On the Implementation of Dynamic Software Product Lines: A Preliminary Study

Michelle Larissa Luciano Carvalho*, Gecynalda Soares da Silva Gomes†, Matheus Lessa Gonçalves da Silva*, Ivan do Carmo Machado*, Eduardo Santana de Almeida *
* Computer Science Department, Federal University of Bahia (UFBA), Salvador - BA, Brazil
† Statistic Department, Federal University of Bahia (UFBA), Salvador - BA, Brazil
Email: [michellellcarvalho, mlessadev]@gmail.com, gecynalda@yahoo.com, [ivanmachado, esa]@dcc.ufba.br

***Abstract*—Dynamic Software Product Lines (DSPL) engineering has emerged as a promising strategy to develop Software Product Lines (SPL) that incorporate reusable and dynamically reconfigurable artifacts. The central purpose of DSPL is to handle adaptability at runtime through variability management, as well as to maximize the reuse of components. Emerging domains such as the mobile applications and software-intensive embedded systems require changes and extensions to the design in terms of both functionality and adaptation capabilities. DSPL should also deal with the unavoidable changes, which reflect both user needs and execution environments. However, the evolution capability in DSPL so far has not been investigated in depth. In this paper, we report on an exploratory study aimed at evaluating the object-oriented and aspect-oriented solutions on DSPL evolutionary scenarios. In this empirical evaluation, the aspect-oriented solution yielded better results in terms of measurements such as Weighted Operations per Component (WOC), Lines Of Code (LOC), Lack of Cohesion Over Operations (LCOO), Coupling between components (CBC), and Response For a Class (RFC). The use of aspects indicates that it provides assets with lower complexity, lower coupling, and higher cohesion.**

***Keywords*—Dynamic Software Product Lines, Dynamic Variability, Self-adaptive Systems, Software Evolution, Exploratory Study, Software Metrics**

I. Introduction

The Dynamic Software Product Lines (DSPL) approach aims to develop self-adaptive systems using the Software Product Lines (SPL) engineering principles [19]. Researchers have proposed this new approach as a promising strategy to deal with reconfiguration at runtime.

The DSPL approach enables software engineers to identify reusable and dynamically reconfigurable artifacts at development time, which are explicitly modeled and developed as dynamic variability. Dynamic variability occurs due to product variations that appears in the execution environment. These variations depend on the context variations [1]. In this sense, the SPL models and variability management practices are used in order to deal with the design and implementation of DSPL, which propose to configure and reconfigure instances by the variability customization at runtime [29].

The variability can be defined as the capability to change or customize a system [24]. Variabilities are specified through features. A feature consists of a prominent or distinctive user-visible element, quality or characteristic of a system, which can be classified as: *(i)* `mandatory` features represent the common functionality that must be present in all products of the family; *(ii)* `or` features group allows the selection of one or more features of this group; *(iii)* `alternative` features are model mutual-exclusive functionality; and *(iv)* `optional` features represent a functionality that may be part of a product [22].

The variability in DSPL can be represented through dynamic features. These features can be activated, deactivated, or updated at runtime. Thus, DSPL provides adaptable and highly configurable features [3]. However, due to the lack of suitable mechanisms and paradigms to support the development of DSPL, its design is may become more complex [21]. In addition, the inherent changes may significantly influence DSPL settings, particularly in terms of existing components and adaptation models [27]. In this sense, every new feature added should be mapped to existing artifacts, thus minimizing the ripple effects of changes.

Existing research usually focuses on modeling variability [5, 20]. Nevertheless, to the best of our knowledge, there are no publications reporting on empirical studies that quantitatively assess the impact of different solutions under software quality in DSPL evolutionary scenarios. This is a particularly important gap to bridge. Since some studies have demonstrated the likely synergies between Aspect-Oriented Programming (AOP) and DSPL [26, 33], we decided to investigate how object-oriented (OO) and aspect-oriented (AO) solutions can affect factors related to code quality in a DSPL project.

The analysis comprised the following measures: *size, cohesion, coupling, design stability*, and *change propagation impact*. The results pointed out few differences between the solutions. In general, the AO solution yielded better results in measures such as *size, cohesion, and coupling*. In this effect, there is an indication that AOP may be a feasible strategy for DSPL implementation.

The remainder of this paper is organized as follows. Section II discusses related work. Section III presents the exploratory study definition. Section IV describes the exploratory study planning. Section V describes the analysis and the interpretation of the results. Section VI reports the lessons learned. Section VII discusses the threats to validity. Section VIII draws concluding remarks and points out future research directions.

