Chapter 18
Reengineering UML Class Diagram Variants into a Product Line Architecture

Wesley Klewerton Guez Assunção, Silvia R. Vergilio, and Roberto E. Lopez-Herrejon

Abstract Software reuse is a way to reduce costs and improve the quality of products. In practice, software reuse is commonly done by opportunistic strategies. In these strategies, the artifacts are simply copied/cloned and modified/adapted to fulfill existing needs. Opportunistic reuse leads to a set of system variants developed independently, generating technical debts. The maintenance and evolution of these independent variants are a costly and difficult task since most of the times the practitioners do not have a global view of such variants nor a clear understanding of the actual structure of the system. In such a case, a systematic reuse approach is paramount. Software product line engineering (SPLE) is a well-established approach to deal with a set of product variants in a specific domain, including systematic reuse in the software development process. One of the main design assets generated during the SPLE is the product line architecture (PLA), which describes how commonalities and variabilities are implemented in an SPL. Designing a PLA from scratch is challenging, since it must contemplate a detailed description of a whole family of products. PLAs can be obtained from existing product variants, requiring less effort and time from practitioners. Commonly, UML class diagrams of system products are available or can be reverse engineered easily. These UML class diagrams are a rich source of information to support PLA creation. In this chapter, we describe our method of reengineering UML class diagram of variants into an initial version of a PLA. Our method relies on a search-based technique to merge a set of UML model variants and insert annotations in model elements to describe

W. K. G. Assunção (✉)
ISSE, Johannes Kepler University Linz, Linz, Austria

OPUS, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil
e-mail: wesley.assuncao@jku.at

S. R. Vergilio
DInf, Federal University of Paraná, Curitiba, Brazil
e-mail: silvia@inf.ufpr.br

R. E. Lopez-Herrejon
LOGTI, ETS, University of Quebec, Montreal, QC, Canada
e-mail: roberto.lopez@etsmtl.ca

E. OliveiraJr (ed), *UML-Based Software Product Line Engineering with SMarty*,
https://doi.org/10.1007/978-3-031-18556-4_18

the system features they belong to. The output of our method is an annotated UML class diagram that shows the whole structure of product variants that allows practitioners to reason better about the adoption of SPLE, aiding communication among stakeholders, supporting SPLE planning, and helping estimate maintenance, evolution, and testing activities.

18.1 Introduction

The development of a software system from scratch is a complex and high-cost activity. Software reuse, which is based on the use of existing artifacts to develop new software systems, is a well-established strategy to reduce costs, improve productivity, and increase quality [13]. The reuse of artifacts can be performed in different levels of abstraction and in different phases of the software development life cycle. Artifacts that can be reused include source code, design models, and test cases, to cite some.

In practice, software reuse is generally carried out using an opportunistic reuse strategy, which is also known as clone-and-own reuse, copy-and-paste reuse, or ad hoc reuse [10]. In this strategy, existing software artifacts are cloned/copied and adapted/modified to fulfill the new requirements. The opportunistic reuse strategy offers a simple way to reuse software artifacts. It does not require an upfront investment and quickly obtains adequate short-term results. However, the extensive use of opportunistic software reuse quickly becomes problematic. For example, opportunistic reuse results in extensive refactoring, adding technical debt, and eventually leading to unanticipated behavior, violated constraints, conflicts in assumptions, fragile structure, and software bloat [14]. To make matters even worse, the simultaneous maintenance of many independent variants of the same system is a complex activity, as duplicated functionalities must be managed individually [7].

We can find several studies in the literature dealing with the reengineering of multiple system variants into SPLs [2, 15]. Only a few of them focus on the definition of a product line architecture (PLA) from existing product variants. Software architectures are artifacts that provide a high-level view of functional parts of systems, allowing analysis of their structure and supporting design decisions [5]. In addition, PLAs describe how commonalities and variabilities are implemented in an SPL. To recover or discover a PLA that best represents an existing family of software products demands high human effort [21]. Furthermore, the few existing approaches to recover/discover are based on source code [8, 11, 12].

Taking into account the aforementioned limitations of existing work, in a previous study, we presented an approach to automatically merge multiple UML model variants to obtain a documented software architecture that is a step toward the definition of a PLA [3]. The goal of our approach is to discover a global model that contains an overview of all the implementation elements spread across the different variants. The input of our approach is a set of UML model variants, and the output

is a complete model, the most similar to all variants. The proposed merging process relies on a search-based technique, in which the evolutionary process is in charge of dealing with domain-specific constraints of systems under consideration and possible conflicts among models merging operations. We implemented our approach with a genetic algorithm and evaluated it using four case studies from different domains and with different sizes. The evaluation of the proposed approach showed that the merging of UML class diagram variants represented good documented architectures to support the maintenance, evolution, and testing of system variants.

In this book chapter, we describe our approach of merging UML models to obtain a documented architecture [3] and introduce an additional step of variability annotations, where UML model elements are annotated to describe existing variability. The goal is to provide a UML-based PLA to aid practitioners to reengineer independent variants into SPLs.

The remainder of this chapter is as follows: In Sect. 18.2, we describe in detail the proposed search-based approach. The evaluation of the proposed approach and the results are presented in Sect. 18.3. Finally, Sect. 18.5 presents the final remarks and suggestions of future work.

18.2 Proposed Approach

In this section, we describe our approach to reengineer UML class diagram variants into PLAs. The proposed approach has two steps: (i) a search-based algorithm to merge UML class diagram variants to obtain a global UML model having as many as possible of the features contained across the variants and (ii) the process to annotate variability information in the UML models.

The input for the proposed approach is a set of UML class diagrams, which is used for the search-based algorithm, and traceability information of features implemented in the products and the model elements mapped to them, which is used for variability annotation. This traceability[1] should be provided by practitioners based on their knowledge or discovered by using automated tools. The output is a PLA, composed of a global UML model and annotated elements that describe variability information. To illustrate how our approach works, we rely on three variants of a banking system [19]. These variants[2] are presented in Fig. 18.1.

[1] Traceability links describe where features are implemented in the source code. The definition or reverse engineering of traceability links is out of the scope of this work. For further details, see [16].

[2] Available at https://github.com/but4reuse/but4reuse/wiki/Examples.

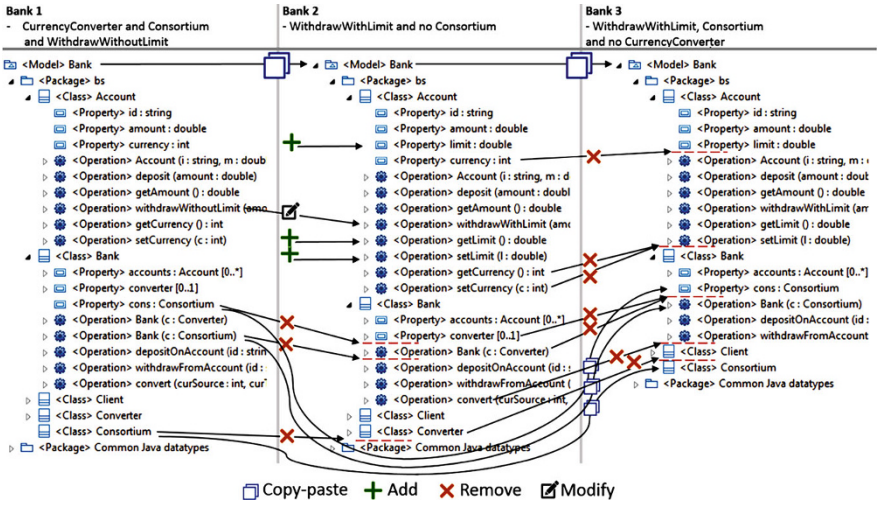


Fig. 18.1 Three banking system model variants, adapted from [19]

18.2.1 Step 1: Search-Based Model Merging

The first step of our approach relies on a genetic algorithm with the goal of merging UML class diagram variants. In what follows, we present the representation of individuals and generation of the initial population, the fitness function, and the genetic operators.

18.2.1.1 Representation of Individuals and Initial Population

Our search-based approach deals with models created with the well-known and widely used Eclipse Modeling Framework (EMF) [22]. We represent the models using EMF-based UML2[3] implementation of the UML^{TM} 2.x metamodel for the Eclipse platform. When models are represented using EMF-based UML2 data types, they can be compared and modified. These operations enabled by EMF tools are the basis of our search-based approach.

Based on the proposed representation and considering the set of UML class diagram variants used as input, the initial population is created by duplicating every variant until reaching the population size. For example, consider a population of 90 individuals for the search-based algorithm and the three variants of Fig. 18.1. The initial population will be composed of 30 copies/duplicates of each input model. Each duplicated mode variant is an individual.

[3] http://wiki.eclipse.org/MDT/UML2.

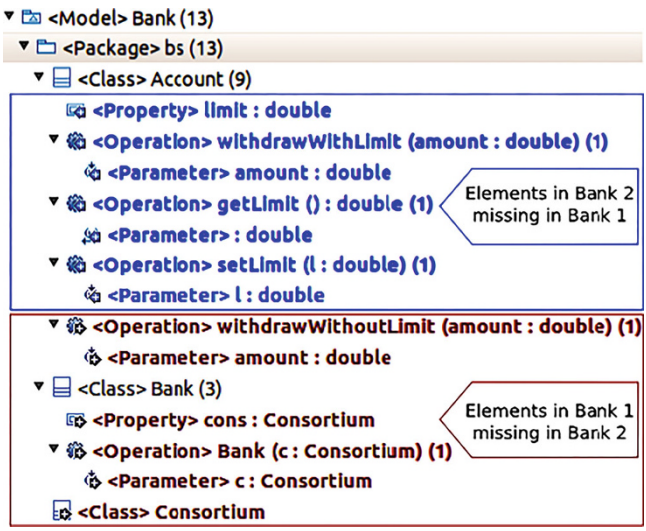


Fig. 18.2 Differences between variants Bank 1 and Bank 2

18.2.1.2 Fitness Function

The fitness function of our approach is based on differences among UML models of system variants. The computation of these differences is based on the Eclipse EMF Diff/Merge tool.[4] EMF Diff/Merge compares two models and returns the differences between them. EMF Diff/Merge computes three essential types of differences between models: (i) presence of an unmatched element, which refers to an element in a model that has no match in the opposite model; (ii) presence of an unmatched reference value, which means that a matched element references another element in only one model; (iii) presence of an unmatched attribute value, where a matched element owns a certain attribute value in only one model.

Figure 18.2 presents the output of EMF Diff/Merge when comparing differences between variants Bank 1 and Bank 2 (Fig. 18.1). The total number of differences is 13, but it is composed of two sets of differences. At the top of the figure, we have seven differences that are elements present in Bank 2 but missing in Bank 1, and at the bottom of the figure, we have six elements that belong to Bank 1 but do not appear in Bank 2.

EMF Diff/Merge tool is able to compare only two or three models at once. However, to evaluate a candidate architecture, we have to compute differences from one model to many model variants. Considering this, we propose a fitness function composed of the sum of differences from one model to all input model variants. Definition 18.1 presents the fitness function called here *model similarity*. The

[4] http://www.eclipse.org/diffmerge/.

function *diff* represents the number of differences found by using EMF Diff/Merge, but here we sum only the set of differences that indicate the elements that exist in the variant v but are missing in the *candidate_model*. There are no distinctions among the three essential types of differences.

Definition 18.1 (Model Similarity (MS)) Model similarity expresses the degree of similarity of the candidate architecture model to a set of model variants:

$$MS = \sum_{v \in Variants} diff(candidate\_model, v) \tag{18.1}$$

To illustrate the computation of MS, we consider the candidate architecture model presented in Fig. 18.3a and the input models in Fig. 18.1. Using EMF Diff/Merge tool, we obtain the sets of differences presented in Fig. 18.3b,c, and d, corresponding to Bank 1, Bank 2, and Bank 3, respectively. For our fitness function, only the differences from the candidate architecture to each variant are relevant, which are highlighted in the figures. There are no differences from the candidate model to Bank 1. From the candidate model to Bank 2, there exist six differences. From the candidate model to Bank, 3 we have also six differences. We can observe 12 differences from the candidate model to all input variants and then MS = 12. The goal is to minimize the value of MS. An ideal solution has MS equal to zero, which indicates that the candidate architecture has all elements from the variants for which we want to discover the corresponding architecture.

18.2.1.3 Genetic Operators

The set of differences returned by EMF Diff/Merge is used to perform crossover and mutation. This result of EMF Diff/Merge also allows duplication and/or modification of models to incorporate the changes done by the operators. The mechanisms of the operators of crossover and mutation are described next.

Crossover
The crossover operator starts with two candidate architectures. From these two models, we generate two children: one with the differences merged and one without the differences. For instance, let us consider any parent models X and Y. The children will be:

- *Crossover Child Model 1*: This model has differences between its parents merged. For example, the elements of X that are missing on Y are merged in this later, or vice versa. Both ways will produce the same child.
- *Crossover Child Model 2*: This child is generated by removing the differences between the parents. For example, the differences of X that are missing on Y are removed, or vice versa. Both ways will produce the same child.

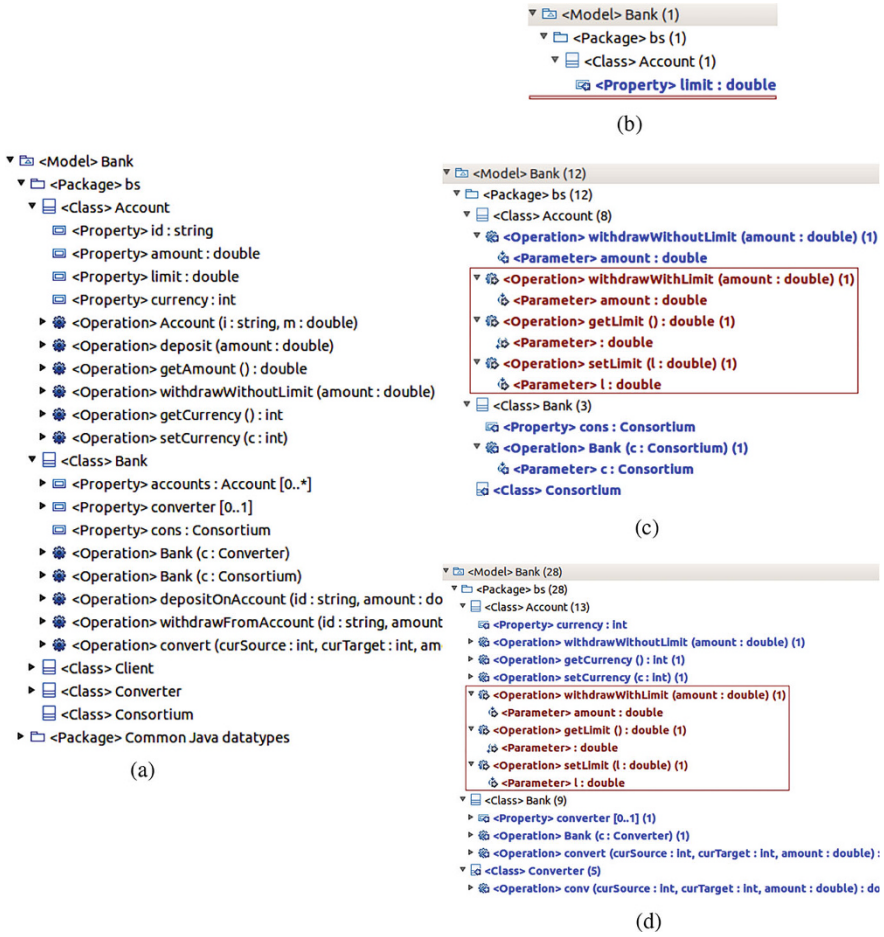


Fig. 18.3 Example of fitness evaluation. (**a**) Candidate model. (**b**) Differences to Bank 1 = 0. (**c**) Differences to Bank 2 = 6. (**d**) Differences to Bank 3 = 6

The strategy adopted by Child Model 1 aims at creating a model that has more elements, going toward a more complete system architecture. On the other hand, the strategy used by Child Model 2 has the goal of eliminating possible conflicting elements from a candidate architecture.

To illustrate the crossover operator, let us consider as parents Bank 1 and Bank 2 presented in Fig. 18.1 and the differences between them, presented in Fig. 18.2. The offspring generated by crossover is presented in Fig. 18.4. In Fig. 18.4a, we have the child with all differences merged (highlighted) and in Fig. 18.4b the child with the differences removed.

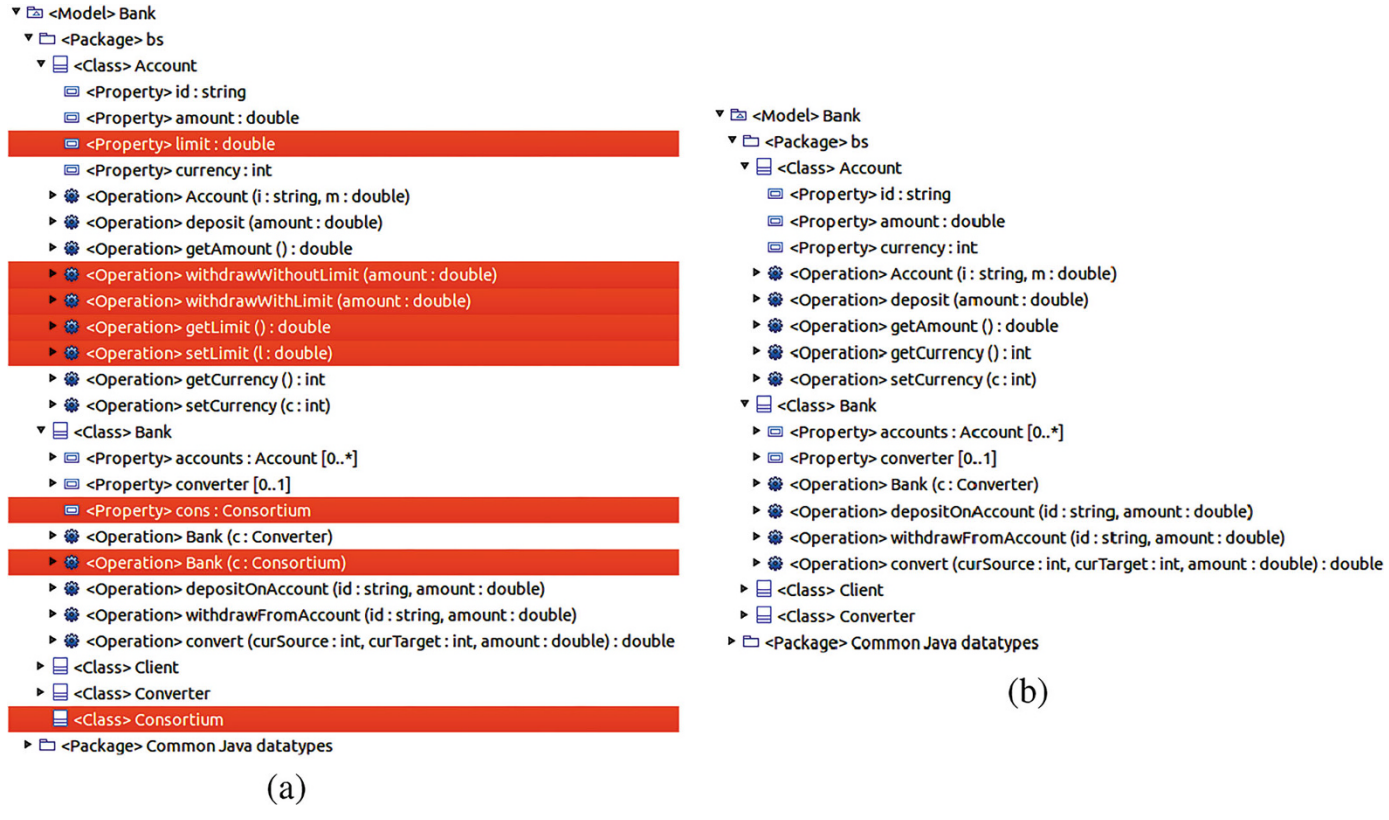


Fig. 18.4 Example of crossover between Bank 1 and Bank 2. (**a**) Crossover Child 1. (**b**) Crossover Child 2

Mutation
The mutation operator aims at applying only one modification in each model parent. The start point of the mutation is two candidate architectures, and the result is also two children. Let us again consider any parent models X and Y. The children are:

- *Mutation Child Model 1*: The first child is created by merging one difference from model Y to model X. After randomly selecting one element of model Y but missing on model X, this element is added into model X.
- *Mutation Child Model 2*: The same process described above is performed again, but in the opposite direction, namely, including one element of model X into model Y.

An example of mutation between Bank 1 and Bank 2 (Fig. 18.1) is presented in Fig. 18.5. Considering the differences shown in Fig. 18.2, we have seven differences to select one to be included in Bank 1 and six differences to select one to be included in Bank 2. As highlighted in Fig. 18.5a, the attribute `limit` was chosen to be included in Bank 1. In the child of Fig. 18.5b we can see that the class `Consortium` was selected to be included in Bank 2.

The mutation process can select a difference that is part of another difference. In such cases, the entire owning difference is moved to the child. For example, when a mutation selects a parameter owned by an operation, the entire operation is moved to the child.

Selection
The genetic algorithm of our approach uses the binary tournament strategy whereby a set of individuals are randomly selected from the population. Among the randomly selected individuals, the ones with the best fitness are chosen to undergo crossover and mutation [9].

18.2.2 Step 2: Variability Annotation

The best solution found during the evolutionary process in Step 1 is the basis for this step. Since in our context a PLA is a global UML class diagram with annotations regarding variabilities, to generate such representation the traceability information provided as input for our approach is used to annotate the class diagram. To include annotation of variability information in the PLA, we decided to use UML-owned comments which are available for each element of a UML class diagram. By adopting this strategy, the obtained PLA can be viewed in any UML editor.

The process of variability annotation is presented in Algorithm 1. Basically, the algorithm goes through all the UML elements comparing them to the traceability links. When there is a matching between the model element of the class diagram and the model element in the traceability information, an owned comment is assigned to the UML element with the name of the feature obtained from the traceability information.

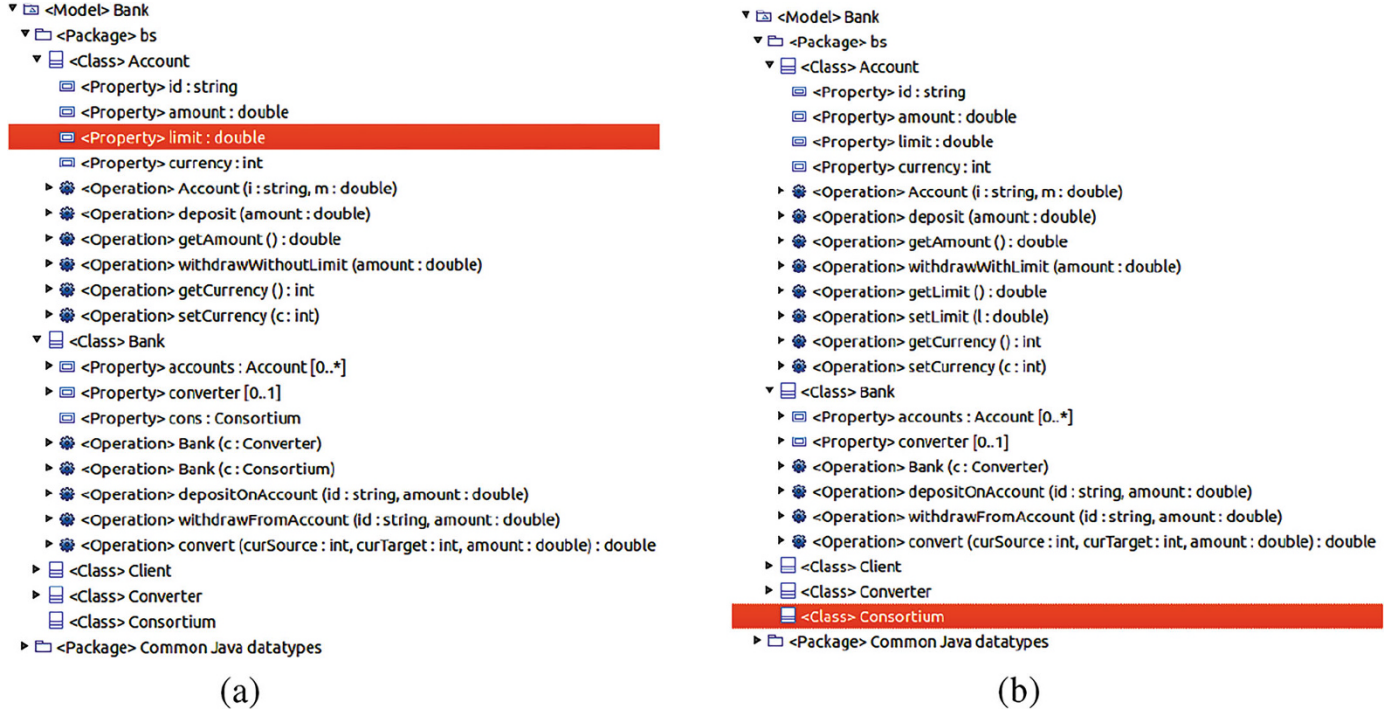


Fig. 18.5 Example of mutation between Bank 1 and Bank 2. (**a**) Mutation Child 1. (**b**) Mutation Child 2

Algorithm 1: Variability annotation

Input: UML class diagram, traceability information
Output: PLA

```
1 modelElements ← all UML model elements from the UML class diagram;
2 traceLinks ← all trace link tuples (feature, modelElement) from the Traceability
    information;
3 for each element ∈ modelElements do
4 |   for each trace ∈ modelElements do
5 |   |   if modelElements.name = trace.modelElement.name then
6 |   |   |   modelElements.ownedComment ← trace.feature.name;
7 |   |   end if
8 |   end for
9 end for
```

Figure 18.6 presents a PLA constructed using the merged model obtained in the first step of our approach. The figure presents the variability information of an attribute of `Bank` with the comment that indicates it belongs to `Converter`.

18.3 Evaluation

In this section, we present the setup and the subject systems used to evaluate the proposed approach, along with the results obtained and their analysis.

18.3.1 Implementation Aspects and Experimental Setup

We implemented our work using JMetal[5] framework which provides several algorithms for multi-objective and mono-objective optimization [6]. We selected the mono-objective generational genetic algorithm (GA) [9]. Our GA was designed to deal with a minimization problem; recall that an ideal solution for our architecture recovery problem is an individual (i.e., candidate architecture) with fitness equal to zero (0).

EMF framework was used to load and save models. For the evolutionary process, where we compare and modify models, we used EMF Diff/Merge. Despite of EMF Diff/Merge having many functionalities, we needed to develop a customized match policy. The default match policies of EMF Diff/Merge only perform comparisons based on XMI:ids. However, model variants could have similar semantics even with different structures. Our customized match police considers qualified names, data types, and relationship types.

[5] Available at: http://jmetal.sourceforge.net/.

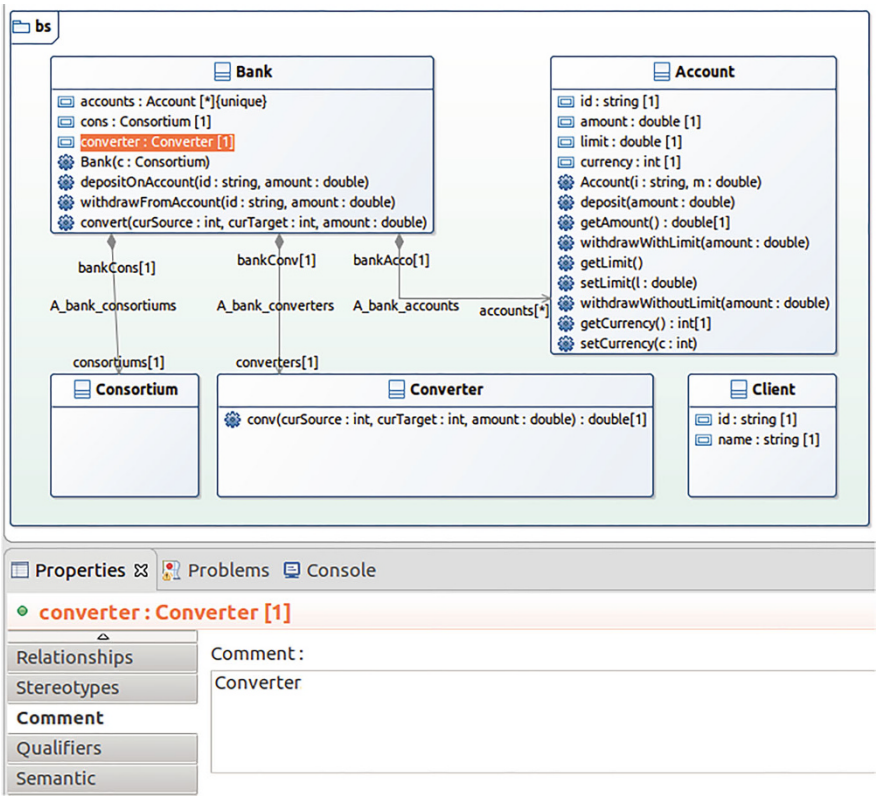


Fig. 18.6 Example of variabilities in the PLA

The GA parameters were population size = 200, crossover probability = 0.95, mutation probability = 0.2, and number of fitness evaluations = 8000. We set the parameters of crossover and mutation based on default values used in other discrete problems available on JMetal. Population size and the number of evaluations were set based on hardware's limitation. When we tried to use greater values for these two latter parameters, they caused limited memory exceptions. The elitism strategy adopted in the generational GA is to copy the best four individuals of one generation to the next one. The number of fitness evaluations is the stop criterion. The experiments were run on a machine with an Intel® Core^{TM} i7-4900MQ CPU with 2.80 GHz, 16 GB of memory, and running a Linux platform.

Table 18.1 Banking system

Variant	Features				#Cl	#Attr	#Op	#Rel
	BS	WL	CON	CC				
1	✓				3	5	6	1
2	✓		✓		4	6	7	3
3	✓	✓			3	6	8	1
4	✓			✓	4	7	11	2
Baseline	✓	✓	✓	✓	5	9	14	4

BS, base; WL, withdraw limit; CON, consortium; CC, currency converter

Table 18.2 Draw product line

Variant	Features						#Cl	#Attr	#Op	#Rel
	DPL	L	R	C	W	F				
1	✓	✓					4	13	26	3
2	✓	✓	✓				5	24	37	4
3	✓		✓				4	18	29	3
4	✓	✓		✓			4	22	27	3
5	✓		✓	✓			4	27	30	3
6	✓	✓			✓		4	15	27	3
7	✓		✓		✓		4	20	30	3
8	✓		✓	✓		✓	4	33	32	3
Baseline	✓	✓	✓	✓	✓	✓	5	42	41	4

DPL, base; L, line; R, rectangle; C, color; W, wipe; F, fill

18.3.2 Subject Systems

In our experiment, we used four subject systems, where each one is a set of UML model variants implementing different system features, and composed of classes, attributes, operations, and relationships. The subject systems are banking system (BS), a small banking application composed of four features [18]; draw product line (DPL), a system to draw lines and rectangles with six features [1]; video on demand (VOD), which implements 11 features for video-on-demand streaming [1]; and ZipMe (ZM), a set of tools to file compression with 7 features [1]. The variants are presented in Tables 18.1, 18.2, 18.3, and 18.4, respectively. These tables show the features, number of classes (#Cl), number of attributes (#Attr), number of operations (#Op), and number of relationships (#Rel) for each variant. This information was computed using SDMetrics.[6] Only BS is originally a set of UML model variants; for other subject systems, we reverse engineered the models from Java code using the Eclipse MoDisco.[7]

[6] http://www.sdmetrics.com.

[7] https://eclipse.org/MoDisco.

Table 18.3 Video on demand

Variant	Features											#Cl	#Attr	#Op	#Rel
	VOD	SP	SelM	StaM	PI	VRC	P	StoM	QP	CS	D				
1	✓	✓	✓	✓	✓	✓						32	362	217	75
2	✓	✓	✓	✓	✓	✓	✓					32	362	217	75
3	✓	✓	✓	✓	✓	✓		✓				33	364	221	77
4	✓	✓	✓	✓	✓	✓	✓	✓				33	364	221	77
5	✓	✓	✓	✓	✓	✓			✓			33	364	221	77
6	✓	✓	✓	✓	✓	✓	✓		✓			33	364	221	77
7	✓	✓	✓	✓	✓	✓		✓	✓			34	366	225	79
8	✓	✓	✓	✓	✓	✓				✓		37	377	232	87
9	✓	✓	✓	✓	✓	✓	✓			✓		37	377	232	87
10	✓	✓	✓	✓	✓	✓		✓		✓		38	379	236	89
11	✓	✓	✓	✓	✓	✓			✓	✓		38	379	236	89
12	✓	✓	✓	✓	✓	✓					✓	35	374	226	82
13	✓	✓	✓	✓	✓	✓	✓				✓	35	374	226	82
14	✓	✓	✓	✓	✓	✓		✓			✓	36	376	230	84
15	✓	✓	✓	✓	✓	✓			✓		✓	36	376	230	84
16	✓	✓	✓	✓	✓	✓				✓	✓	40	389	241	94
Baseline	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	42	393	249	98

VOD, base; SP, start player; SelM, select movie; StaM, start movie; PI, play Imm; VRC, VRC interface; P, pause; StoM, stop movie; QP, quit player; CS, change server; D, details

We have variants with all possible feature combinations for every subject system. However, we selected only variants that implement at most half of the non-mandatory features. To select these variants, we followed the rule:

$$threshold = (RoundUp\left(\frac{\#all\_features - \#mandatory\_features}{2}\right) + \#mandatory\_features)$$

We selected for our experiment only variants that implement a number of features below the threshold. The reason to select only a subset of variants is to have the combinations of features spread on different variants, to assess the ability of our approach to merge the models and get good system architectures. For each subject system, we also had a variant that implements all features, i.e., the most complete variants. We use this variant as a baseline for our analysis, since we consider this variant as the most similar model to a known system architecture. In the last line of Tables 18.1, 18.2, 18.3, and 18.4, there is information about the baseline.

Observing the information in the subject system tables (Tables 18.1, 18.2, 18.3, and 18.4), we can see that there are no variants with as many features as the baselines. Furthermore, the number of classes, attributes, operations, and relationships in the variants of all systems is smaller than the baselines.

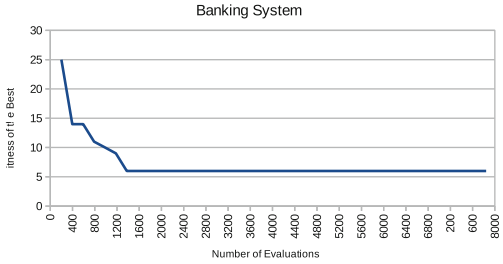
Table 18.4 ZipMe

Variant	Features							#Cl	#Attr	#Op	#Rel
	ZM	C	CRC	AC	GZIP	A32	E				
1	✓	✓						22	212	241	64
2	✓	✓	✓					23	215	251	66
3	✓	✓		✓				22	212	243	66
4	✓	✓	✓	✓				23	215	253	68
5	✓	✓			✓			25	223	263	68
6	✓	✓	✓		✓			26	229	282	72
7	✓	✓		✓	✓			25	223	265	70
8	✓	✓				✓		23	216	263	69
9	✓	✓	✓			✓		24	219	273	71
10	✓	✓		✓		✓		23	216	265	71
11	✓	✓			✓	✓		26	227	285	73
12	✓	✓					✓	23	219	262	70
13	✓	✓	✓				✓	24	223	279	74
14	✓	✓		✓			✓	23	219	264	72
15	✓	✓			✓		✓	26	230	284	74
16	✓	✓				✓	✓	24	223	284	75
Baseline	✓	✓	✓	✓	✓	✓	✓	28	241	334	87

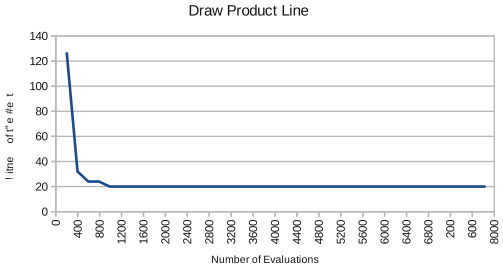
ZM, ZipMe; C, compress; CRC, CRC-32 checksum; AC, archive check; GZIP, GZIP format support; A32, Adler32 checksum; E, extract

18.3.3 Results and Analysis

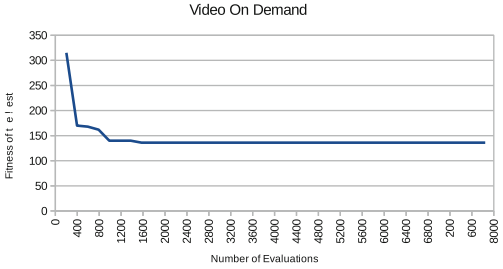
Figure 18.7 shows the evolution of the best candidate architecture in each GA generation during the first step of our approach. The best individual of each system after the first 200 fitness evaluations is an input model from the initial population that has the least difference from other input models. For BS, the best individual is variant 4 that has 25 differences from the input. For DPL, the best initial individual is variant 2 with 127 differences. For VOD, the best initial candidate architecture is variant 16 with 315 differences. Finally, variant 11 of ZM is the best individual of the initial population having 854 differences from the input. These individuals are the first solutions presented in the charts in Fig. 18.7. Observing the figures, we can see how the evolutionary process is able to find better candidate architectures by reducing the number of differences. On average, the best solution is found after 1400 fitness evaluations. VOD is the simplest subject system, since the best solution was reached with approximately 1000 fitness evaluations. On the other hand, ZM is the most complex system, needing approximately 1800 fitness evaluations to reach the best solution. As expected for a GA, in all subjects, there is a great improvement in the number of found solutions in the initial generations, and then the search remains stable.



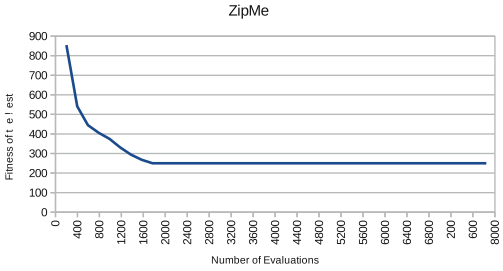
(a)



(b)



(c)



(d)

Fig. 18.7 Evolution of the best individual. (**a**) Banking system. (**b**) Draw product line. (**c**) Video on demand. (**d**) ZipMe

Table 18.5 Candidate architectures

System	Model	MS	#Cl	#Attr	#Op	#Rel
BS	Baseline	20	5	9	14	4
	Best individual	6	5	9	14	3
DPL	Baseline	40	5	42	41	4
	Best individual	20	5	42	41	4
VOD	Baseline	162	42	393	249	98
	Best individual	136	42	393	249	98
ZM	Baseline	633	28	241	334	87
	Best individual	250	28	241	381	79

#Cl, number of classes; #Attr, number of attributes; #Op, number of operations; #Rel, number of relationships

Another piece of information gathered during the evaluation of the first step of our approach is the runtime. The amount of time spent by the GA to perform the entire evolutionary process was BS = 55 s 740 ms, DPL = 6 m 13 s 17 ms, VOD = 1 h 46 m 55 s 698 ms, and ZM = 2 h 10 m 29 s 267 ms. GA ran very fast for BS, which has the smallest number of features, classes, attributes, operations, and relationships. DPL has more features and model elements (Table 18.2) than BS, and for this system, the GA took a little more than 6 min. A huge difference in the runtime is observed for VOD and ZM. VOD needed almost 2 h to be finished. ZM is the subject system that required the biggest amount of time; it took more than 2 h.

Now, let us consider the details of the best candidate architecture found, i.e., global UML class diagram. Table 18.5 shows the information of candidate architectures and baseline models. The values of MS presented in the third column are based on the input models. Regarding the number of classes, attributes, operations, and relationships, the baseline model and the best individual model are very similar. For BS, there is only a single difference in the number of relationships, where the best individual has one relationship less. In DPL and VOD, the number of model elements is the same. For ZM, the number of model elements is different in operations and relationships. Despite having a similar number of model elements, we can observe that the values of MS are not similar. As mentioned before in Sect. 18.2.1.2, the fitness function EMF Diff/Merge computes the presence of elements, presence of attributes values, and presence of reference values. This latter difference happens when a model element references to, or belongs to, different model elements. This explains the reason why baselines and best individuals have a similar number of model elements but different values of MS.

Table 18.6 presents the differences between baseline and the best individuals for each system. Since the comparison of EMF Diff/Merge has two directions, we show the number of differences existing from baseline to the best individual (candidate architecture), and vice versa. For example, considering BS, there are seven differences needed for baseline having all elements of the candidate architecture. On the other hand, candidate architecture needs 14 existing differences to have all elements of baseline. In the values of Table 18.6, we can observe that the baseline is less different for systems BS, DPL, and VOD. This means that it is easier to transform

Table 18.6 Differences between baseline and candidate architectures

System	From baseline to best	From best to baseline
BS	7	14
DPL	5	451
VOD	20	3425
ZM	4155	200

baseline in the best than vice versa. For ZM, the solution obtained by the GA is the most similar to the baseline.

The analysis of Tables 18.5 and 18.6 reveals that a model having all features does not imply that it is the most similar to a set of model variants. We can infer this by considering that the best individual obtained by the GA for each system is the most similar to the model variants than the baseline (third column in Table 18.5) and, on the other hand, baseline is more similar to the best individual when comparing these two models (second and third columns of Table 18.6). To illustrate this situation, let us use the models of BS presented in Fig. 18.8. In Fig. 18.8a, the baseline has all features implemented, and in Fig. 18.8b, the best solution found is the most similar to the input models. Observe that in the best solution there exists an operation `withdrawWithoutLimit(amount: double)`. This operation is present in the variants that do not implement the feature WL (see Fig. 18.1), i.e., it is present in three out of four variants. This operation is not present in the baseline model, so this baseline model does not provide a global overview of the variants. The baseline would not serve as a reference for maintaining variants that do not have the feature WL. However, in the architecture, we can find out where the operation `withdrawWithoutLimit` is located.

The results of the second step of our approach, namely, variability annotation, are presented in Table 18.7. The number of model elements annotated with the traceability information is presented in the second column. The runtime for each application is in the last column. Applications VOD and ZM have the largest models; therefore, the variability grafting algorithm took the largest runtime. However, the runtime did not take more than 10 s.

18.4 UML-Based SPLs

Our approach to merge UML class diagrams in order to obtain a documented UML-based PLA is a step toward the reengineering of independent variants into SPLs. On one hand, we used the standard UML class diagram model and widely adopted modeling tools, such as Eclipse Modeling Framework and Eclipse EMF Diff/Merge tool. These tools are commonly adopted for designing of single-product development, which can ease the extractive adoption of SPLs. On the other hand, this same tolling support is limited on covering the whole SPL development life cycle and dealing with variability management [4].

Fig. 18.8 Baseline and best solution for banking system. (**a**) Baseline. (**b**) Best solution

Table 18.7 Variability annotation results

System	Model elements annotated	Runtime	
		s	ms
BS	44	1	481
DPL	103	1	811
VOD	728	8	972
ZM	857	9	31

Considering the above limitations, we envisage the use of SPL-based tools for dealing better with the design and management of variability. For example, SMarty is an approach to manage variabilities in UML diagrams based on a profile and respective guidelines [20]. SMarty is flexible for use since it relies on profile stereotypes to represent variability in use case diagrams, class diagrams, component diagrams, activity diagrams, and sequence diagrams [17]. As tool support, we can mention SMartyModeling[8] that is an environment for engineering UML-based SPLs in which variabilities are modeled using the SMarty approach.

18.5 Final Remarks

This chapter presented our approach to reengineer UML class diagram variants into PLAs. The approach is composed of two steps, in which firstly a model-based software architecture is discovered by merging UML model variants and secondly variability annotation is included based on traceability information. The first step relies on a search-based technique that does not require information regarding domain constraints or conflicting model elements in advance. The variability annotation is a basic matching between UML model elements and traces information, using the name of the features to include UML-owned comments.

We performed an evaluation of our approach with four case studies from different domains and of different sizes. The results show that our approach is able to find good PLAs even when features are spread across multiple variants. Furthermore, we could observe that having a variant that implements all features of a system does not imply that this variant has all model elements of all individual variants.

We acknowledge that some results could be influenced by internal aspects of the subject systems; however, our approach is an easy way to support the reengineering of UML model variants into PLAs. Furthermore, the PLAs found by using our approach can help practitioners during maintenance by (i) providing a global view of a set of variants that supports the identification of bad smells and refactoring activities, (ii) allowing design reconciliation of different variants (potentially inconsistent) implemented by many designers, and (iii) showing clearly which product will be affected by modifications, since commonalities and variabilities are explicitly

[8] https://github.com/leandroflores/demo_SMartyModeling_tool.

shown. The documented architecture supports evolution by (i) being a start point to transform artifacts into an SPL and (ii) reducing the time to produce products with a new combination of features.

Acknowledgments The work is supported by the Brazilian funding agencies CAPES and CNPq (Grant 305968/2018), by the Carlos Chagas Filho Foundation for Supporting Research in the State of Rio de Janeiro (FAPERJ), under the PDR-10 program, grant 202073/2020, and by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant RGPIN-2017-05421.

References

1. Assunção, W.K.G., Lopez-Herrejon, R.E., Linsbauer, L., Vergilio, S.R., Egyed, A.: Extracting variability-safe feature models from source code dependencies in system variants. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 1303–1310. ACM, New York (2015). https://doi.org/10.1145/2739480.2754720
2. Assunção, W.K.G., Lopez-Herrejon, R.E., Linsbauer, L., Vergilio, S.R., Egyed, A.: Reengineering legacy applications into software product lines: a systematic mapping. Empir. Softw. Eng. 1–45 (2017). https://doi.org/10.1007/s10664-017-9499-z
3. Assunção, W.K.G., Vergilio, S.R., Lopez-Herrejon, R.E.: Discovering software architectures with search-based merge of UML model variants. In: Botterweck, G., Werner, C. (eds.) Mastering Scale and Complexity in Software Reuse, pp. 95–111. Springer International Publishing, Berlin (2017)
4. Berger, T., Steghöfer, J.P., Ziadi, T., Robin, J., Martinez, J., et al.: The state of adoption and the challenges of systematic variability management in industry. Empir. Softw. Eng., 1755–1797 (2020)
5. Dobrica, L., Niemela, E.: A survey on software architecture analysis methods. IEEE Trans. Softw. Eng. **28**(7), 638–653 (2002). https://doi.org/10.1109/TSE.2002.1019479
6. Durillo, J.J., Nebro, A.J.: jmetal: a Java framework for multi-objective optimization. Adv. Eng. Softw. **42**, 760–771 (2011). https://doi.org/10.1016/j.advengsoft.2011.05.014. http://jmetal.sourceforge.net/
7. Faust, D., Verhoef, C.: Software product line migration and deployment. Softw. Pract. Exp. **33**(10), 933–955 (2003)
8. Garcia, J., Ivkovic, I., Medvidovic, N.: A comparative analysis of software architecture recovery techniques. In: International Conference on Automated Software Engineering (ASE), pp. 486–496. IEEE, Piscataway (2013)
9. Goldberg, D.E., Deb, K., Clark, J.H.: Genetic algorithms, noise, and the sizing of populations. Complex Syst. **6**, 333–362 (1992)
10. Holmes, R., Walker, R.J.: Systematizing pragmatic software reuse. ACM Trans. Softw. Eng. Methodol. **21**(4), 1–44 (2013). https://doi.org/10.1145/2377656.2377657
11. Hussain, I., Khanum, A., Abbasi, A.Q., Javed, M.Y.: A novel approach for software architecture recovery using particle swarm optimization. Int. Arab J. Inf. Technol. **12**(1), 32–41 (2015)
12. Jeet, K., Dhir, R.: Software architecture recovery using genetic black hole algorithm. ACM SIGSOFT Softw. Eng. Not. **40**(1), 1–5 (2015)
13. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2), 131–183 (1992). https://doi.org/10.1145/130844.130856
14. Kulkarni, N., Varma, V.: Perils of opportunistically reusing software module. Softw. Pract. Exp. **47**(7), 971–984 (2017). https://doi.org/10.1002/spe.2439
15. Laguna, M.A., Crespo, Y.: A systematic mapping study on software product line evolution: from legacy system reengineering to product line refactoring. Sci. Comput. Program. **78**(8), 1010–1034 (2013). https://doi.org/10.1016/j.scico.2012.05.003

16. Linsbauer, L., Lopez-Herrejon, E.R., Egyed, A.: Recovering traceability between features and code in product variants. In: 17th International Software Product Line Conference, SPLC'13, pp. 131–140. ACM, New York (2013). https://doi.org/10.1145/2491627.2491630
17. Marcolino, A.S., OliveiraJr, E.: Comparing SMarty and plus for variability identification and representation at product-line uml class level: a controlled quasi-experiment. J. Comput. Sci. **13**(11), 617–632 (2017). https://doi.org/10.3844/jcssp.2017.617.632
18. Martinez, J., Ziadi, T., Klein, J., Traon, Y.L.: Identifying and visualising commonality and variability in model variants. In: 10th European Conference Modelling Foundations and Applications (ECMFA), pp. 117–131 (2014). https://doi.org/10.1007/978-3-319-09195-2_8
19. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., l. Traon, Y.: Automating the extraction of model-based software product lines from model variants. In: International Conference on Automated Software Engineering (ASE), pp. 396–406 (2015)
20. OliveiraJr, E., Gimenes, I.M.S., Maldonado, J.C.: Systematic management of variability in uml-based software product lines. J. Univ. Comput. Sci., 2374–2393 (2010). https://doi.org/10.3217/jucs-016-17-2374
21. Pohl, K., Böckle, G., van Der Linden, F.J.: Software Product Line Engineering: Foundations, Principles And Techniques. Springer Science & Business Media, Berlin (2005)
22. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education, London (2008)