm 3 presents the partial view of the new method, which is called evaluateLinkOverload, with respect to the fitness penalization. Some parts of the method were hidden due to simplicity.

The new fitness evaluation method, presented in Algorithm 3, called evaluateLinkOverload, initially checks which objective functions were selected for optimization (stored in the selectedMetrics variable), then a loop runs through the entire list of solutions (line 3) to penalize the solutions that exceeded the threshold (Table 3). The value of the penalty by objective function is calculated in line 7, which is the current fitness value multiplied by 1000. This value was empirically defined. The new fitness value is defined in line 8, where the current fitness value is added to the weight multiplied by the number of smell violations (returned by the getExceedLink method). The objective function evaluateLinkOverload, is executed after the application of the crossover and mutation operators of the NSGA-II algorithm. In this sense, penalties are applied for all solutions that exceed the threshold in the same proportion of the smelly elements (classes and interfaces).

4 Empirical Study Definition

The goal of our empirical study, following the GQM approach [4], is: Analyze the automatically generated PLA design solutions for the purpose of evaluating with respect to the effectiveness of OPLA-Tool-ASP in preventing the architectural smells: Unused Interface, Unused Brick, Concern Overload and Link Overload from the point of view of the architect in the context of the search-based PLA design. To accomplish such a goal, we defined the following question:

RQ - Is OPLA-Tool-ASP effective to detect and prevent Unused Interface, Unused Brick, Concern Overload and Link Overload?.

To answer the RQ, we used three metrics: (i) the fitness of the solutions (alternative designs) generated by OPLA-Tool and OPLA-Tool-ASP, composed by the three objective functions which are explained below, (ii) the number of occurrences of architectural smells in the alternative designs, and (iii) the number of smelly architectural elements.

Subject PLA Designs. Three PLA designs were used in the empirical study: Arcade Game Maker (AGM) [37], Mobile Media (MM) [10] and BET [12]. AGM was cre-

Algorithm 3: Partial view of the implementation for the evaluateLinkOverload method

```

1 public void evaluateLinkOverload(Solution solution)
2 List< Fitness > fitnesses = new ArrayList<>();
3 for (String selectedMetric : selectedMetrics) do
4     ObjectiveFunctions metric = ObjectiveFunctions.valueOf(selectedMetric);
5     fitnesses.add(new Fitness(metric.evaluate((Architecture) solution.getDecisionVariables()[0])));
6     for (int i = 0; i <= fitnesses.size(); i++) do
7         double weight = fitnesses.get(i).getValue()*1000;
8         solution.setObjective(i, fitnesses.get(i).getValue() + (weight * ((Architecture)solution.
9             getDecisionVariables()[0]).getExceedLink()));
9     end
10 end

```

ated by the Software Engineering Institute (SEI). It is composed of three arcade games: Brickles, Bowling and Pong. MM is a mobile application composed of features that handle with music, video, and photo for portable devices [42]. BET [12] is a SPL for public transport bus service management. It offers features such as the use of an electronic card for transport payment, automatic tollgate opening, and unified traveling payment. Table 2 presents the architectural elements numbers of the PLA designs.

Experiment Configuration. The study encompasses two experiments, named OPLA-Tool and OPLA-Tool-ASP. The first experiment is the baseline because it uses the current version of OPLA-Tool [17] to optimize PLA designs, which does not apply any guidelines to prevent architectural smells. In the second experiment, the PLA designs were optimized by OPLA-Tool-ASP, which contains guidelines to identify and prevent the four aforementioned smells.

Search-Based Algorithm and Parameter Settings. Both experiments (OPLA-Tool and OPLA-Tool-ASP) used NSGA-II [11]. We chose NSGA-II because it has been successfully used in many previous works [8]. We used the same parameter settings for both experiments for all PLA designs. NSGA-II was set with a population size of 100 individuals and 30,000 fitness evaluations (300 generations), which was the stopping criterion. The objective functions were: Relational Cohesion (COE), Class Coupling (AClass) and Feature Modularization (FM) [38]. All mutation operators were applied under the mutation rate equal to 0.8. The crossover operators called Feature-Driven Crossover [36] and Complementary Crossover [36] were applied with a crossover rate of 0.4. This parameter setting is the same adopted in [36], after an experimental parameter calibration of OPLA-Tool with the goal of identifying the best configuration for its parameters. We executed 30 independent runs for each PLA design, as recommended by Arcuri and Fraser in [2].

Architectural Smells Screening. The threshold values were obtained by applying the detection strategy of each architectural smell, presented in Table 1, over the original PLA designs provided as input to both tools. So, the same threshold values were used to verify each non-dominated solution. Table 3 presents the threshold values adopted for each architectural smell defined according to the detection strategies presented in Ta-

ble 1. These threshold values were used by both tools during the optimization process in order to prevent the architectural smells manifestation. We decided to use the original PLA design as a baseline because: (i) it is expected that during the optimization process the obtained solutions will be better than the original design, so this one would be the worst case, and (ii) calculating the threshold for each obtained solution represents additional computational cost what impacts on the runtime.

Fractional numbers resulting from the threshold calculus have been rounded up. None threshold was defined for Unused Interface and Unused Brick since the original designs did not contain occurrences of these smells. So, every occurrence of these smells detected in the design was marked as a smelly element.

After the 30 runs of each experiment for the three PLA designs, we selected the non-dominated solutions generated by OPLA-Tool and OPLA-Tool-ASP. Then, each non-dominated solution was inspected in order to identify the remaining architectural smells. To do so, the first author visually inspected the 81 non-dominated PLA designs generated by both tools observing and counting the manifestation of Unused Interface and Unused Brick. Such an inspection had the goal of certifying that both tools did not generate solutions with these smells. The number of occurrences of the Concern Overload, Large Class and Link Overload smells were counted automatically. The automation of this process was validated before the study's conduction. During the architectural smells screening after the execution of the experiments, the threshold values presented in Table 3 were also considered.

Analysis of the Experiment Results. The results obtained by both experiments were compared in terms of number of the occurrences of architectural smells, fitness values and analysis of the excerpts of the design that contain smelly elements.

5 Results and Analysis

Table 4 presents the number of non-dominated solutions found by both experiments in the context of our empirical study. A total of 81 non-dominated solutions were found after 30 independent runs for the three PLA designs. OPLA-Tool-ASP generated fewer solutions than

OPLA-Tool. The entire set of non-dominated solutions is used in the analysis presented in the next sections.

5.1 Fitness of the Generated Solutions

Figures 1, 2 and 3 present the solutions on the search space, considering their fitness values, for AGM, MM and BET, respectively. For all figures, the blue line represents solutions found by OPLA-Tool, the red line represents solutions found by OPLA-Tool-ASP and the original design is depicted in green. For all graphs, the objective function FM (Feature Modularization) is presented in the Y axis. The function COE (Relational Cohesion) is presented in the X axis of the graph on the left whereas the function ACLASS (Class Coupling) is in the X axis of the graph on the right.

These graphs allow the comparison among the fitness values of the solutions generated by OPLA-Tool and OPLA-Tool-ASP and the original PLA design. The lower the values, the better the results, since we want to minimize all objective functions. It is possible to notice that the solutions found by OPLA-Tool-ASP and OPLA-Tool are better than their respective original design with respect to FM function. Hence, all obtained solutions have better feature modularization than the original design.

With regards to cohesion (COE function), the entire set of solutions found by OPLA-Tool-ASP for AGM and MM has better values of cohesion than the original design as can be seen in the graphs on the left side of Figures 1 and 2. The majority of solutions found by OPLA-Tool for AGM and MM has also better cohesion than the original design (100% of solutions for AGM and 90% of solutions for MM).

On the other hand, 40% of solutions found for AGM and 50% of solutions found for MM by both experiments has worse class coupling (ACLASS function), as can be seen in the graphs on the right side. However, there are solutions with better coupling than the original design. Particularly, 57.5% of solutions found by OPLA-Tool have better coupling than the original design for AGM and MM.

Considering the BET PLA design, all solutions generated by OPLA-Tool-ASP and OPLA-Tool have better coupling than the original design (Figure 3). Conversely, only 4 (out 19) solutions found by OPLA-Tool have lower values of COE than the original design. None solution generated by OPLA-Tool-ASP has better value of COE than the original one. Hence, it is clear that coupling and cohesion are measured by conflicting objective functions, i.e., when the values of COE increase, often the values of ACLASS decrease and vice-versa. The parallel coordinate graphs depicted in Figures 4, 5 and 6 support this finding. In these graphs, each objective function is represented by a coordinate and each line represents a solution. For instance, the red line in Figure 4a depicts one solution whose fitness values are COE= 15, ACLASS=7 and FM=270. Another example is the solution represented by the yellow line in Figure 6a that has the following fitness values: COE=203,

ACLASS=21 and FM= 825.

Another interesting observation is that the maximum fitness values of solutions achieved by OPLA-Tool-ASP are lower than the values of solutions generated by OPLA-Tool for AGM and MM (Figures 4 and 5), what means that OPLA-Tool-ASP obtained solutions with lower fitness values than some solutions of OPLA-Tool, as can also be noticed in Figures 1 and 2. Hence, OPLA-Tool-ASP has potential to optimize the fitness of the generated solutions as much as OPLA-Tool. A similar behavior is observed in Figures 6 and 3 for BET regarding coupling and cohesion. On the other hand, OPLA-Tool achieved solutions with better feature modularization than OPLA-Tool-ASP for BET.

Finding #1: Both experiments achieved solutions with better fitness than the original design. Feature Modularization is the architectural property most benefited by both experiments. Several solutions found by OPLA-Tool-ASP are better than solutions found by OPLA-Tool in terms of fitness.

5.2 Occurrences of Architectural Smells

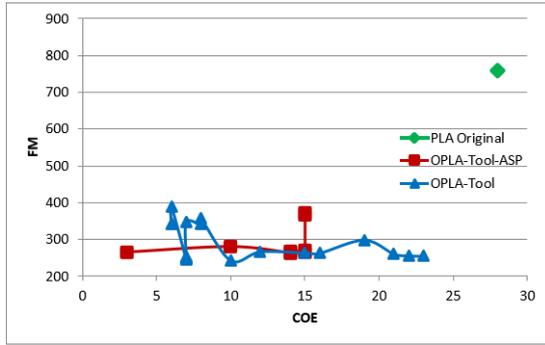
So important as the fitness of the solutions is the presence of architectural smells in the alternative designs generated automatically. In this sense, Tables 5 and 6 present the number of occurrences of each kind of architectural smell per solution obtained by OPLA-Tool-ASP and OPLA-Tool for AGM, MM and BET, as well as the number of occurrences of smells in the original design of these PLAs. Each solution is identified by S[n], where n is the number of the obtained solution. Each cell contains the number of occurrences of each smell in that solution.

The numbers reported to Unused Interface are related to the counting of interfaces without any relationship with another architectural element. For Unused Brick, we counted packages whose interfaces have no relationship. For Concern Overload, classes and interfaces assigned to more concerns (SPL features) than the threshold value were considered smelly (Table 3). Similarly, classes and interfaces that exceed the threshold of at least one type of relationship (input, output or bidirectional) were considered smelly for the Link Overload smell.

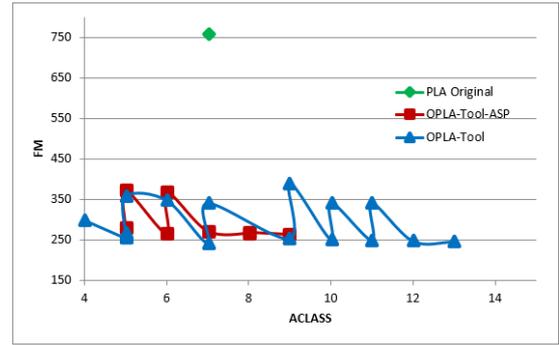
5.2.1 Unused Interface and Unused Brick

It is possible to notice in Table 5 that Unused Interface and Unused Brick are not present neither in the original designs nor in the solutions generated by OPLA-Tool-ASP. On the other hand, occurrences of these architectural smells were found in several solutions obtained by OPLA-Tool (Table 6).

All solutions obtained by OPLA-Tool for AGM contain at least one of these two smells, totalizing 121 occurrences of Unused Interface and 33 occurrences of Un-

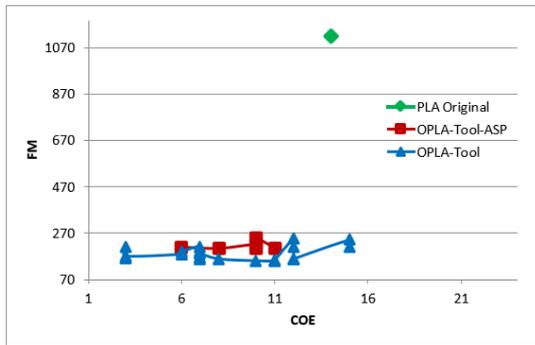


(a) Objective Functions: FM x COE

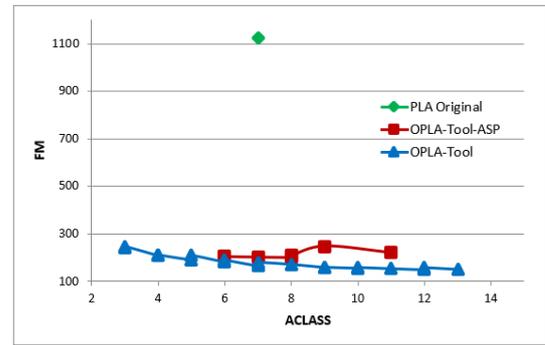


(b) Objective Functions: FM x ACLASS

Figure 1. Solutions found for AGM.

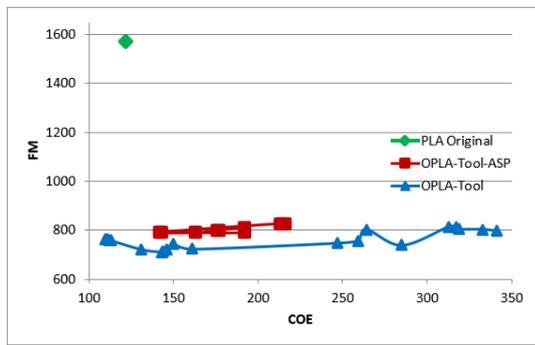


(a) Objective Functions: FM x COE

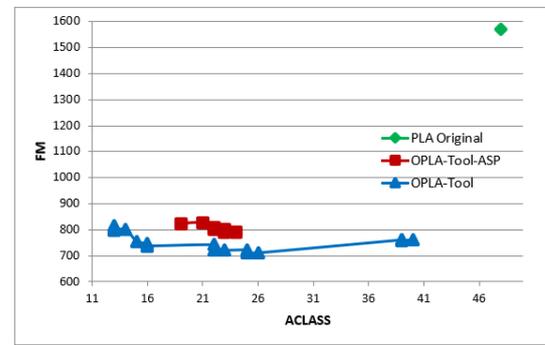


(b) Objective Functions: FM x ACLASS

Figure 2. Solutions found for MM.

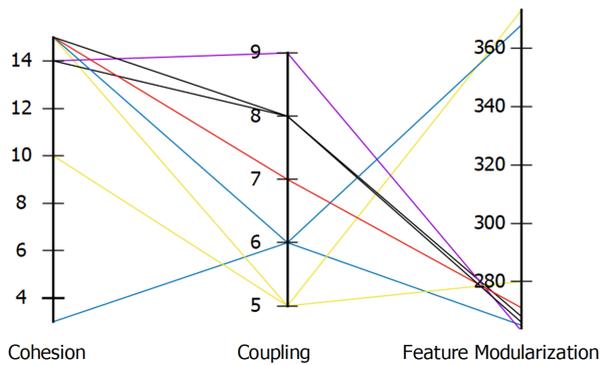


(a) Objective Functions: FM x COE

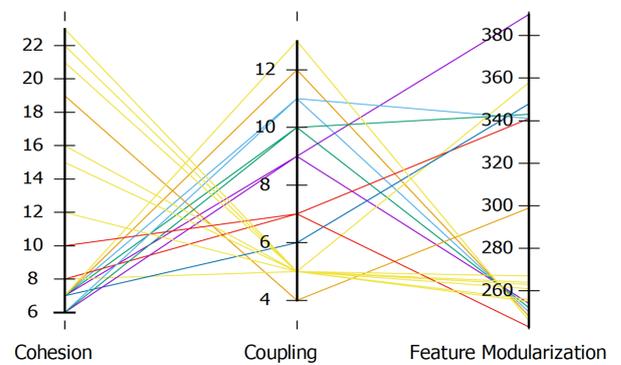


(b) Objective Functions: FM x ACLASS

Figure 3. Solutions found for BET.



(a) OPLA-Tool-ASP



(b) OPLA-Tool

Figure 4. Parallel Coordinates of Solutions Found for AGM.

used Brick. For MM and BET only two solutions contain these smells. We observed that the smelly elements

are usually new packages created by mutation operators, which contain interfaces that are not connected

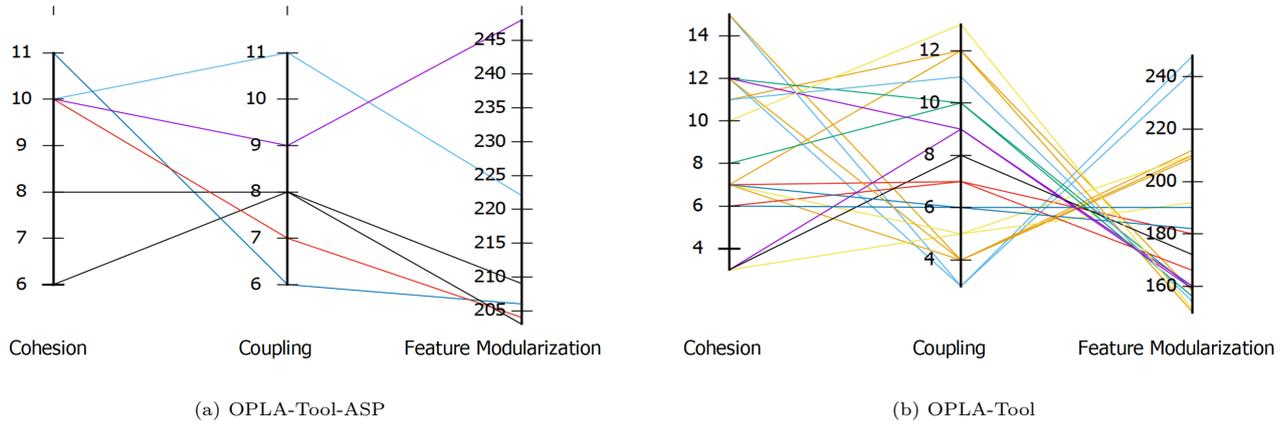


Figure 5. Parallel Coordinates of Solutions Found for MM.

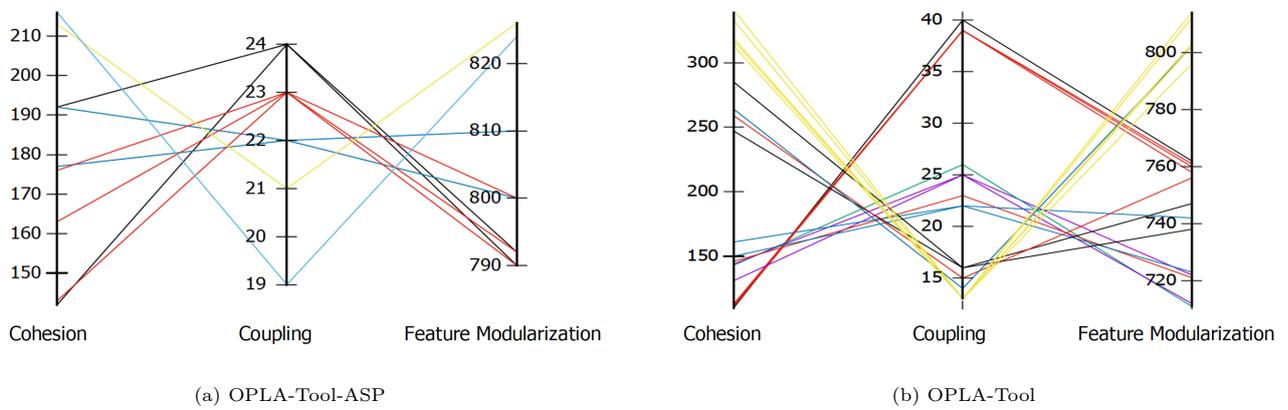


Figure 6. Parallel Coordinates of Solutions Found for BET.

with other elements in the design. These new packages are usually created to modularize a single feature in order to improve feature modularization, decreasing the FM objective function. However, some unknown action introduces at least one of these smells.

The obtained results indicate the efficacy of the guidelines implemented in OPLA-Tool-ASP to prevent Unused Interface and Unused Brick as the `isValidSolution` method (Section 3.1) discards every solution that contains unused interfaces. In addition, due to the action of search operators, OPLA-Tool-ASP obtained solutions whose fitness values of the FM objective function are almost as low as the solutions obtained by the OPLA-Tool.

Finding #2: OPLA-Tool-ASP generated solutions without introducing the Unused Interface and Unused Brick architectural smells.

5.2.2 Concern Overload

With regards to the Concern Overload architectural smell, both experiments achieved equivalent performance for AGM maintaining the number of occurrences of this smell as in the original PLA design (Table 5). In all solutions in which there are occurrences of Concern Overload the smelly classes are Puck and Sprite (Figure 7). It is possible to notice that the Puck class has

stereotypes related to 7 different features, what means that it has attributes and methods to realize the features `<< play >>`, `<< save >>`, `<< movement >>`, `<< collision >>`, `<< bowling >>`, `<< brickles >>` and `<< pong >>`. The Sprite class is also smelly since it realizes 5 features: `<< play >>`, `<< collision >>`, `<< bowling >>`, `<< brickles >>` and `<< pong >>`. The threshold of Concern Overload for AGM is 2 features (Table 3).

Features are well-modularized in the original design of AGM [36]. Hence, there are a few smelly classes in the design. This fact justifies the equivalent performance of both experiments, since the effects of Feature-driven Operator, which improves feature modularization, act over a randomly selected class.

For MM and BET, both experiments have decreased the number of occurrences of this smell in the obtained solutions. OPLA-Tool-ASP had better performance for BET whereas OPLA-Tool had better performance for MM.

The original design of MM has 4 classes with Concern Overload. The solutions generated by OPLA-Tool-ASP have 2, on average, smelly classes whereas several solutions obtained by OPLA-Tool usually have 1 smelly class. In the original design, the smelly elements are three classes (Media, MediaCtrl and MediaMgr) and one interface (IMediaMgt). In the solutions generated by OPLA-Tool-ASP, IMediaMgt and MediaCtrl are not

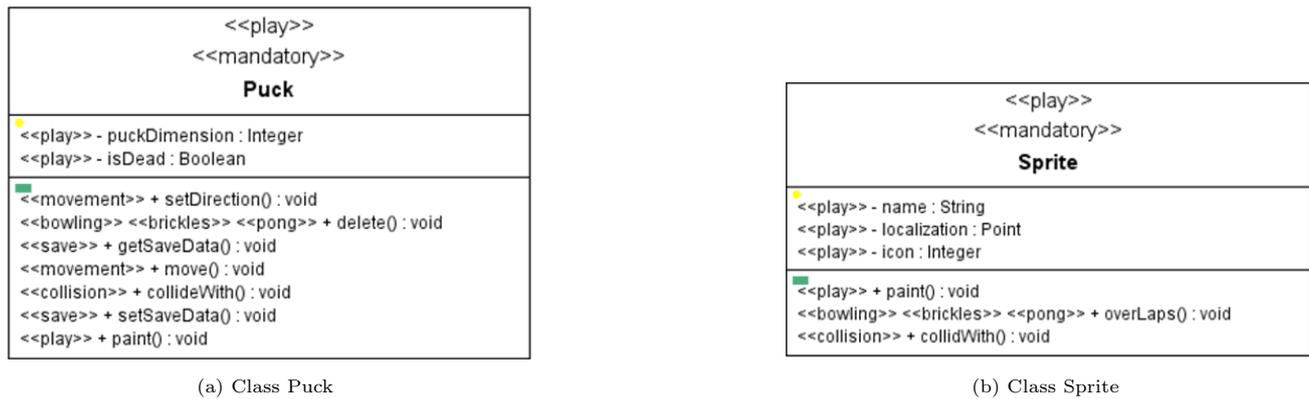


Figure 7. Classes with Concern Overload in AGM Original Design.

smelly elements. In the solution S1, MediaMgr is not smelly too. The Media class is the smelly element of the solutions obtained by OPLA-Tool-ASP and OPLA-Tool where there is a single smelly element. This class realizes 6 features, exceeding the threshold of 5 features (Table 3).

The original design of BET has 7 smelly classes, exceeding the threshold of 3 features (Table 3). The solutions obtained by OPLA-Tool have 5.4 smelly classes on average, whereas the solutions of OPLA-Tool-ASP have 5.0 smelly classes on average, reducing 29% of these smell occurrences compared to the original design. In the original design, the smelly elements are the interface ICartaoMgt and the classes CargaCartaoLimPassagensEmpresaUsuarua, CartaoMgr, CartaoPagamentoCartaoCtrl, ViagemCtrlTempoNumViagens, ViagemIntegracaoCtrl and GerenciaSistViarioNumCartoesTempoNumViagem. However, CartaoPagamentoCartaoCtrl and ICartaoMgt are not smelly elements in the solutions obtained by OPLA-Tool-ASP.

For the sake of illustration, Figure 8-a presents interface ICartaoMgt extracted from the original design of BET. ICartaoMgt manifests Concern Overload since it realizes 4 features, exceeding the threshold. The features realized by this interface are << pagamentocartao >>, << adicional >>, << tipopassageiro >> and << acabasico >>. Figure 8-b illustrates an excerpt of the solution S1 of OPLA-Tool-ASP, where ICartaoMgt is not a smelly interface since it realizes 3 concerns. Interface7419 was created by the Feature-driven Operator in order to modularize the feature << adicional >>, reducing the number of features realized by ICartaoMgt. On the other hand, we can observe that ICartaoMgt of solution S16 generated by OPLA-Tool (Figure 8-c) is equal to the original design, manifesting the architectural smell.

We observed that OPLA-Tool-ASP presented satisfactory results regarding Concern Overload due to the decreased number of features realized by classes/interfaces of the obtained solutions. This decrease contributed to generating alternative designs with fewer smelly classes/interfaces. Thus, the guideline proposed by Perissato et al.[35] and implemented in OPLA-Tool-ASP is able to detect and reduce the

occurrences of Concern Overload in the generated alternative designs. However, we infer that the search algorithm would need to evolve for more generations to optimize the FM function even more and then generate solutions without occurrences of Concern Overload or with a lower number of smelly elements. Even though, it might be impossible to reduce to zero the occurrences of Concern Overload given certain characteristics of the original design, such as the existence of features that are crosscutting and/or tangled by nature.

Finding #3: The guideline implemented in OPLA-Tool-ASP contributes to the reduction and prevention of the Concern Overload architectural smell.

5.2.3 Link Overload

OPLA-Tool-ASP achieved excellent results for AGM with respect to the Link Overload architectural smell, decreasing the mean of occurrences of this smell in the obtained solutions. On the other hand, OPLA-Tool achieved solutions with twice the occurrences of this smell when compared to the original PLA, as presented in Tables 5 and 6.

Figure 9-a presents an excerpt of the original design including the GameBoardCtrl and its interfaces. The threshold for AGM is 2 input links, 2 output links and 1 bidirectional link. Thus, GameBoardCtrl is considered a smelly class since it realizes 3 output links. Figure 9-b presents the equivalent excerpt for the solution S7 of OPLA-Tool-ASP, where the methods related to the feature << play >> of IGameBoardData and ICheckScore were moved to the IGameMgt interface, which is implemented by GameCtrl. This action is typical of the behavior of the Feature-driven Operator to improve feature modularization. In this way, GameBoardCtrl is not a smelly class because it realizes only 1 output link. Figure 9-c presents part of the solution S1 generated by OPLA-Tool. In addition to the interfaces that GameBoardCtrl realizes in the original design, this class is also realizing Interface28728 of Package64435Ctrl, a smelly class.

Taking into account the threshold values for MM re-

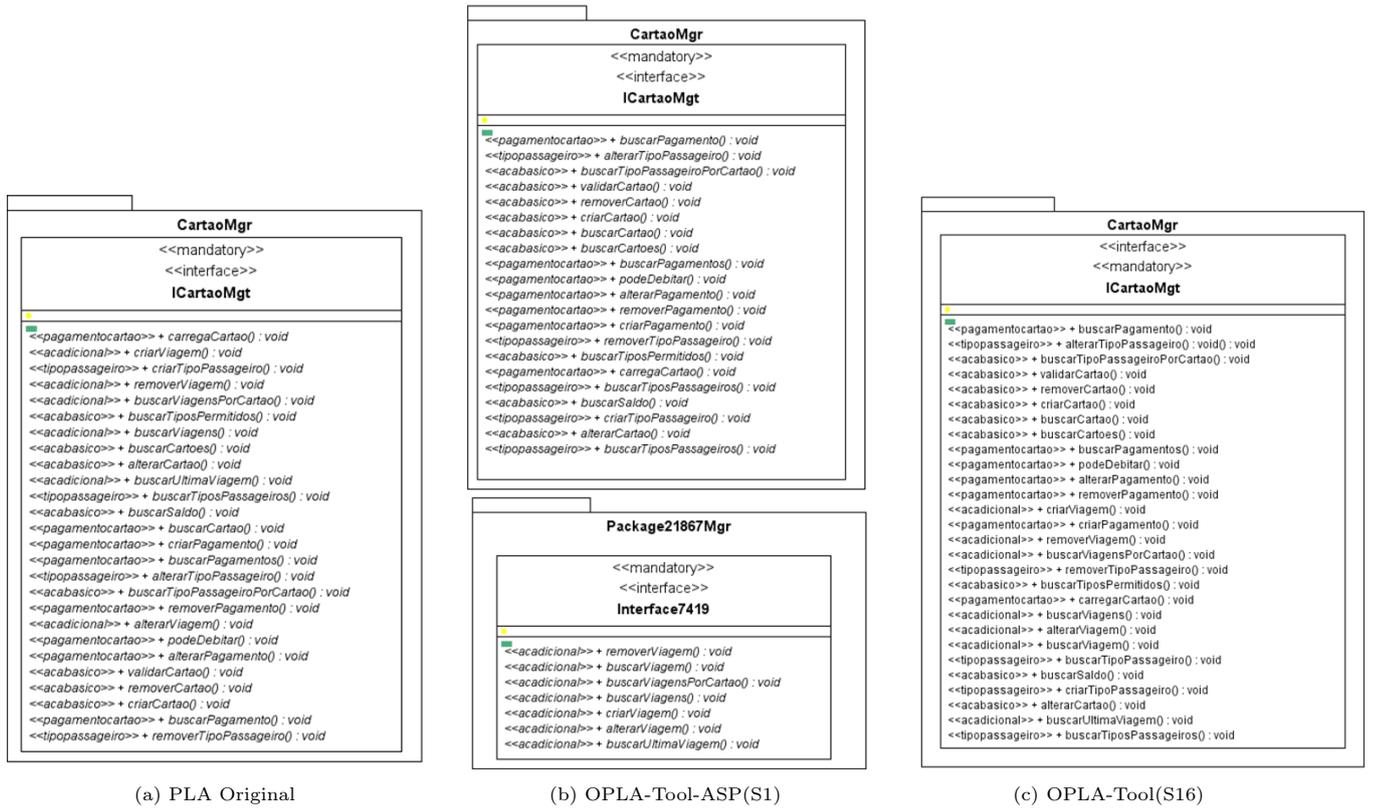


Figure 8. Example of reduction of Concern Overload in solution of BET.

garding the Link Overload smell (Table 3), the smelly classes in the original design are Media with 4 bidirectional links, MediaCtrl with 7 output links, and MediaGUI with 7 input links. The results obtained by OPLA-Tool-ASP were similar to the original design. However, the results of OPLA-Tool were significantly worse than the OPLA-Tool-ASP ones. This is probably due to the Feature-driven Operator action that creates new packages and classes to improve feature modularization, increasing the class coupling.

For BET, solutions generated by OPLA-Tool-ASP has significantly fewer occurrences of Link Overload than the original design whereas the results of OPLA-Tool were similar to the original PLA, as can be observed in Tables 5 and 6.

Figure 10-a illustrates the original design of the ViagemTempoLinhaIntegradaCtrl class of BET. This class was considered smelly because it implements 5 interfaces, overcoming the threshold of output links that is 4 (Table 3). On the other hand, in the solution S8 of OPLA-Tool-ASP, ViagemTempoLinhaIntegradaCtrl is not smelly since it realizes only 3 interfaces. In S8, the ICartaoMgt interface is implemented by GerenciaCtrl. The IRegistrarArrecadacao interface does not exist and its operations are now realized by a new interface assigned to `<< bet >>` feature: Interface8992. The new interface is realized by AquisicaoCartao inside Package26782GUI, which was created to modularize the referred feature. Figure 10-c presents the equivalent excerpt of solution S2 generated by OPLA-Tool. In this solution, ViagemTempoLinhaIntegradaCtrl implements 4 interfaces (output links) and has 1 bidirectional

link. So it does not overcome the threshold. Although ViagemTempoLinhaIntegradaCtrl is not a smelly class in both solutions, in S8 of OPLA-Tool-ASP this class has fewer responsibilities than in S2 of OPLA-Tool, what is considered better.

Taking into account the mean number of the Link Overload occurrences for BET presented in Tables 5 and 6, it is noticeable that the solutions of OPLA-Tool-ASP have around 12 smelly classes – 60% less occurrences than the original design, whereas the solutions obtained by OPLA-Tool have equal or higher number of occurrences than the original design.

The excellent results point out that this difference between the performance of OPLA-Tool-ASP and OPLA-Tool with respect to Link Overload is due to the guideline to prevent this smell, which penalizes the fitness of smelly solutions as presented in Section 3.3. In this way, solutions with worse fitness have less chance of being selected for the next generation of the search process. It is worth highlighting that every solution generated by OPLA-Tool-ASP that contained classes or interfaces surpassing the threshold value was penalized depending on the number of elements that exceeded this value.

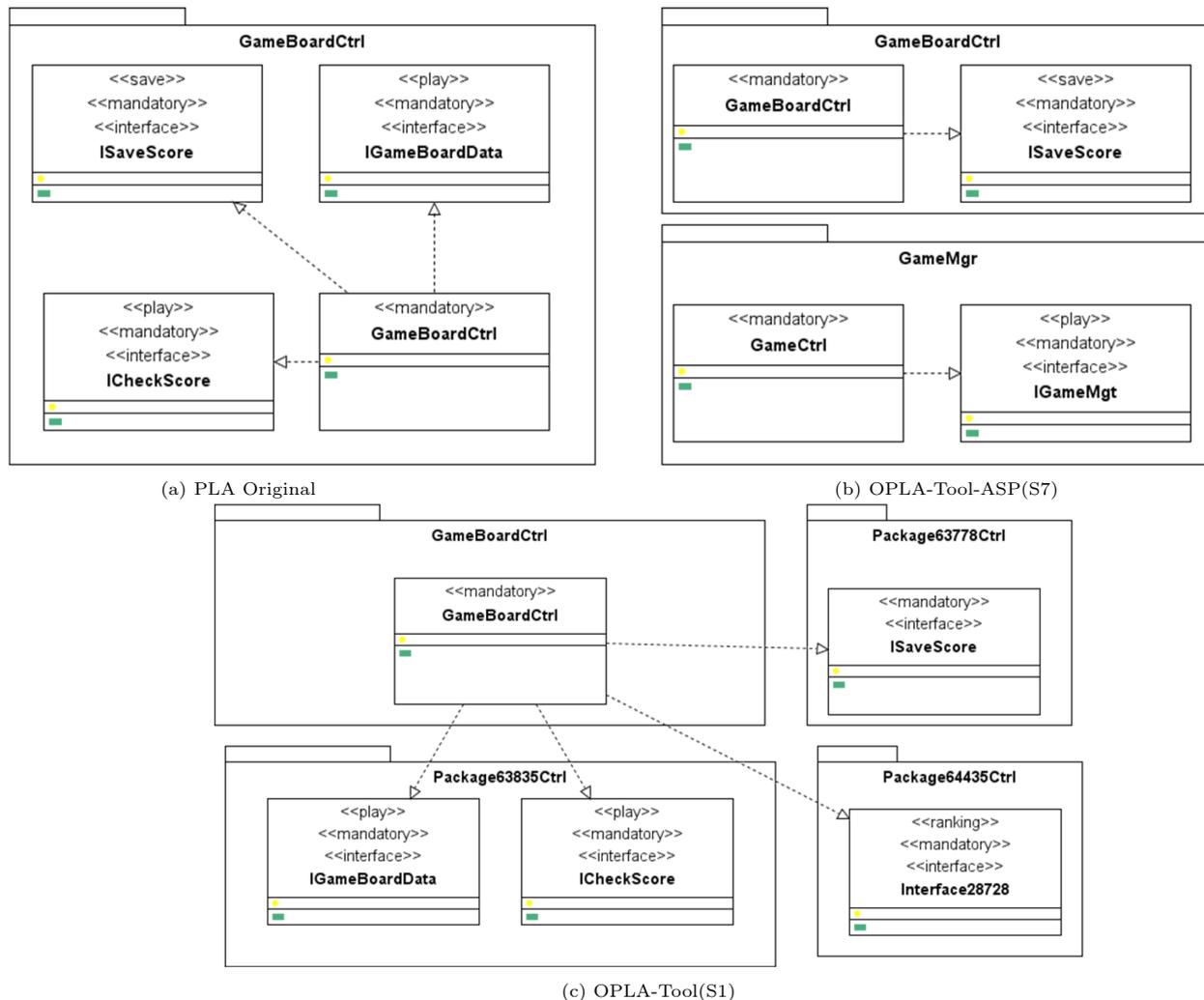


Figure 9. Occurrences of Link Overload in solution of AGM.

Finding #4: The penalty applied in the fitness of the solutions together with the feature modularization improved by the Feature-driven Operator were essential to the excellent results achieved by OPLA-Tool-ASP for the three PLAs, indicating that the guideline to prevent Link Overload implemented in our tool is effective.

5.3 Answering the Research Question

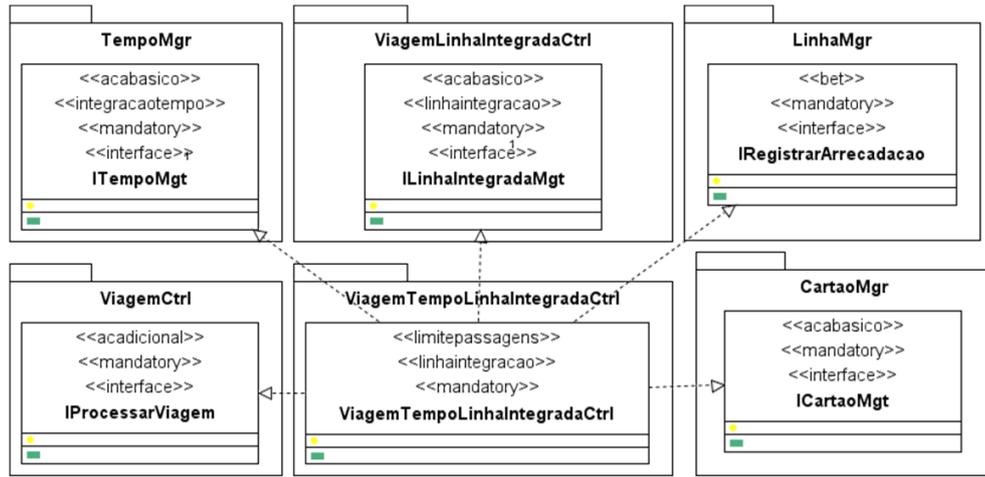
Another point to be observed before answering the research question is the total number of smelly elements of each solution. In this sense, Table 7 presents the number of smelly elements per solution taking into account the smells: Unused Interface, Unused Brick, Concern Overload and Link Overload. Numbers related to the original design of AGM, MM and BET are presented in the column labeled as Original. The numbers related to the solutions found by OPLA-Tool-ASP are presented in the remaining columns. Table 8 presents the number of smelly elements per solution found by OPLA-Tool.

OPLA-Tool-ASP found a lower number of solutions than OPLA-Tool and sometimes its solutions have

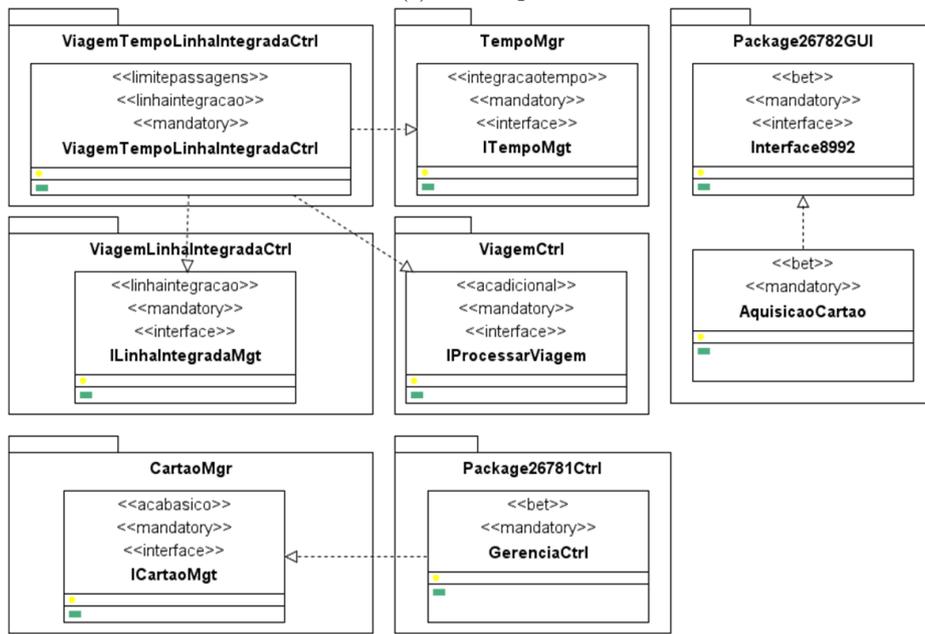
worse fitness than some solutions of OPLA-Tool. However, its solutions contain fewer smelly elements than solutions of OPLA-Tool considering the four architectural smells that were mostly found in the solutions generated by OPLA-Tool in the study of Perissato et al.[35]. OPLA-Tool-ASP also satisfactorily optimizes the objective functions during the evolutionary process as shown in Section 5.1.

However, we observe that there are opportunities to improve our tool. As the results of preventing Concern Overload were not so significant, it is worth investigating whether better results can be reached by using a greater number of generations during the search process.

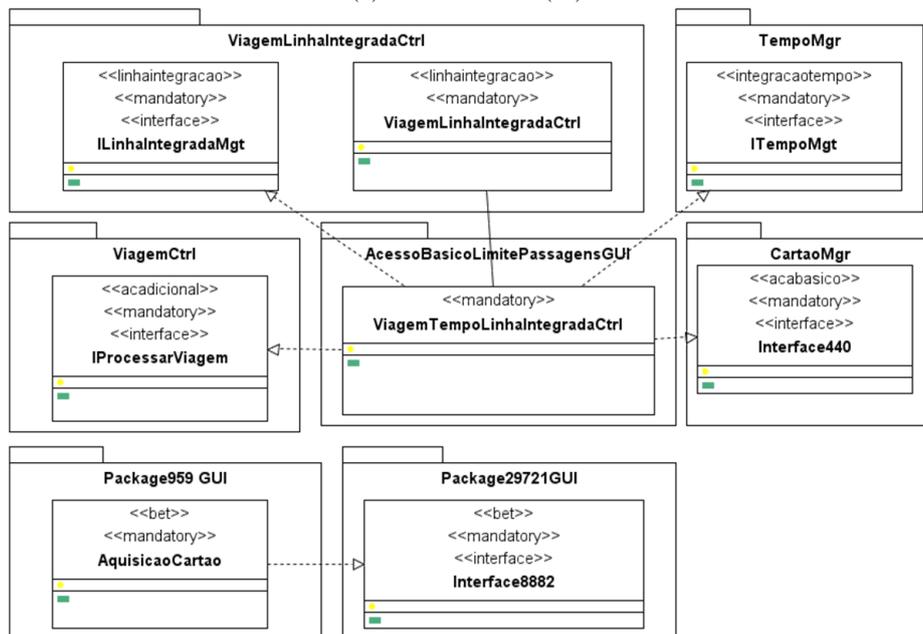
Results regarding to Unused Interface, Unused Brick and Link Overload presented effective results. No occurrence of Unused Interface and Unused Brick was detected in the solutions found by our tool. The penalty applied to the fitness of solutions that exceed the threshold of Link Overload was also effective in achieving satisfactory results for the three PLA designs.



(a) PLA Original



(b) OPLA-Tool-ASP(S8)



(c) OPLA-Tool(S2)

Figure 10. Occurrences of Link Overload in solution of BET.

Answering RQ: By analyzing quantitative and qualitative results we can state that OPLA-Tool-ASP is effective to optimize the PLA design and to simultaneously prevent or reduce the smells Unused Interface, Unused Brick, Concern Overload and Link Overload.

5.4 Discussion

In this section we discuss the main contributions of this work against related work.

The results of the systematic mapping (Section 2.4) pointed out a research gap regarding the automation of strategies to detect and prevent architectural smells in PLA design. Such a gap reinforces the relevance of the present work to the PLA design optimization using SBSE techniques.

In this sense, our work increases state of the art by providing OPLA-Tool-ASP, which aims to detect and prevent the generation of PLA design alternatives that contain the following architectural smells: Unused Interface, Unused Brick, Concern Overload and Link Overload.

The aforementioned empirical results attest the effectiveness of OPLA-Tool-ASP in detecting and preventing these kinds of architectural smells in several obtained solutions.

Our work serves as a starting point to the application of guidelines to detect architectural smells, originally proposed to software design, to the context of PLA design. There is room for improvements in this sense, however the results are promising.

The results also corroborate that the PLA design optimization should be solved using multi-objective algorithms, since COE and ACLASS objective functions are in conflict, following the recommendation of Coello et al. [7] of employing multi-objective algorithms to simultaneously optimize competing factors.

Our study also contributes to ascertaining the suitability of the guidelines proposed in [35], which were implemented in OPLA-Tool-ASP.

6 Threats to Validity

In this section we discuss the possible limitation of our study, and how we mitigate them, based on the main types of threats to validity described by Wohlin et al. [40]: internal, external, construct and conclusion.

The threats to the internal validity are related to the experiment's settings. We used as baseline the state-of-the-art tool, namely OPLA-Tool, as it is the tool most related to ours. The PLA designs used in the experiments were extensively used in other studies [10, 12, 14, 23]. We used the most used search-based algorithm [8], which was configured according to previous works [5, 32, 36]. During the architectural smells screening, to compare the obtained results, the visual inspection searching for Unused Interface and Unused

Brick was performed by the first author and was checked by the second author.

Another threat concerns the way of the threshold values were calculated. We adopted the calculus-based on average plus standard deviation proposed by Garcia [18]. However, we did not analyze if the objective functions follow a normal distribution, otherwise, the threshold might not be a good value. We softened such a threat during the visual inspection when the authors have observed the entire model with the goal of validating the threshold values adopted by OPLA-Tool-ASP, and no value seemed to be inappropriate.

External threats to validity are related to the set of PLA designs used in the study. Our work is limited on the number of subject PLAs, which can impact the generalization of the results. To soften this threat, we used SPL of different domains, with different sizes, designed by different architects and widely used in related work. The obtained solutions were generated from two academic SPL (AGM, MM) and one real SPL (BET). However, it was possible to observe the occurrence of different kinds of architectural smells in the obtained solutions.

The construct validity is related to the configuration of our empirical study. All the choices and definitions regarding the empirical study rely on existing literature. The subject PLAs are well-known, the baseline is the state-of-the-art tool, NSGA-II is widely used for multi-objective optimization. The analysis of the results were similar to other studies in the same research topic, such as our previous study [28]. Also, one of the authors visually inspected the 81 solutions obtained by both experiments with respect to Unused Interface and Unused Brick. To mitigate this threat, another author has verified the identified occurrences. The identification of occurrences regarding Concern Overload and Link Overload was automatic to reduce the human effort and the error-proneness. This source code was properly tested before the empirical study conduction.

The main conclusion validity is the number of PLA designs (3) and the number of instances of design analyzed during the empirical studies (81). In comparison with our previous study [28], the current study involved the increase of 120% additional instances and a real SPL. Despite the results of our study cannot be generalized, it was possible to identify differences between the original PLA designs and the obtained solutions, which allowed evaluating the impact of OPLA-Tool-ASP on the prevention of architectural smells as well as the comparison between the results generated by OPLA-Tool-ASP and the current version of OPLA-Tool, which did not employ guidelines to detect and prevent architectural smells.

7 Concluding Remarks

This work presented OPLA-Tool-ASP, a tool whose goal is automatically to generate PLA design alternatives that optimize the objectives selected by a software

engineer while preventing the existence of some architectural smells in the alternative designs. OPLA-Tool-ASP includes the original functionalities of OPLA-Tool and applies guidelines originally proposed by Perissato et al. [35] to prevent architectural smells. The tool includes guidelines to prevent the four architectural smells: Unused Interface, Unused Brick, Concern Overload and Link Overload.

Quantitative and qualitative empirical results pointed out the solutions (alternative designs) generated by OPLA-Tool-ASP present less smelly elements and with better fitness values than the solutions generated by the current version of OPLA-Tool. The objectives optimized during the empirical study involved feature modularization, class coupling and cohesion, which are important architectural properties for PLA design.

The main contributions of this work are: (a) an evolution to state of the art in PLA design optimization using search-based algorithms; (b) the empirical results that indicate the effectiveness of OPLA-Tool-ASP to prevent the aforementioned architectural smells; and, (c) our empirical study contributes to ascertaining the suitability of the guidelines proposed in [35], which were implemented in OPLA-Tool-ASP.

In further works we intend to perform a qualitative evaluation of the solutions generated by OPLA-Tool-ASP with experts aiming at identifying improvement opportunities of our tool. We also intend to include the human interaction during the search process to evaluate an acceptable element, giving to her the option of labeling an element as not smelly even in cases where the element exceeds the threshold.

Another future work is adding the prevention of other smells, such as Connector Envy, Dependency Cicle and Sloppy Delegation. We postponed the inclusion of these smells in OPLA-Tool-ASP because they are not frequent in solutions generated by OPLA-Tool and their detection depend on the interaction with a software architect during the search process. The evaluation of other ways to obtain the threshold values, such as the methodology for deriving software metric thresholds proposed in [1], is also an interesting work to be done.

Acknowledgements

This research is supported by CNPq Grant 428994/2018-0.

References

- [1] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In 2010 IEEE International Conference on Software Maintenance, pages 1–10, 2010.
- [2] A. Arcuri and L. Briand. A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.
- [3] U. Azadi, F. Arcelli Fontana, and D. Taibi. Architectural smells detected by tools: a catalogue proposal. In 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), pages 88–97, 2019.
- [4] V. Basili, G. Caldeira, and H. Rombacj. The goal question metric approach. *Encyclopedia of Soft. Eng.*, 2:528–532, 1994.
- [5] J. Choma Neto, T. Gaieski, A. M. Amaral, and T. E. Colanzi. Quanti-qualitative analysis of a memetic algorithm to optimize product line architecture design. In 2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI), pages 498–505, Volos, Greece, 2018.
- [6] P. Clements, F. Bachmann, and L. Bass. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2 edition, 2010.
- [7] C. A. C. Coello, G. B. Lamont, D. A. Van Veldhuizen, et al. *Evolutionary algorithms for solving multi-objective problems*, volume 5. Springer, 2007.
- [8] T. E. Colanzi, W. K. Assunção, S. R. Vergilio, P. R. Farah, and G. Guizzo. The symposium on search-based software engineering: Past, present and future. *Information and Software Technology*, 127:106372, 2020.
- [9] T. E. Colanzi, S. R. Vergilio, I. Gimenes, and W. N. Oizumi. A search-based approach for software product line design. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 237–241. ACM, 2014.
- [10] A. C. Contieri, G. G. Correia, T. E. Colanzi, I. M. Gimenes, E. A. Oliveira, S. Ferrari, P. C. Masiero, and A. F. Garcia. Extending uml components to develop software product-line architectures: Lessons learned. In *European Conference on Software Architecture*, pages 130–138. Springer, 2011.
- [11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
- [12] P. M. Donegan and P. C. Masiero. Design issues in a component-based software product line. In *Brazilian Symposium on Components, Architectures and Software Reuse (SBCARS)*, pages 3–16. Citeseer, 2007.
- [13] É. L. Féderle, T. do Nascimento Ferreira, T. E. Colanzi, and S. R. Vergilio. Opla-tool: a support tool for search-based product line architecture design. In *Proceedings of the 19th International Conference on Software Product Line*, pages 370–373. ACM, 2015.
- [14] E. Figueiredo et al. Evolving software product lines with aspects: an empirical study on design stability. In *Proc. of ICSE’08*, pages 261–270. ACM, 2008.
- [15] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA,

- 1999.
- [16] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.
- [17] W. M. Freire, M. Massago, C. A. Zavadski, A. M. M. Amaral, and T. E. Colanzi. Opla-tool v2.0: a tool for product line architecture design optimization. In *Proceedings of 34th Brazilian Symposium on Software Engineering (SBES '20)*, October 21–23, 2020, Natal, Brazil. ACM, 2020.
- [18] J. Garcia. *A unified framework for studying architectural decay of software systems*. University of Southern California, 2014.
- [19] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258. IEEE, 2009.
- [20] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, March 2009.
- [21] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: a survey and directions for future work. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, pages 5–18, 2014.
- [22] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11, 2012.
- [23] E. A. O. Junior, I. M. S. Gimenes, and J. C. Maldonado. Systematic management of variability in uml-based software product lines. *Journal of Universal Computer Science*, 16(17):2374–2393, sep 2010.
- [24] D. Le, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural decay in open-source software. In *IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018.
- [25] F. J. Linden, K. Schmid, and E. Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Bus.Media, 2007.
- [26] R. E. Lopez-Herrejon, L. Linsbauer, and A. Egyed. A systematic mapping study of search-based software engineering for software product lines. *Information and Software Technology*, 61(C):33–51, May 2015.
- [27] I. Macia, A. Garcia, C. Chavez, and A. von Staa. Enhancing the detection of code anomalies with architecture-sensitive strategies. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 177–186. IEEE, 2013.
- [28] T. T. Madrigar, T. E. Colanzi, W. Oizumi, and A. Garcia. Prevention of architectural anomalies in optimizing product line architecture design. *Ibero-American Conference on Software Engineering (CibSE) - Technical Symposium 2020*, 2020.
- [29] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb. Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal*, 25(2):529–552, Jun 2017.
- [30] T. Mariani and S. R. Vergilio. A systematic review on search-based refactoring. *Information and Software Technology*, 83:14–34, 2017.
- [31] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, 47(05):1008–1028, 2019.
- [32] J. C. Neto, C. H. da Silva, T. E. Colanzi, and A. M. M. M. Amaral. Are as profitable to search-based product-line architectures design? *IET Software*, 13(6):587–599, 2019.
- [33] C. Nunes, U. Kulesza, C. Sant’Anna, I. Nunes, A. Garcia, and C. Lucena. Assessment of the design modularity and stability of multi-agent system product lines. *Journal of Universal Computer Science*, 15(11):2254–2283, jun 2009.
- [34] W. Oizumi, A. Garcia, L. D. S. Sousa, B. Cafeo, and Y. Zhao. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 440–451. IEEE, 2016.
- [35] E. G. Perissato, J. C. Neto, T. E. Colanzi, W. Oizumi, and A. Garcia. On identifying architectural smells in search-based product line designs. In *VII Brazilian Symposium on Software Components, Architectures, and Reuse.*, pages 13–22, 2018.
- [36] D. F. d. Silva, L. F. Okada, T. E. Colanzi, and W. K. G. Assunção. Enhancing search-based product line design with crossover operators. In *Genetic and Evolutionary Computation Conference (GECCO 20)*, page 12501258, 2020.
- [37] Software Engineering Institute. *Arcade game maker: Pedagogical product line*. Software Engineering Institute, (10):115–118, 2016.
- [38] Y. D. Verdecia, T. E. Colanzi, S. R. Vergilio, and M. C. B. Santos. An enhanced evaluation model for search-based product line architecture design. In *XX Ibero-American Conference on Software Engineering (CibSE2017)*, Buenos Aires, 2017.
- [39] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos. Identifying architectural problems through prioritization of code smells. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SB-CARS)*, pages 41–50. IEEE, 2016.
- [40] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2014.

- [41] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. Identifying and quantifying architectural debt. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pages 488–498. IEEE, 2016.
- [42] T. J. Young. Using aspectj to build a software product line for mobile devices. PhD thesis, University of British Columbia, 2005.

Table 1. Catalog of architectural smells prevented by OPLA-Tool-ASP

Type	Definition
Unused Interface	<p>Description: An interface that is not linked to any component, adding unnecessary complexity to the system [18].</p> <p>Detection: Identification of isolated interfaces.</p> <p>Guideline: Discard solutions that contain such a smell.</p>
Unused Brick	<p>Description: Occurs when all interfaces of a component have the Unused Interface smell [18].</p> <p>Detection: Identification of packages containing only isolated interfaces.</p> <p>Guideline: Discard solutions that contain such a smell.</p>
Concern Overload	<p>Description: A component that addresses an excessive number of features, breaking the single responsibility principle [18].</p> <p>Detection: Consider the features of each SPL as the concerns of the PLA design. Use the algorithm proposed by Garcia [18] to define the threshold and count the features associated with the classes and interfaces of the design.</p> <p>Guideline: Apply the Feature-driven Operator to modularize some of the features associated to the smelly class/interface.</p>
Link Overload	<p>Description: A component that has excessive dependence on other components, thus impairing the separation of concerns and the isolation of changes. These dependencies can manifest as inbound or outbound links [18].</p> <p>Detection: We used an adaptation of the algorithm proposed by Garcia [18] to define the threshold for each type of link. Associations, dependencies, realizations were considered as links. In addition, we included links for classes having attributes that are of the type of other classes. The threshold was calculated for each direction – input, output, and bidirectional – rounding up fractional values. Then, we counted the number of links for all classes and interfaces. The adaptation is related to do not consider inheritance as a link in the threshold definition.</p> <p>Guideline: Penalize the fitness of the smelly solution, so that it is worse evaluated in the process of selecting the best designs to remain in the optimization process.</p>

Table 2. PLA Information.

PLA	#Packages	#Interfaces	#Classes	#Features
AGM	9	14	30	11
MM	8	15	14	14
BET	56	36	126	118

Table 3. Thresholds Used in the Empirical Study.

Architectural Smell	AGM			MM			BET		
Concern Overload	2			4			7		
Link Overload	Input	Output	Bidirectional	Input	Output	Bidirectional	Input	Output	Bidirectional
	2	2	1	3	3	2	3	5	2

Table 4. Number of Non-dominated Solutions by Experiment.

PLA	OPLA-Tool-ASP	OPLA-Tool
AGM	8	19
MM	6	20
BET	9	19
Total	23	58

Table 5. Number of Occurrences of Architectural Smells Detected in Alternative Designs Generated by OPLA-Tool-ASP.

Architectural Problem	Original Design	OPLA-Tool-ASP									
		S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
AGM											
Unused Interface	0	0	0	0	0	0	0	0	0	0	-
Unused Brick	0	0	0	0	0	0	0	0	0	0	-
Concern Overload	2	2	2	2	2	2	2	2	2	2	-
Link Overload	5	3	4	4	4	4	4	4	4	4	-
MM											
Unused Interface	0	0	0	0	0	0	0	-	-	-	-
Unused Brick	0	0	0	0	0	0	0	-	-	-	-
Concern Overload	4	2	1	2	2	2	2	-	-	-	-
Link Overload	3	3	3	4	4	4	4	-	-	-	-
BET											
Unused Interface	0	0	0	0	0	0	0	0	0	0	0
Unused Brick	0	0	0	0	0	0	0	0	0	0	0
Concern Overload	7	5	5	5	5	5	5	5	5	5	5
Link Overload	31	12	12	12	12	12	12	12	12	11	11

Table 6. Number of Occurrences of Architectural Smells Detected in Alternative Designs Generated by OPLA-Tool.

Architectural Problem	Original Design	OPLA-Tool																			
		S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
AGM																					
Unused Interface	0	6	4	6	6	13	15	14	12	5	6	6	5	2	2	8	4	1	4	1	-
Unused Brick	0	1	1	1	1	6	2	6	2	4	4	4	4	2	4	6	4	4	5	0	-
Concern Overload	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-
Link Overload	5	8	8	8	7	6	6	7	7	9	9	9	9	11	10	8	12	11	11	11	-
MM																					
Unused Interface	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Unused Brick	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Concern Overload	4	3	2	2	2	1	1	1	1	1	2	1	1	1	1	2	0	0	0	0	0
Link Overload	3	7	7	7	7	7	6	6	8	7	7	7	7	7	8	7	5	5	5	6	5
BET																					
Unused Interface	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	-
Unused Brick	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	-
Concern Overload	7	6	6	6	6	6	5	5	5	4	5	5	5	5	5	5	6	6	5	6	-
Link Overload	31	38	31	37	37	32	36	31	31	36	27	29	32	30	29	29	26	27	29	31	-

Table 7. Number of Smelly Elements (Classes, Interfaces and Packages) in the Original Design and in the Solutions Found by OPLA-Tool-ASP.

PLA	Original Design	OPLA-Tool-ASP									
		S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
AGM	7	5	6	6	6	6	6	6	6	-	
MM	7	5	4	6	6	6	-	-	-	-	
BET	38	17	17	17	17	17	17	17	16	16	

Table 8. Number of Smelly Elements (Classes, Interfaces and Packages) in the Solutions Found by OPLA-Tool.

PLA	OPLA-Tool																			
	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
AGM	17	15	17	16	27	25	29	23	17	18	18	17	15	15	21	19	15	19	14	-
MM	12	11	9	9	8	7	7	9	8	9	8	8	8	9	9	5	5	5	6	5
BET	44	37	43	43	38	42	36	36	41	31	34	37	35	34	34	34	35	34	37	-