978-1-5090-5086-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SBCARS.2016.13

II. Related Work

The evolution of DSPL regarding unexpected software changes has received attention from researchers investigating variability management. However, only a few experience reports present suitable mechanisms to implement dynamic variability [4]. Moreover, several studies address variability modelling [11], nevertheless, they have different focus since they do not describe the implementation of a DSPL application.

Evolution has been widely studied in the SPL field. Most existing work has focused on the evolution of *problem space* [31]. However, evolving a variability model may also affect the *solution space* and vice versa. The problem space refers to the system's specifications established during the domain analysis and requirements engineering phases, whereas the solution space refers to the related assets created during the architecture, design, and implementation phases [2].

In contrast, existing research only recently started to investigate evolution in the DSPL field, and especially its impact on the running system. Evolving DSPL poses significant challenges as both problem and solution spaces. The evolution of problem or solution spaces can lead to inconsistencies within the given space, between spaces, and with respect to rules for the runtime adaptation of the system [27].

Talib *et al.* [32] present a classification of required operations for jointly evolving problem and solution space in a DSPL. However, they use the general term variability model to describe any model of the variability of a software system. In addition, they analyzed the impact of evolution operations on the consistency of the DSPL and an architecture of a tool-supported approach that addresses some issues and supports the evolution of DSPL.

They presented some requirements for DSPL and categorized them in terms of dynamic reconfiguration and evolution. However, these requirements do not take into consideration design quality. The DSPL approach offers automated product reconfiguration capabilities but are not evolvable in the sense that they lack support for unanticipated change. In general, the evolution in this context encompasses addition and update of features or maintenance tasks. In this way, it is important to investigate these issues in more detail. Thus, our work may provide an important contribution to the DSPL engineering field.

Some works use AOP to improve *(i)* the isolation of specific features, *(ii)* increase the code quality, and *(iii)* reduce the impact of using *aspects* in conventional SPL evolutionary scenarios [14] and designs of single systems [17]. They report quantitative assessment targeted at maintenance and reuse scenarios and software design stability. However, they did not assess OO and AO implementations by considering crosscuting concerns and evolution scenarios in the DSPL field.

Since dynamic adaptation is often a factor that crosscuts the application logic, it is important to investigate the existing paradigms that offer language abstractions to cope with crosscuting concerns. We believe that this assessment is essential to achieve a higher degree of maintenance, reuse, and robustness robustness DSPL demands.

III. Exploratory Study Definition

This section presents the exploratory study design. It was conducted in an academic environment at Federal University of Bahia, Brazil. In addition, the OO version of the DSPL was developed from scratch. Then, we proceeded with the modelling and the implementation of the AO version, based on the former.

A. Objective

The study is aimed at **analyzing** the aspect-oriented and object-oriented solutions to implement dynamic variability **for the purpose of** evaluation of the source code with respect to its *size, cohesion, coupling*, and *instability* **from the point of view of** Software Engineers and researchers **in the context** of an evolving DSPL.

B. Research Questions (RQs)

RQ1. **How complex is the design of AO and OO solutions in DSPL?**

The software *complexity* is related to the required maintenance effort as a result of modifications as well as understandability issues. Large and highly complex components lead to difficulties in terms of understanding and maintenance.

RQ2. **How cohesive is the design of AO and OO solutions in DSPL?**

Cohesion refers to the degree that operations of a component belong to another component, *i.e.*, it measures the interdependence of a component. High cohesion is generally used in support of low coupling. High cohesion means that the responsibilities of a given element (*e.g.*, a software component) are strongly related and highly focused [23]. The reuse, maintenance, and extensibility are the advantages achieved with highly cohesive components.

RQ3. **How coupled is the design of AO and OO solutions in DSPL?**

Coupling describes the relationship or dependency between two or more components. The more coupling between any two components, more difficult it is for a programmer to comprehend a given component. A system with high coupling means there are strong interconnections between its modules. As a consequence, high coupling may lead to difficulties in terms of maintenance and reuse. Thus, the coupling factor can directly impact the software maintenance activities.

RQ4. **How different is the stability measurement when comparing AO and OO solutions in DSPL?**

It is possible to evaluate the design stability of a software system by comparing and analyzing data from different versions of the system [14]. Whether the differences among its quality measures are insignificant, then the software design is stable.

RQ05. **How different is the change propagation impact measurement when comparing AO and OO solutions in DSPL?**

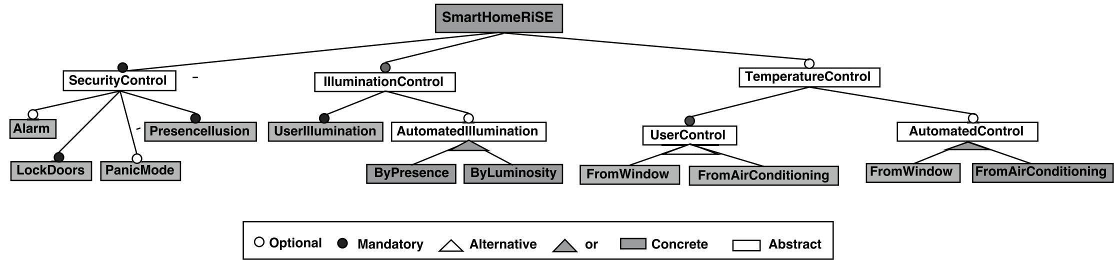

Fig. 1: SHR Feature Model

Change is an inherent aspect of software development. Changes are usually inevitable and thus controlling them is the most important part of maintenance tasks. The design stability is directly related to effects of the changes. The variability mechanisms should provide support to extend the system functionality.

C. Metrics

The metrics are described in terms of class, aspects, methods, and advices. Whereas the components encompass class, interfaces, and aspects, the methods and advices are called operations. Due to similarities among its constructions, class, aspects, methods, and advices are treated and measured in the same way during the assessment.

M1. **Weighted Operation per Component (WOC):** The WOC metric was used to help answering the questions *RQ1* and *RQ4*. This metric measures the complexity of a component based on its operations [9].

M2. **Number of Lines Of Code (LOC):** The LOC metric was used to help answering the questions *RQ1* and *RQ4*. This metric counts the number of lines of code, and it is a traditional measure of size [12].

M3. **Lack of Cohesion Over Operations (LCOO):** The LCOO metric was used to help answering the questions *RQ2* and *RQ4*. This metric measures the lack of cohesion of a component [28].

M4. **Coupling Between Components (CBC):** The CBC metric was used to help answering the questions *RQ3* and *RQ4*. This metric counts the number of other components to which it is coupled [28].

M5. **Response For a Class (RFC):** The RFC metric was used to help answering the questions *RQ3* and *RQ4*. The RFC metric is a class coupling measure [9].

M6. **Traditional change impact metrics:** Considering different levels of granularity – components, operations, and lines of code [17], these metrics were used to help answering the questions *RQ4* and *RQ5*. They measure the number of lines of code, operations, and components added and removed.

IV. Exploratory Study Planning

This section discusses the planning and procedures used to perform the exploratory study.

A. Project

In this exploratory study, we used a self-adaptive and context-aware system in the domain of smart homes, named `SmartHomeRiSE` [1] (SHR). The SHR DSPL provides convenience for users in terms of control the luminosity, temperature, and security based on its automation functions [30].

Figure 1 shows the variability model of the SHR project. The project encompasses the two essential DSPL activities [25]: *monitoring* the current situation for detecting events that might require adaptation and *controlling* the adaptation through the variability management. In addition, it encompasses the following set of non-functional properties: *adaptability*, *autonomy*, *context-sensitivity*, and *autonomic or self-adaptive decision-making*. A smart home provides its residents with a more comfortable, safe, energy-efficient, and all-times-convenient environment, which makes their lives easier.

A range of studies in DSPL engineering have employed smart homes systems in their investigations, as they might support a diversity of product configurations at runtime, thus being suitable to represent the challenges this field faces [6, 7, 8]. Methods and techniques to support smart homes implementation have been proposed and investigated [10, 34], but key issues such as reuse, evolution, and maintenance have been left aside.

B. Quantitative Analysis Mechanisms

The exploratory study was organized in the following steps: *(i)* identification and selection of suitable implementation mechanisms to implement DSPL variability; *(ii)* definition of the application domain, feature model, reference architecture, reconfiguration rules, and possible adaptations; *(iii)* building a prototype which encompasses the scaling model and devices (*e.g.* sensors, micro-controllers, and actuators); *(iv)* development of the subject DSPL with complete releases; *(v)* measurements and metrics calculation; and *(vi)* quantitative analysis of the results.

Whether the binding time is clearly known, it is possible to make recommendations about which variability mechanisms to use. [15]. Since the dynamic variability is achieved at

[1]The SHR website is available at: https://sites.google.com/site/smarthomerise/

runtime, we decided to initially use the binding time as a criterion to select the candidate mechanisms to implement the DSPL application. In this way, the SHR was implemented by using the following variability mechanisms: parameterization, aggregation, reflection, and polymorphism [15, 18]. However, the complete description and a more detailed discussion of the data of this implementation are beyond the scope of this paper.

The SHR DSPL project was implemented in Java and comprises 11 concrete features, including 9 packages. The OO version of the project was implemented with 52 classes, and 4059 lines of code. The AO version was implemented with 52 classes, 3 aspects, and 4087 lines of code. The SHR design aimed at modularizing user interface, core, and features. These are layers in both OO and AO architectures. However, the OO version fails to completely prevent code tangling and scattering, since adaptation to the current context is often an element that crosscuts the application logic.

Although in the AO design it is difficult to organize the codebase so that it does not compromise the separation of concerns, the AO implementation modularize some concerns that were tangled and scattered in the OO decomposition counterpart. In this design, the crosscutting elements concerning to the business rules, features, and exception handling were modularized as aspects, namely *HomeAspect*, *FeatureUIAspect*, and *FeatureAspect*. For instance, the hardware management concern was removed from the core layer and encapsulated in the *HomeAspect*. In addition, *FeatureUIAspect* encapsulates some behavioral variations, which were scattered in the OO design.

The complete releases correspond to respective scenario changes and simulations of maintenance tasks and reuse. Seven scenarios were considered in SHR, resulting in eight releases. The scenarios encompass different types of changes involving **illumination, security,** and **temperature** control systems and `mandatory`, `or`, `optional`, and `alternative` features.

Table I shows the scenario changes of the SHR according to the feature types. The filled circles represent an included feature, and the blank ones represent features that are not included. In both designs, release **R01** contains the DSPL core. All subsequent releases were designed to incorporate the required changes in order to include the corresponding `optional`, `or`, and `alternative` features. For instance, in the release **R02**, two `optional` features were added. In the release **R07**, the illumination feature was added, and in the release **R08**, the temperature feature was added.

Conversely, from Table I it is possible to see that some features were removed, from release to release. It was necessary to meet the requirements concerning to the particular kinds of features. For instance, in the releases **R07**, the security and temperature features were removed. In the release **R08**, the temperature feature was added and the illumination feature was removed.

The quantitative assessment considered the measure: *complexity, cohesion, coupling*, and *change propagation impact*. These measures are predictor factors for external attributes as *reusability* and *maintenance* [28]. In addition, it is possible to evaluate the design stability of a software system by comparing its versions.

TABLE I: Reuse and Maintenance Scenarios in `SHR`

	Evolution							
Feature	**Reuse**							
	R01	**R02**	**R03**	**R04**	**R05**	**R06**	**R07**	**R08**
01	●	●	●	●	●	●	○	○
02	●	●	●	●	●	●	●	○
03	○	●	●	●	●	●	○	○
04	○	●	●	●	●	●	○	○
05	○	○	●	●	●	●	○	○
06	○	○	○	●	●	●	●	○
07	○	○	○	●	●	●	●	○
08	○	○	○	○	●	●	○	●
09	○	○	○	○	●	●	○	●
10	○	○	○	○	○	●	○	●
11	○	○	○	○	○	●	○	●

(01): PresenceIllusion; (02): UserIllumination; (03): Alarm; (04): PanicMode; (05): LockDoors; (06): ByPresence; (07): ByLuminosity; (08): FromWindow(User); (09): FromAirConditioning(User);(10): FromWindow(Automated); (11): FromAirConditioning(Automated); ●: Included; ○: Not included

The metrics were collected using the following tool: **AOPMetrics**[2]. For quantitative data, the analysis included descriptive statistics, such as *mean* values, standard derivation (*StDev*), *variance*, and *boxplots* aiming to explore gathered data.

V. Analysis and Interpretation

This subsection provides an in-depth analysis of the gathered data. The evaluation was conducted through descriptive statistics in order to get a general view of a data set and its distribution.

A. Complexity analysis

This subsection presents a quantitative analysis to answer **RQ1** and **RQ4**. Figure 2 and Figure 3 show the complexity criterion in each solution concerning two measures: Weighted Operations per Component (WOC) and Lines Of Code (LOC), respectively. A general interpretation of these measures is that lower values indicate less complex solutions. We next provide an in-depth descriptive analysis of each measure.

Weighted Operations per Component (WOC). For this measure in AO, we counted advice blocks as operations belonging to *aspects*. Table II shows the descriptive statistics to WOC measure for every release. In general, concerning the number of operations, the variation among the releases was almost similar. Nonetheless, in releases **R02**, **R03**, **R04**, **R05**, **R06**, and **R07**, the WOC value is increasing. The plot shows that these releases have the upper whisker greater in the OO scenario.

Additionally, we can see an outlier in all releases of both OO and AO. This observation came from a fundamental class, called *HouseFacade*, which is responsible for implementing several crosscutting concerns. This class allows to access the business collections, implemented in other core classes.

[2]AOPMetrics http://aopmetrics.tigris.org/

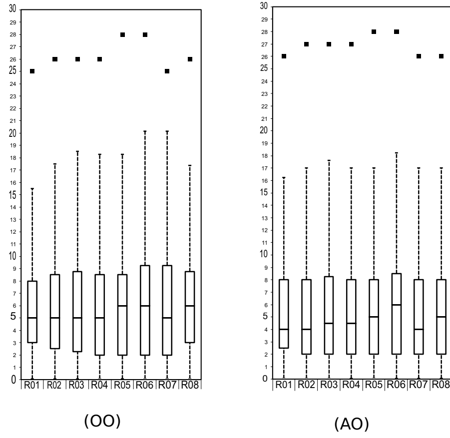

Fig. 2: Weighted Operations per Component (WOC)

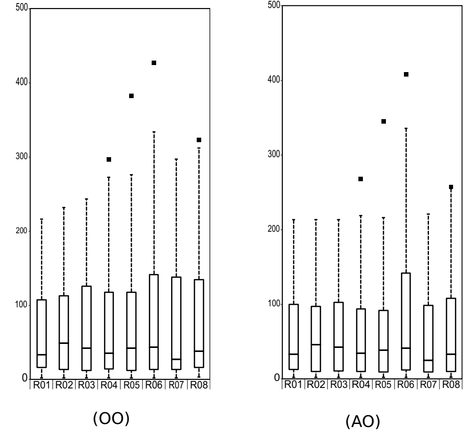

Fig. 3: Lines Of Code (LOC)

TABLE II: WOC Descriptive Statistics

Release	Variable	Components	Mean	StDev	Variance
R01	**OO**	21	6,140	5,680	32,230
	AO	23	5,960	5,760	33,130
R02	**OO**	27	6,070	5,530	30,610
	AO	29	5,860	5,560	30,910
R03	**OO**	30	6,167	5,446	29,661
	AO	32	5,969	5,480	30,031
R04	**OO**	39	6,282	5,301	28,103
	AO	42	6,048	5,437	29,559
R05	**OO**	47	6,277	5,352	28,639
	AO	50	5,960	5,580	31,141
R06	**OO**	52	6,558	5,345	28,565
	AO	55	6,364	5,338	28,495
R07	**OO**	28	6,500	5,640	31,810
	AO	31	6,030	5,690	32,370
R08	**OO**	30	6,700	5,440	29,597
	AO	33	6,273	5,375	28,892

TABLE III: LOC Descriptive Statistics

Release	Variable	Components	Mean	StDev	Variance
R01	**OO**	21	64,8	66,8	4461,4
	AO	23	60,4	63,9	4085,5
R02	**OO**	27	67,9	69,3	4806,5
	AO	29	63,7	65,4	4274,4
R03	**OO**	30	70,0	72,4	5247,9
	AO	32	66,0	68,1	4634,8
R04	**OO**	39	71,5	80,0	6403,7
	AO	42	67,1	74,2	5507,9
R05	**OO**	47	74,3	89,0	7915,2
	AO	50	70,4	86,5	7483,6
R06	**OO**	52	78,1	95,2	9066,1
	AO	55	74,3	86,0	7390,3
R07	**OO**	28	70,7	83,9	7044,7
	AO	31	61,4	69,9	4892,2
R08	**OO**	30	73,3	81,9	6711,6
	AO	33	66,0	74,2	5501,2

Although the AO implementation adds more pointcuts and advices needed to modularize *concerns*, observing the *mean* value for all releases we have considered that the OO implementation is more complex regarding the WOC measure.

Lines Of Code (LOC). In general, the variation of the data set among both solutions was similar. In OO, the *median* value among all releases was close and irregular. The same applies with the releases in AO. However, the LOC *mean* value in OO design is higher than in AO for all releases. Table III shows the descriptive statistics to LOC measure for all releases.

In both solutions, the increase in maximum values until release **R06** resulted in the appearance of outliers, since new features were included during the evolution task. The outliers have occurred in the *Main* and *HouseFacade* classes due to the insertion of `or` features *ByPresence*, *ByLuminosity*, *FromWindow*, and *FromAirConditioning* (automated control), besides the `alternative` features *FromWindow* and *FromAirConditioning* (user control). In this case, several concerns and joint points to the user interface classes were implemented.

The upper whisker of the releases **R06** and **R08** was greater in AO than OO. We have associated it with the occurrence of `or` and `alternative` features in the same release. In this case, new pointcuts and advices were included resulting in more lines of code and increased complexity. On the other hand, the *median* value to LOC measure showed that OO is more complex than in AO, since the use of *aspects* reduce the number of lines of code for all components.

As a general indicator for the size attribute, both OO and AO presented similar variation, resulting in very similar stability for evolution task and reuse.

B. Cohesion analysis

This subsection presents a quantitative analysis to answer **RQ2** and **RQ4**. Figure 4 shows the cohesion criterion in each

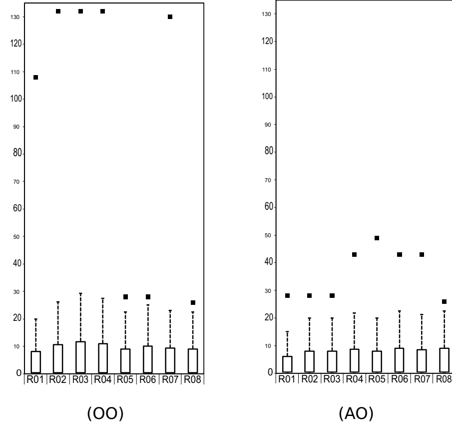

Fig. 4: Lack of Cohesion Over Operations (LCOO)

TABLE IV: LCOO Descriptive Statistics

Release	Variable	Components	Mean	StDev	Variance
R01	**OO**	21	9,430	23,82	567,26
	AO	23	4,130	7,630	58,210
R02	**OO**	27	9,740	25,57	653,58
	AO	29	4,520	7,410	54,970
R03	**OO**	30	9,600	24,39	594,73
	AO	32	4,880	7,680	58,950
R04	**OO**	39	9,380	21,99	483,66
	AO	42	6,020	10,06	101,15
R05	**OO**	47	6,110	8,800	77,40
	AO	50	6,120	10,97	120,35
R06	**OO**	52	6,870	9,260	85,810
	AO	55	6,840	10,13	102,55
R07	**OO**	28	10,36	25,23	636,39
	AO	31	5,770	10,85	117,78
R08	**OO**	30	6,030	8,940	79,900
	AO	33	6,450	9,090	82,630

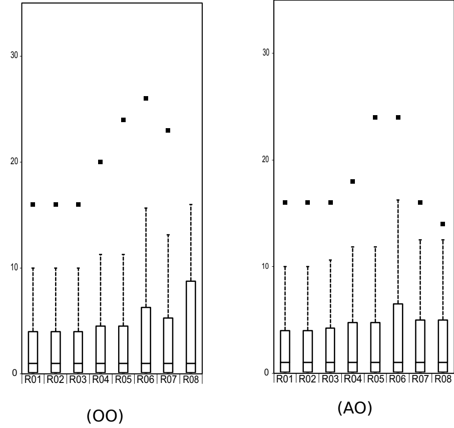

Fig. 5: Coupling Between components (CBC)

TABLE V: CBC Descriptive Statistics

Release	Variable	Components	Mean	StDev	Variance
01	**OO**	21	3,100	4,610	21,290
	AO	23	2,957	4,517	20,407
02	**OO**	27	3,593	5,139	26,405
	AO	29	3,448	4,968	24,685
03	**OO**	30	3,870	5,490	30,120
	AO	32	3,750	5,328	28,387
04	**OO**	39	4,026	5,728	32,815
	AO	42	3,810	5,452	29,719
05	**OO**	47	4,255	6,212	38,586
	AO	50	4,180	6,190	38,314
06	**OO**	52	4,404	6,350	40,324
	AO	55	4,236	6,077	36,925
07	**OO**	28	3,710	5,860	34,290
	AO	31	3,161	4,734	22,406
08	**OO**	30	3,833	5,325	28,351
	AO	33	3,182	4,647	21,591

solution concerning to Lack of Cohesion Over Operations (LCOO). A general interpretation of this measure is that a lower value indicates a less cohesive solution. We next provide a descriptive analysis of the measure.

Lack of Cohesion Over Operations (LCOO). The LCOO variation presented similar behavior comparing both OO and AO during evolution. Table IV shows the descriptive statistics of the LCOO measure for every release. In general, AO yielded better results, since the *aspects* filter out crosscutting behavior. It means that AO is more cohesive than OO.

In OO, the releases **R01**, **R02**, **R03**, **R04**, and **R07**, the LCOO measure to the *HouseFacade* component had a value greater than one hundred resulting in an outlier. In contrast, in the releases **R05**, **R06**, and **R08**, this same component presented a value equal to zero. We have associated this sharp variance with the inclusion of `optional` and `or` features.

The high value as well as the value equal zero indicates that the component has low cohesion. Whereas a LCOO measure of zero cannot support evidence that a component is cohesive nor it holds a stable interface, the high LCOO indicates a component that shall be considered for splitting it into several components.

In summary, it indicates that AO is more stable than OO for cohesion measurement. It can be seen that more scenarios in OO were affected regarding cohesion. The only major change for AO occurred in only one release, while in OO the major change occurred in five of them.

C. Coupling analysis

This subsection presents a quantitative analysis to answer **RQ3** and **RQ4**. Figure 5 and Figure 6 show the coupling criterion in each solution concerning two measures Coupling Between components (CBC) and Response For a Class (RFC), respectively. A general interpretation of these measures is that

lower values indicate a less coupled solutions. We next provide a descriptive analysis of each measure.

Coupling Between Components (CBC). Both solutions presented a constant *median* among every release. The data set had a similar behavior comparing both OO and AO from releases **R01** to **R05**. However, outliers are showed in these releases. We consider the outliers in the components that implement concerns related to features and hardware management. In addition, the third quartile presented greater value in OO than AO in the releases **R01**, **R03**, **R06**, **R07**, and **R08**. It indicates that OO is more coupled than in AO. Table V shows the descriptive statistics to CBC measure for all releases.

In general, the presence of *aspects* is likely to decrease the coupling between core classes and increase the coupling between core class and *aspects*. This is because *aspects* are new entities on which core classes depend upon. Decreasing the coupling between core classes is a beneficial issue, and increasing coupling between aspects and core classes in return can be seen as a good trade-off. Given that a design might involve coupling between classes, it would be better to have this coupling occurring from classes-to-aspect, rather than having it happening only from classes-to-classes.

Response For a Class (RFC). AO presented a growing *median* value in the releases **R04**, **R05**, **R06**, **R07**, and **R08**. It might have occurred due to the inclusion of `or` and `alternative` features, since the components targeted at implementing of these features had a high RFC value.

Table VI shows the descriptive statistics to RFC measure for all releases. The variance of the data set was similar. Nevertheless, the *mean* value is higher in OO than in AO for all releases. It indicates that OO implementation is more coupled regarding the RFC measure.

Additionally, the core-to-aspect invocations are counted when calculating RFC. Thus, the RFC value increased in the presence of *aspects*. It means that the number of entities that a class communicates with has been increased, and thus, the classes have to communicate with the *aspects*. However, we have noticed that, by using *aspects* it is possible to encapsulate the logic and the objects with which a class communicates in a modular way.

In summary, for the coupling measures, the variation between the solutions was very similar. The CBC and RFC graphs show that in both solutions these measures presented more irregular values for releases, in which the `or` and `alternative` features were included. However, the OO solution presented more stable values for all releases.

D. Change Propagation Impact

This subsection presents a quantitative analysis to answer **RQ4** and **RQ5**. In particular, we aimed at knowing how the different paradigms affect the evolution task and reuse in the context of DSPL.

Change is a key element in software development, embracing evolving requirements and improvements in existing software artifacts. Change effects are directly related to design

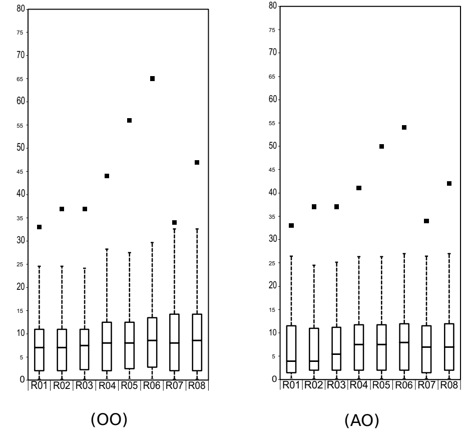

Fig. 6: Response For a Class (RFC)

TABLE VI: RFC Descriptive Statistics

Release	Variable	Components	Mean	StDev	Variance
01	**OO**	21	9,480	10,08	101,66
	AO	23	8,870	10,37	107,48
02	**OO**	27	9,780	10,85	117,64
	AO	29	9,140	10,76	115,77
03	**OO**	30	10,20	11,11	123,34
	AO	32	9,560	10,96	120,06
04	**OO**	39	10,74	11,27	127,09
	AO	42	10,10	11,49	131,94
05	**OO**	47	11,02	12,08	145,85
	AO	50	10,42	12,20	148,78
06	**OO**	52	11,67	12,71	161,44
	AO	55	11,33	12,73	162,00
07	**OO**	28	10,71	10,83	117,25
	AO	31	9,710	11,00	120,95
08	**OO**	30	11,23	11,66	136,05
	AO	33	10,39	11,49	132,00

stability. The same still holds true for DSPL engineering. In DSPL engineering, evolution should be conducted through non-intrusive and self-contained changes that favor insertions and do not require deep modifications into existing components.

In this current investigation, the considered changes mainly comprised either the addition or removal of features. As a result, it was possible to provide an additional indication whether a specific paradigm could provide a better design stability support for some types of change.

Variability management plays an important role in DSPL evolution, as it is necessary to avoid inconsistent adaptations at runtime. In this sense, the variability mechanism should provide support to extend the system functionality, ensuring the DSPL architecture stability, without decreasing modularity.

In this way, traditional change impact metrics were used considering different levels of granularity: components, oper-

TABLE VII: Summary of the SHR implementation

			R01	R02	R03	R04	R05	R06	R07	R08
Absolute value	**Lines Of Code**	**OO**	1361	1834	2100	2787	3490	4059	1980	2198
		AO	1390	1846	2112	2819	3521	4087	1904	2177
	Operations	**OO**	129	164	185	245	295	341	182	201
		AO	137	170	191	254	298	350	187	202
	Components	**OO**	19	24	27	36	43	48	26	28
		AO	20	26	29	39	47	51	29	31
Lines Of Code	**Added**	**OO**	-	473	266	687	703	569	0	1042
		AO	-	456	266	707	702	566	0	1011
	Removed	**OO**	-	0	0	0	0	0	2206	454
		AO	-	0	0	0	0	0	2183	463
Operations	**Added**	**OO**	-	35	21	60	50	46	0	45
		AO	-	33	21	63	44	52	0	42
	Removed	**OO**	-	0	0	0	0	0	90	21
		AO	-	0	0	0	0	0	93	21
Components	**Added**	**OO**	-	6	2	2	7	5	0	13
		AO	-	6	2	2	9	5	0	13
	Removed	**OO**	-	0	0	0	0	0	24	7
		AO	-	0	0	0	0	0	24	7

ations, and lines of code. A general interpretation of these measures is that the lower the change impact measures the more stable and resilient the design is to a specific change. Table VII shows the summary of the SHR implementation and change propagation metrics.

In general, removing the number of lines of code, operations, and components did not bring significant difference and variation between the measures of OO and AO. In addition, both solutions had similar change propagation values. However, AO generally require additional components, operations, and lines of code to implement the changes during evolution task and reuse.

By observing the absolute values in releases **R02**, **R05**, **R07**, and **R08**, they indicate that OO solution required existing components to be modified more extensively in order to implement the changes. This behavior was confirmed in the change propagation metrics, since its values showed more extensive changes in terms of added lines of code and operations. Conversely, in **R04** and **R06**, the same observation did not show up, as the number of added lines of code and operations were exceeded in AO. In addition, release **R03** presented similar values in both solutions.

The analysis of these scenarios through change propagation metrics confirmed that the inclusion of the `optional` features *Alarm* and *PanicMode* and the `alternative` features *FromWindow* and *FromAirConditioning* (user control) have a greater impact on the design stability in OO than in AO. