On the Interest of Combining Several Variability Tactics to Design the Implementation of Product Lines

Maouaheb Belarbi and Vincent Englebert[a]
NADI Research Institute, University of Namur, Faculty of Computer Science, Belgium

Keywords: Product Line, Variability, Mechanism, Implementing Variability.

Abstract: In Software Product Line (SPL) field, several variability mechanisms have been proposed to realize system requirements. Despite that each mechanism is relevant to satisfy only specific engineering criteria, it is unable to face several challenges. We believe that using several variability realization techniques allows to get trade-off for all the benefits and strength of the considered realization techniques. In this paper, we provide an answer for the fact that if combining several programming techniques is worth investment which, at the best of our knowledge, was not tackled before in the literature. The current study implements, at first, an open source product line entirely with different variability realization techniques, and secondly, combines these mechanisms in the same product line. An assessment of the obtained product lines is performed according to quality attributes,code quality metrics, and product line concepts. Together, the present findings confirm that combining several variability mechanisms hits the best rates in most quality evaluation metrics and provides a median to achieve most relevant software quality criteria.

1 INTRODUCTION

The growth of the enterprise application system in the last few decades has justified the need to industrialize the software development industry (Jalil and Bakar, 2017b). Software Factory (SF) approach provides a solution that can transform software development project into a more systematic approach imitating factory manufacturing concept (Jalil and Bakar, 2017a). SF was defined as a configuration of languages, patterns, frameworks, and tools that can be used to produce an open-ended set of variants belonging to a family products(Özgür, 2007). This last provides a context wherein many problems and requirements common to family members can be solved collectively. Building on System Product Line (SPL) concepts, SF exploits this context to provide family wide solutions, while managing variation among the family members (White et al., 2008). Instead of waiting for serendipitous opportunities for ad-hoc reuse to arise under arbitrary circumstances, an SF captures knowledge of how to produce the members of an SPL family.

In the design of complex and variable software systems, one of the key steps is to select the variability mechanisms that defines how features are realized in code level. Herein, several research seek to discuss the practical benefits and drawbacks of the mechanisms and present the cases of their successful use. SF emphasizes the use of the variability implementation mechanisms and tools to streamline software products development. We consider the categorization mentioned in (Kästner and Apel, 2008) which discusses the following mechanisms: Cloning, Conditional Compilation, Conditional Execution, Polymorphism, Module Replacement, Aspect Orientation, Frame Technology, and Design Patterns. Indeed, these variability mechanisms vary in *binding-time, granularity, quality criteria, etc*. For example, mechanisms with early binding-time as compile-time allow a well definition of features, however, mechanisms with late binding-time such as run-time enhance the system flexibility and adaptability but with a restricted behavior (Tërnava and Collet, 2017). Besides, the classification of mechanisms considers the separation of concerns to meet the increasing stakeholders and customers requirements. Here, industrial case studies showed that each mechanism hit advantages but brought challenges during each software development process (Zhang et al., 2016). In other words, each mechanism is relevant to satisfy only specific engineering criteria set but not all of them.

[a] https://orcid.org/0000-0001-8201-4294

Belarbi, M. and Englebert, V.
On the Interest of Combining Several Variability Tactics to Design the Implementation of Product Lines.
DOI: 10.5220/0011630700003402
In *Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2023)*, pages 105-116
ISBN: 978-989-758-633-0; ISSN: 2184-4348
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

We propose a SF which aims to bear programming challenges by combining a set of variability mechanisms to build software product systematically. We argue that using several variability realization techniques allows to get trade-off for all the benefits and strength of the considered realization techniques. In this paper, we provide an answer for the fact that if combining several programming techniques is worth investment which, at the best of our knowledge, was not tackled before in the literature. We propose to implement an open source Product Line (PL) entirely with different variability realization techniques. Thereafter,we implement the PL with a combination of mechanisms. An assessment of the obtained product lines is performed according to quality attributes, literature measures, and product line concepts.

The rest of the paper is organized as follows: section 2 explains the methodology and the real use case study. Section 3 clarifies the misunderstandings that may be confused with our proposal. Thereafter, section 4 is carried out to present quality metrics considered throughout the evaluation performed in section 5. Some potential weakness threatening our proposal are identified in the section 8 with some attempts to mitigate them. Finally, some of the relevant related work are presented in section 7 followed by a conclusion and some of our perspectives in section 9.

2 METHODOLOGY AND USE CASE STUDY

This section introduces the considered use case study to give insight the application of the presented approach as well as we the methodology adopted in this research.

2.1 The Elevator System

In traffic control of elevator systems, each one is ordered to move up or down, to stop or start, and to open or close the door. The following features were integrated into the base elevator system as shown in the feature model in Fig 1:

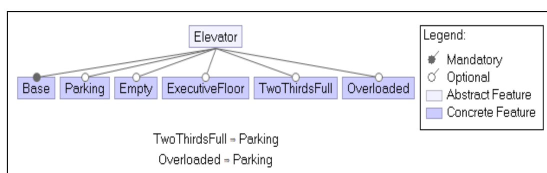


Figure 1: Feature Model of elevator system.

- **Parking.** When a lift is idle, it goes to a specified floor (typically the ground floor) and opens its doors. The parking floor may be different at different times of the day, anticipating upwards-travelling passengers in the morning and downwards-travelling passengers in the evening.
- **Lift 2/3 Full.** When the lift detects that it is more than two-thirds full, it does not stop in response to landing calls, since it is unlikely to be able to accept more passengers. Instead, it gives priority to passengers already inside the lift, as serving them will help reduce its load.
- **Overloaded.** When the lift is overloaded, the doors will not close and some passengers must get out to fulfill the movement requests.
- **Empty.** When the lift is empty, it cancels any calls which have been made inside the lift. Such calls were made by passengers who changed their mind and exited the lift early, or by practical jokers who pressed lots of buttons and then got out.
- **Executive Floor.** The lift gives priority to calls from the executive floor.

For comprehensive investigation we considered the Elevator PL case to meet different criteria: The elevator system describes a real world case study, which was developed in the context of an industrial project (Plath and Ryan, 2001) and published in (Meinicke et al., 2017) to provide an open Java source code [1]. In fact, real industry case applications raise the domain real relevant challenges. Hence, they present a way to check the usability and credibility of our approach when it crashes into reality. Elevator case provides support for SPL context since (i) several variability points are related to heterogeneous concepts of elevator systems and (ii) many alternative and optional functionality exist.

2.2 The Methodology of Study

The elevator system was adapted to SPL context and is available in FeatureIDE plugin with Java language. The elevator PL will be adapted again and to fetch the variability mechanisms illustrated in the right side of Fig 2. In other words, we provide several versions of the original elevator PL according to specific variability mechanism. All these versions [2] except the original PL are implemented by a middle-level developer. The resulting product lines will all be evaluated against literature quality attributes, industrial measures, and PL criteria. The assessment

[1] See http://spl2go.cs.ovgu.de/projects/16

[2] See https://github.com/Maouaheb/ElevatorsSystem.git

process confronts the statistics of both PL developed with only one realization technique and the PL built with several programming techniques. The choice of the involved techniques is justified as follows:

- **Strategy:** pattern is useful here because many related classes differ only in behavior. In this context, Strategy is applied to handle: elevator direction variation (up or down), the status (in service or blocked, stopped or in movement), a shift request called by an normal or executive floor, etc.
- **State:** pattern is not applied here to resolve a concept variability in the considered PL. However, a state machine is set up in the current use case to handle the following variability states: the elevator direction state (up or down), the elevator state (in service or blocked), the transition state (stopped or in movement), the shift request called by a normal or an executive floor, etc. Thus, we call the State pattern to hold the presented states and let them evolve over time. PL implemented with both State and Strategy patterns are similar since they converge to the same structure.
- **Aspect Oriented Programming (AOP):** is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does so by adding additional behavior to existing code (an advice) without modifying the code itself, instead separately specifying which code is modified via a "pointcut" specification. This allows behaviors that are not central to the business logic to be added to a program without cluttering the code core to the functionality. Considering the original version of elevator PL, we apply AOP here to add the executive floor specification, overloaded feature, and two thirds full elevator function.
- **Observer:** pattern defines an object called the `subject` which maintains a list of its dependents, called `observers`, and notifies them automatically of any state changes (Hunt, 2013). In case the elevator is overloaded or two-thirds full, the corresponding `overloaded` and `two-thirds` observers will be notified and executed.
- **Polymorphism:** is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Building elevator PL with polymorphism defines elevator super-type and sub-type classes, calling actions super-type class (i.e, to request call from executive floor), etc.
- **Combination:** The objective is to implement PL elevator with combination of the aforementioned mechanisms as follows: We consider *Strategy pattern* to resolve the elevator call actions when the executive floor is calling then the shift strategy gives priority to that request than the others. Since two shift elevator directions exist, hence, moving up or down is performed with Strategy pattern. Besides, *State pattern* is applied to realize the elevator state movements whereas the elevator is continuing towards the target direction or if it is stopping. Whenever, the maximum weight of persons inside the elevator exceeds the maximum weight threshold, then, an *Observer* is notified to stop the elevator movement and warn users. *Polymorphism* is used to resolve the call specification variability: if the user is of type Handicap, if the current floor is executive or normal, etc. Finally, when the elevator receives a shift call from a disabled passenger, then, an adapted behavior is set up to serve him. AOP is applied to inject specific shift aspect for disabled people.

The aforementioned elevator PL versions will be compared according to attributes quality, PL criteria, and industrial measures to confirm or refute the privileges of combining several variability mechanisms.

3 DISCLAIMER

In this section, we present the following delimitation that exceeds the objective of our study: First, we propose here an intuitive analysis to motivate combining different tactics together, however, this does not note a valid scientific conclusion to be admitted for all product lines. As well, the present paper does not present a guideline about how to combine and apply the selected variability implementation techniques. This task will be postponed over later.

Second, this study does not evaluate the performance of variability realization techniques individually or give a comparison between them. For example, comparing the PL versions implemented by State and Observer patterns does not assess the two aforementioned patterns but the performance and code quality of both product lines. It is an evaluation of how acts these mechanisms in the context of the elevator PL.

Finally, the mechanisms chosen were the most appropriate for the context of the elevator use case and for its original Java implementation. Thus, several tactics were dropped out such that conditional compilation which is unsuitable for the Java language.

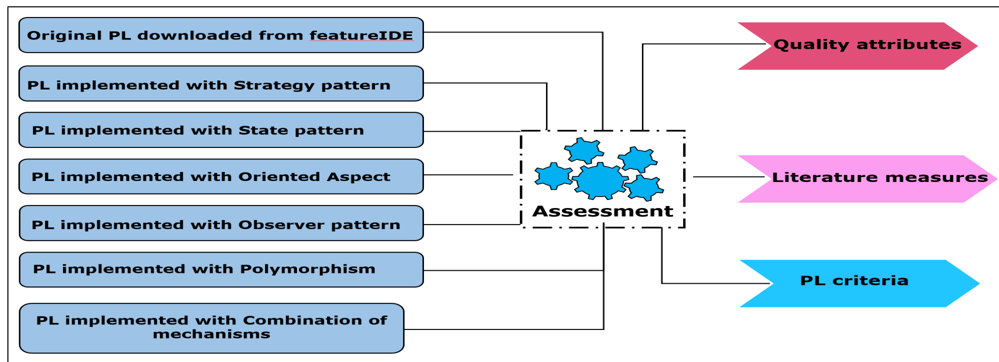


Figure 2: Overview of the methodology research study.

4 ASSESSMENT CRITERIA

In this section, we present the assessment criteria revealed in the literature and considered during this study. The evaluation and comparison are performed against: the software quality attributes, the industrial code metrics, and the PL criteria.

4.1 Quality Attributes

Within systems engineering, quality attributes deals with non-functional requirements to evaluate the performance and the software system quality (Bachmann et al., 2005). We believe that we cannot satisfy all of them, thus, we prioritize the followings:

Modifiability: is the degree of ease to carry out a change to a system and the flexibility experienced by this latter to adapt to these changes (Barbacci, 2004). We have planned the following three scenarios to explore the modifiability of the different PL versions source code:

- Add a function: Handicap call is a new function added to the system to ensure carrying disabled persons.
- Update existing function: Shift elevator function will be adapted to make the system suitable for disabled people.
- Delete function: checking if the elevator is two thirds full new requests will be ignored in order to avoid to reach the maximum weight. Deleting this feature implies that elevator continues treating calls till getting the maximum weight threshold. The reliability of the system is not threatened by the exclusion of this functionality which optimizes the operation but does not affect it.

Reusability: means that a segment of source code can be used again when adding new functionalities, with only slight or no modifications. Code reusability can also be achieved by inheritance as a derived class inherits its properties from parent class. Several existing metrics have been proposed to estimate reusability factor of software code. We consider the following metrics mentioned in (Padhy et al., 2018):

- Reusability estimation Rank metric (RR) is determined using the ratio of elements that are reused to the sum of all components available in the system. A proposal that reuse should be measured as the number of lines of code incorporated in a system without modification. *RR* is given as follows:

$$\boldsymbol{RR} = \frac{\text{reused elements}}{\text{sum of all elements}} \tag{1}$$

- Reusability of class: The reusability is addressed and calculated for each class according to three parameters: At first, Depth Inheritance Tree (DIT) signifies the highest length from the root class to the node across the tree. Secondly, Number Of Children (NOC) metric calculates the total number of adjacent sub-classes subsidiary to the considered class in the associated class hierarchy. Thirdly, Coupling Between Objects (CBO) measures, for each class, number of classes connected together. At this level, the reusability of class is calculated as follows:

$$\boldsymbol{Reusability} = X \times (DIT) + Y \times (NOC) + Z \times (CBO) \tag{2}$$

Where X, Y, Z are experimental constants. For expediency, we take X = Y = 1 and Z = 0.5. By convention in (Padhy et al., 2018), the class having the most reusability :

$$\boldsymbol{Class\ diagram\ Reusability} = Max(reusability(class_i)) \tag{3}$$

Where $i = 1,...,n$ and n defines the total number of classes.

Integrability: is about the degree to which an architect has anticipated and designed for integration tasks that the system may undergo to predict the costs and risks of integrating some units at some future point in time (Kazman et al., 2019). To evaluate this, we need to measure the potential dependencies between the system components. Code program graphs will be performed to model the code assets dependency in a graph.
Testability: is the degree to which a system or component facilitates the establishment of test criteria and the performance of tests (Tsai et al., 2006). Unit tests are applied to separate variants, whereas integration, system and acceptance tests are applied to a combination of bound variants. During this study, we check for each PL version which the test types that can be performed.

4.2 Code Quality Metrics

Based on different standards considered in the domain of evaluating code source quality, we have chosen a set of the most used metrics in source code evaluation (Boja et al., 2017). Table 1 describes the software metrics that are used for a static analysis. For each metric, we provide the description, the calculation formula, and the range of values. As showing in table below, Stability Index (SI) metric is calculated to assess system change and modification between different versions of same software (Koziolek et al., 2012). The higher the SI is, the maintainability of source code is considered better. Besides, McCabe complexity (M) metric is used to assess complexity and maintenance of software systems. This metric measures the maximum number of linearly independent circuits in a program control graph. Lower the cyclomatic complexity is , lower the risk to modify and easier to understand (Mccabe, 1996). Finally, we check considered engineering skills required to apply a mechanism which can be low, medium, or intense.

4.3 PL Criteria

Many research argue that SPL is an efficient approach to build software similar systems in lower effort (Mohammed et al., 2019). Therefore, if a PL version checks the following requirements, then, it shows proof for good quality:

- **Variability:** designates parts of software that remain flexible so that variations of the system will be supported by a single source with multiple instances (Metzger and Pohl, 2007). Hence, it is important to verify for each PL elevator version if the variability is explicit in source code or all common and variant parts are mixed together.
- **Reification:** designates that each feature presented in model level, such that the feature model, is retrieved directly in code level. Features can be reified to classes, attributes, methods, aspects, etc. The reification is crucial for the traceability and the ability to describe and follow the life of a requirement, and by the way, it is essential for product derivation and SPL evolution (Shen et al., 2009).
- **Extend Variability:** Since customer requirements and domain specifications evolve through the time, it is compulsory to be able to extend the behavior of a product by adding additional instructions or assets independently (Schmid and Eichelberger, 2017). Extending variability allows systems to evolve, keep up and satisfy customer requirements.
- **Size of Variability:** A variation point or variant in the core-code assets can have different sizes: an expression, statement, methods, classes, aspects, interfaces , packages, etc (Tërnava and Collet, 2017). Hence, variability size indicates to what extend the engineers can be involved.
- **Expressivity:** is the ability that a PL version can progress in the future.

5 THE EVALUATION OF ELEVATOR PL VERSIONS

The evaluation procedure is performed according to the three groups of quality characteristics. Code metrics will be considered to justify some quality attributes.

5.1 The Evaluation of Quality Attributes

Modifiability: The scenarios defined in section 4.1 were applied on the elevator PL systems to experiment the modifiability criterion when adding, updating, and deleting program parts in the PL code source. Table 3 sums up the observations noticed when performing this practical exercise. As shown, we needed to locate *where* are the instructions in which it was prone to add, update or delete the requirement. Hence, for each PL it was necessary to answer if locating the program part that ensures the concerned feature was direct to identify or if it was unavoidable to iterate through all the classes in the

Table 1: Code quality metrics description.

Metric	**Acronym**	**Description**	***Range***
Lines Of Code	LOC	Source code length as physical lines of code	***>0***
Comment Lines	CL	Number of commented lines	***>0***
Comment Density	CD	The percent of comment lines in a source code **CD=CL/LOC**	***<=0.2***
Cyclomatic complexity	M	McCabe cyclomatic complexity	***1-10***
Number of defined functions	FNUM	Total number of functions	***>0***
Number of calling functions	FCALL	Number of sub-functions called by function	***0-5***
Number of code lines per function	FSTM	Empty functions fall through	***1-50***
Number of modified statements	STMMOD	Number of statements in a function which have changed	***>=0***
Number of deleted statements	STMDEL	Number of statements in a function which have been deleted between the previous and the current version.	***>=0***
Number of new statements	STMNEW	Number of statements in function which have been added between the previous and the current version.	***>=0***
Stability Index	SI	Measures the number of changes between two versions of a software. **SI =(FSTM-(STMMOD+STMDEL+STMNEW))/FSTM**	***0-1***

Table 2: Index Stability Evaluation.

PL implemented with:	STMMOD	STMNEW	STMDEL	FSTM	**SI**
Original	0	30	0	71	0,57
Strategy	1	9	0	45	0,77
Polymorphism	1	30	0	65	0,52
State	1	6	0	25	0,76
AOP	0	6	0	36	0,83
Observer	0	9	0	31	0,70
Combination	0	6	0	36	0,83

program looking for it. Secondly, *components* designates which assets in the code level (i.e, class, instruction , aspects, interfaces,etc) are involved during the modification scenario. Finally, at what point it was necessary to propagate the modification applied on the existing parts *to rectify* them and ensure that the program keeps its consistency. In case of adding a new feature in the system according to the logic with which it was built, the criterion *behavior* was dedicated to determine in which PL version this function can be extended or not. Our findings hint that:

- *The Original PL Version :* in this PL all the requirements implementation were mixed together and IF-ELSE statements are used to specify which code will be executed. Hence, adding and deleting functions implies to iterate overall the components program to locate where to perform the update. Adding handicap call was spread over three other components, which is the highest number over the PL versions. Besides, the function is added by the same reasoning as the original implementation. Hence, it is tangled with other requirements which implies that the new feature is also hardly extensible in the future.
- *The AOP PL Version:* This PL gives better results when modifying its implementation. The components are directly located and easily extensible with low spreading effects.
- *The Polymorphism PL version:* Modification appeals to look for the super-type class responsible for adding and updating the elevator shift operation. However, deleting a feature from the parent class implies its automatic suppression from all the children classes.
- *The State and Strategy PL Versions:* Since these two design patterns are similar, they present same findings in terms of modifiability. Components are directly located via the Strategy (resp. the State) classes which implies that contributions are easy to carry. The deletion removes the class conceived for the two thirds full feature.
- *The Observer PL Version:* Here, the handicap call function is injected directly when encountering a call of type Handicap. The behavior is extensible and separated with other program assets. Finally, the deletion consists in suppressing the responsible observer and the instructions to attach it to the subject class (the elevator class).
- *The Combination of Mechanisms PL Version:* Here, the feature call handicap was ensured by AOP and the two thirds full was performed with strategy pattern to select shift operation algorithm. Hence, the findings here ties well with previous results presented by the AOP PL version to add and update a feature and Strategy PL version to delete function. From these results it is clear that the PL version performed with combination of mechanisms is easily modifiable: Adding Hand-

icap call function and update the elevator shift operation are performed directly and without supplying the broadcasting of change over several components. Finally, to eliminate the two thirds full feature, we delete directly its implementation class, and the instructions to instantiate it as a strategy variant.

Testability: Overall, all the elevator PL versions support the unitary tests performed with JUnit plugin. Besides, integration tests were carried out on the PL versions to test modules integration. Original PL version has been delimited for this kind of tests since both separation of concerns and modularization are missing off.

Reusability: Calculating the reusability metrics given by the equations 1 and 2 have led to the results given in the table 4. We also calculate the reusability average rate of code source by the quotient of the sum of reusability of each class by the total number of classes in the project. The higher the reusability rank and reusability average are, the more PL is qualified to be reusable. From these results, it is clear that the original version presents the lowest rate of reusability which is not surprising since the features are mixed in same classes and it is difficult to make an object apart from its context. Besides, the implementation of a same concern is scattered and tangled over the code source. Hence, reusing existing assets is hardly performed. Contrarily to the original version system, the elevator PL implemented with Polymorphism presents the better reusability rank and the second best reusability average. A popular explanation of that is that Polymorphism paradigm was created to this purpose. Combining several mechanisms in the same code source presents also best results for reusability rank and reusability average. It is important to highlight the fact that separating and modularizing components with patterns, AOP, Design Patterns, and Polymorphism principles ensures the program parts to be more reusable and mature in order to enhance software evolution despite the order of magnitude.

Integrability: To be able to integrate modules of a system in another one, we evaluate each PL integrability according to criterion mentioned in table 5. First of all, Separation of concerns (SoC) is to decompose and organize software into more comprehensive parts in order to improve software quality. Secondly, if a system is well modularized it is easier to change it and also to evaluate the side effects of a change (Li et al., 2021). Modularization evaluation is built on the assessment of coupling and cohesion in code source, where, coupling is the degree of dependency between the modules, whereas, cohesion is the inter-dependency within a single module. Finally, each PL will be qualified if it reduces or enhances the dependency comparing it to the original implementation source code dependency level. From the short review above, key findings emerge followings:

The PL elevator implemented with respectively AOP technique, Strategy, State, and Observer patterns present a solution to resolve SoC, provide high level of modularization and a ensure low dependency between system components. Developing elevator PL with a combination of mechanism yielded to similar last integrability qualification. In this PL version AOP and design patterns are required to verify claims concerning separating concerns into independent elements. In addition, gross decomposition of a system by patterns and AOP into interacting components using proper abstractions for component interaction defines the modularity of the system which can be understood, reused, and modified independently, without regard for where the code is used.

Finally, another promising finding shows that elevator PL realized with Polymorphism resolves SoC by separating concepts in super-types and sub-types which reduces interring concepts inside the same java construct and enhances the modularizaion level.

5.2 The Evaluation of Code Quality Metrics

We start by evaluating the SI of the PL versions. In table 2 we show that the combination of mechanisms provides the highest value of SI and hence a better quality of code. This confirms that the corresponding PL is the easiest version to maintain and understand which confirms our previous observations for the Modifiability quality attribute. The evaluation of the rest of code metrics leads to results shown in table 6. We explore the applicability of undirected graphs in understanding source code nuances: In Fig 3, we modeled the program units (classes, interfaces,etc) by nodes and the dependencies between program parts with edges. Based on these graphs we calculate the average dependecy, the component number, and the McCabe cyclomatic complexity.

Results provide a good fit to the observations noticed when evaluating quality attributes: elevator PL versions implemented with State and Strategy patterns and the combination of mechanisms present the lowest rate of dependency. In addition, in the line of combining variability we notice that despite the corresponding PL shows almost same results as Strategy and State PL versions, it ensures the lowest number of code components. The idea here, is to demonstrate that using several realization techniques

Table 3: Evaluation of Modifiability PL versions.

PL implementation tactic	Add and update handicap call function				Delete two thirds full feature		
	Where	New components	To rectify	Behavior	Where	Components	To rectify
Original	iterate	class handicap	3	hardly extensible	iterate	delete instructions	nothing
AOP	direct	class handicap and aspect	1	easily extensible	direct	aspect	nothing
Polymorphism	iterate	class handicap	2	extensible	iterate	delete from super-type	sub-type rectified automatically
Strategy	direct	class handicap, handicap-shift strategy class, call-handicap strategy class	1	easily extensible	direct	class of the feature	instructions to call the strategy class
State	direct	class handicap, handicap-shift state class, call-handicap state class	1	easily extensible	direct	class of the feature	instructions to call the state class
Observer	direct	class handicap, Observer shift-handicap	1	easily extensible	direct	class of the feature	instructions to attach the observer class
Combination	direct	class handicap, aspect shift-handicap	1	easily extensible	direct	class of the feature	instructions to call the strategy class

Table 4: Reusability evaluation for the implemented PL versions.

PL version implemented with:	**NOC**	**DIT**	**CBO**	**Reusability Max**	**Reusability average**	**RR**
Original	1	2	2	4	2.4	0.09
AOP	5	6	3	12.5	3.25	0.11
Polymorphism	1	2	4	5	3.11	0.36
Observer	1	2	2	4	2.71	0.10
Strategy	6	7	2	14	3.45	0.20
State	6	7	1	13.5	3.10	0.22
Combination	6	7	2	14	3.20	0.23

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 5: Integrability evaluation for the implemented elevator PL versions.

PL version implemented with	**SoC**	**Modularity**	**Dependency**
Original	low	low	high
AOP	valid	high	reduced
Polymorphism	valid	high	reduced
Strategy	valid	high	reduced
State	valid	high	reduced
Observer	valid	high	reduced
Combination	valid	high	reduced

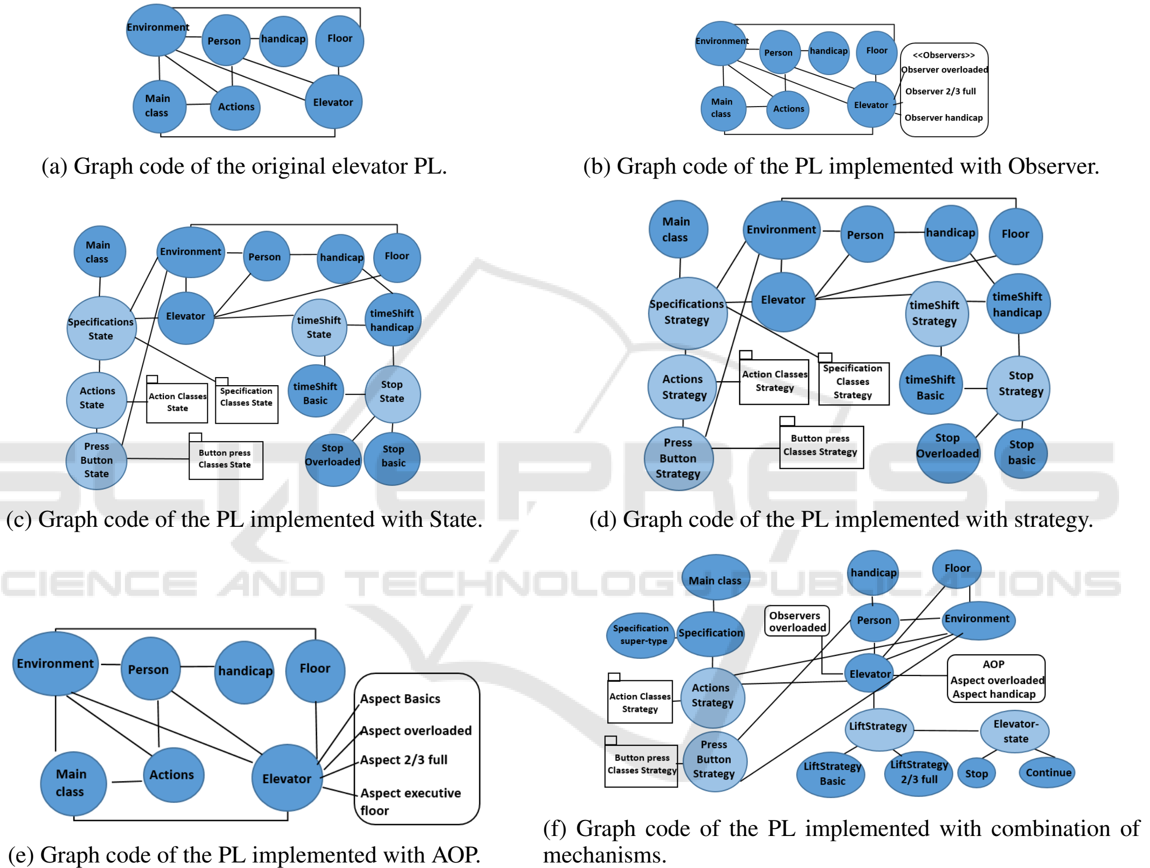
together achieves the median trade off between most important software quality criteria. Besides, analyzing code-graphs allows to determine the number of complete sub-graphs inside each model. Since in a complete graph vertices are all adjacent, meaning that each node is linked to all others. We inferred by this metric how many there are program parts that dependent all to each other. This metric witnesses the reusability and integrability of programs. We found that the Polymorphism PL has the lowest number of complete sub-graphs followed by the PL implemented with combination of mechanisms. This confirms our interpretations for the reusability attribute through the practice exercise to emphasize that combining mechanisms ensures reusability.

Another promising finding was that McCabe cyclomatic complexity (M) manifests itself in the lowest rate for PL version implemented with combination of variability mechanisms. As we mentioned before, this metric used for identifying heuristically subprograms which might be regarded as `<<overly complex>>`. Therefore, the lower M metric is, the better quality the corresponding PL will be. These results lead to similar conclusion when evaluating integrability quality attributes. We confirm here that combining several mechanisms together to implement the elevator PL have led to the best integrability rate assessment by practice exercise and with calculation method.

On another flap, the calculation of the code quality measures that are still used in the literature have shown the followings: PL version implemented with AOP presents the highest number of LOC which can be explained by the fact that AOP adds supplement modules to the original implementation. Another im-

Table 6: Evaluation of code quality metrics.

PL version implemented with	Average dependency	Number components	Number of complete sub-graph	M	LOC	CL	CD	FNUM	FCALL	Ability
Original	0.55	6	6	6	670	22	0.03	105	55	low
Strategy	0.061	34	3	4	650	32	0.04	93	76	low
Polymorphism	0.3	9	1	4	510	23	0.04	87	61	low
State	0.061	34	3	4	675	41	0.06	97	81	low
AOP	0.28	10	6	6	909	82	0.09	102	53	high
Observer	0.32	9	6	6	610	31	0.05	98	77	medium
Combination	0.07	29	3	3	608	37	0.06	97	77	medium



(a) Graph code of the original elevator PL.

(b) Graph code of the PL implemented with Observer.

(c) Graph code of the PL implemented with State.

(d) Graph code of the PL implemented with strategy.

(e) Graph code of the PL implemented with AOP.

(f) Graph code of the PL implemented with combination of mechanisms.

Figure 3: Graph code for PL code sources.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

portant witness is that when using a mixture of variability mechanisms have not led to a huge program size since its corresponding LOC metric is still up in standard with the other PL versions.

5.3 The Evaluation of PL Criteria

In this part, we illustrate some experimental results for the PL criteria shown in Table 7. The evaluations reveals the following finding:

The original PL version presents several gaps to explicitly identify variability, and recognize the corresponding program parts for each feature in the feature model. PL versions implemented with Strategy and State patterns explicitly define variability inside code where Variants are identified by sub-classes inheriting from the variation point class. The modularization of the code allows variability to be extended and hence the PL to evolve. Using Polymorphism to realize elevator PL defines explicitly variability by considering mother classes as variation points and children by variants. However, features can be amalgamated inside the same class, hence, the reification criterion is not always respected. In addition, manipulating super

types classes and having deep inheritance trees make the PL difficult to progress in the future. Moreover, our findings hint that implementing PL elevator with AOP and Observer pattern does not specify explicitly variation points and variants except if we consider an abstract aspect (resp. observer class)and children aspects (resp. observer classes) extending it.

Finally, combining variability mechanisms in a PL allows to define variability explicitly, recognize each feature in the feature model and its program assets in code level since it enhances both modularization and SoC. In addition, as we have illustrated in the superior evaluations, this PL version is easily modifiable which ensures that extending variability is easy to perform and can evolve to be suitable with the future challenges.

6 SUMMARY OF RESULTS

Overall, the elevator PL version realized with a set of variability mechanisms techniques presents the robustest results and hits the best rates in most of evaluation experiments: Assessing modifiability by the practice exercise showed that the combination of mechanisms achieves the lowest cost of change. Besides, by calculating the SI, we found evidence that combining several realization techniques records the highest SI and reusability values as shown in Fig 4 which confirms that this PL version presents the easiest way to accommodate changes. In addition, this PL version is considered to have less cohesion and dependency between modules with best of SoC criteria which was hardened by calculating the cyclomatic complexity and dependency average since we get the lowest rate as shown in Fig 5.

Moreover, despite that getting best evaluation results was not always achieved, we obtain always values among the best rates and we provide the median of the most important quality criteria. For example, using strategy pattern presents the lowest dependency average followed by the PL with combination of mechanisms which provides lowest number of program components. Hence, we assume that our proposal yields to satisfy as much as possible of quality criterion even if there is a negligible decline on a level.

Finally, we find that combining mechanisms provides a better quality while remaining within the standard size of the software since we had acceptable values for LOC, CL, FCALL, FNUM metrics.

7 RELATED WORK

Actually, we argue that in the literature, combining several variability techniques to implement PL is still in infancy. Hence, in this section we sum up most notable relevant research to implement variability in SPL field. Most of implementation mechanisms found in the literature were usually classified as compositional or annotative approaches (Kästner and Apel, 2008). Compositional approaches implement features as distinct code units and compose them usually at compile-time or deploy-time. Whereas, annotative approaches implement features by planting implicit or explicit annotations in the source code.

Clone-and-own: is a the easiest way of producing a variant of a certain software product by copying a code or non-code artifact and evolving it further without keeping connection with original version (Dubinsky et al., 2013). This leads to different problems, especially for software maintenance due to a merging process that is high cost and unscalable for a large number of products.

Conditional Compilation: is one of the most widely annotative techniques using preprocessors to conditionally include or exclude variability with `#ifdef` annotations which is easy-to-use (Couto et al., 2011). However, variability is not separated from the whole code and is explicit to identify. Besides they received severe criticism because the code is usually complex and hard to understand since `#ifdef` statements are nested with a complex structure.

Conditional Execution: is an another annotative approach that aims to realize variability by coding it explicitly using conditional `if-else` code parts (Meinicke et al., 2017) to enable or disable features at runtime. Therefore, it provides high flexibility to adapt the system to unforeseen requirements. However, it presents several challenges: Variants must be fine-grained to be implemented and it is hard to distinguish between variation logic and code functionality because they are nested together. Finally, the compilation speed and variation at run-time are degraded because of the inclusion of all variant elements from code compilation until running.

Polymorphism: is used when an overridden method is called through a reference of parent class, then type of the object determines which method is to be executed (Géraud et al., 2001). Thus, this determination is made at run time. As a compositional approach, variability is separated in different files from common features. However, the adoption of Polymorphism in practice increases the risk of software defects at runtime errors such as illegal pointers.

Table 7: PL criteria evaluation for the implemented PL versions.

PL version implemented with:	Variability	Reification	Extend variability	Variability size	Expressivity
Original	No explicit	No	No	Instructions, Methods	No
Strategy	Explicit	Yes	Yes	Instructions, Methods, Classes	Yes
Polymorphism	Explicit	No	Yes	Instruction, Methods, Classes	Not easy
State	Explicit	Yes	Yes	Instruction, Methods, Classes	Yes
AOP	No explicit	Yes	Yes	Instruction Methods, Classes, Aspects	Yes
Observer	No explicit	Yes	Yes	Instruction, Methods, Classes	Depends
Combination	Explicit	Yes	Yes	Instruction, Methods, Classes, Aspects	Yes

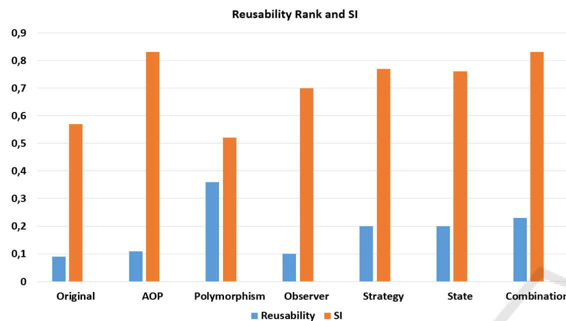


Figure 4: Graphic of SI and Reusability.

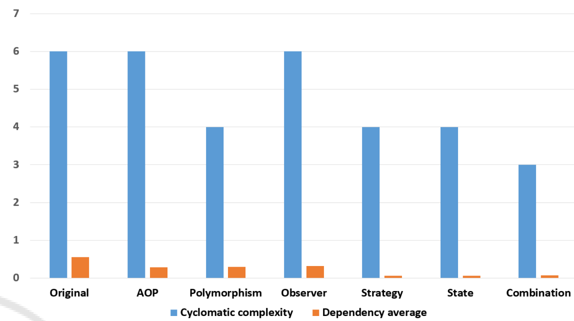


Figure 5: Graphic of Cyclomatic complexity and dependency average.

Aspect-Oriented-Programming (AOP): relies on code weaving techniques that require external tool support such as AspectJ (Zhang et al., 2016). As a compositional mechanism, it separates common and variable features into separate files. Depending on the aspect weaver, variability can be resolved at both compile-time and run-time. Despite the benefits above, AOP is not supported natively by programming languages and therefore it is difficult to apply it rapidly in development process without learning efforts and it increases code size which affects code comprehensibility (Fazal-e Amin and Oxley, 2010).

Software design pattern (Guizzo et al., 2019) is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations.

8 POTENTIAL WEAKNESS

Since any study has flaws, we identify the following weakness points: The literature presents infancy in providing PL implemented with different programming techniques, hence, we considered a use case presented by the FeatureIDE plugin and we adjusted it to fit the different tactics. Intuitively, the engineer must choose the most suitable programming tactics for the features. Choosing the right choice of mechanism is necessary to obtain a software with highest quality. This cannot be avoided as this is an immediate consequence of the engineer competence.

Finally, using some quality attributes plugins can afford more precise values and different quality metrics besides to some mathematics analysis diagrams of program dependency.

9 CONCLUSION AND FUTURE WORK

This paper provides a practice exercise to illustrate that combining several variability mechanisms is worth the investment. The obtained PL fits well most of the quality attributes, the code metric, and the SPL criteria. Besides, it achieves the median trade off between most important software quality criteria which is a challenging task.

As a future work, we plan for extending this approach and identifying the dependencies between different programming tactics to be able to employ all of them automatically by the software factory framework and generate software products.

REFERENCES

Bachmann, F., Bass, L., Klein, M., and Shelton, C. (2005). Designing software architectures to achieve quality attribute requirements. *IEE Proceedings-Software*, 152(4):153–165.

Barbacci, M. (2004). Software quality attributes: modifiability and usability. *Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA*, 15213.

Boja, C., Madalina, Z.-D., Marius, P., and Cristian, T. (2017). Code quality metrics evaluation platform in software engineering education. In *Proceedings of the 16th International Conference on INFORMATICS in ECONOMY (IE 2017), Education, Research & Business Technologies, ISSN*, pages 2284–7472.

Couto, M. V., Valente, M. T., and Figueiredo, E. (2011). Extracting software product lines: A case study using conditional compilation. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 191–200. IEEE.

Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., and Czarnecki, K. (2013). An exploratory study of cloning in industrial software product lines. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 25–34. IEEE.

Fazal-e Amin, A. K. M. and Oxley, A. (2010). A review on aspect oriented implementation of software product lines components. *Information Technology Journal*, 9(6):1262–1269.

Géraud, T., Demaille, A., and Duret-lutz, R. (2001). Design patterns for generic programming in c+. In *In In the Proceedings of the 6th USENIX Conference on ObjectOriented Technologies and Systems COOTS*. Citeseer.

Guizzo, G., Colanzi, T. E., and Vergilio, S. R. (2019). Applying design patterns in the search-based optimization of software product line architectures. *Software & Systems Modeling*, 18(2):1487–1512.

Hunt, J. (2013). Gang of four design patterns. In *Scala design patterns*, pages 135–136. Springer.

Jalil, D. and Bakar, M. S. A. (2017a). Adapting software factory approach into cloud erp production model. *International Journal of Computer Science and Information Security*, 15(1):221.

Jalil, D. and Bakar, M. S. A. (2017b). Enabling software factory with job workflow automation. *IJCSIS*, 15(4).

Kästner, C. and Apel, S. (2008). Integrating compositional and annotative approaches for product line engineering. In *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40.

Kazman, R., Bianco, P., Ivers, J., and Klein, J. (2019). Integrability. Technical report, Carnegie-Mellon University, Software Engineering Institute Pittsburgh United ....

Koziolek, H., Domis, D., Goldschmidt, T., Vorst, P., and Weiss, R. J. (2012). Morphosis: A lightweight method facilitating sustainable software architectures. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 253–257. IEEE.

Li, Y., Ni, Y., Zhang, N., and Liu, Z. (2021). Modularization for the complex product considering the design change requirements. *Research in Engineering Design*, pages 1–16.

Mccabe, T. (1996). Cyclomatic complexity and the year 2000. *IEEE Software*, 13(3):115–117.

Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., and Saake, G. (2017). *Mastering software variability with FeatureIDE*. Springer.

Metzger, A. and Pohl, K. (2007). Variability management in software product line engineering. In *29th International Conference on Software Engineering (ICSE'07 Companion)*, pages 186–187. IEEE.

Mohammed, L. A. A. et al. (2019). *A Customized Quality Model To Evaluate Domain Engineering In Software Product Line*. PhD thesis, Sudan University of Science & Technology.

Özgür, T. (2007). Comparison of microsoft dsl tools and eclipse modeling frameworks for domain-specific modeling in the context of model-driven development.

Padhy, N., Singh, R., and Satapathy, S. C. (2018). Software reusability metrics estimation: algorithms, models and optimization techniques. *Computers & Electrical Engineering*, 69:653–668.

Plath, M. and Ryan, M. (2001). Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84.

Schmid, K. and Eichelberger, H. (2017). Variability modeling with easy-producer. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*, pages 251–251.

Shen, L., Peng, X., and Zhao, W. (2009). A comprehensive feature-oriented traceability model for software product line development. In *2009 Australian Software Engineering Conference*, pages 210–219. IEEE.

Tërnava, X. and Collet, P. (2017). On the diversity of capturing variability at the implementation level. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*, pages 81–88.

Tsai, W.-T., Gao, J., Wei, X., and Chen, Y. (2006). Testability of software in service-oriented architecture. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 2, pages 163–170. IEEE.

White, J., Schmidt, D. C., Benavides, D., Trinidad, P., and Ruiz-Cortés, A. (2008). Automated diagnosis of product-line configuration errors in feature models. In *2008 12th International Software Product Line Conference*, pages 225–234. IEEE.

Zhang, B., Duszynski, S., and Becker, M. (2016). Variability mechanisms and lessons learned in practice. In *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*, pages 14–20.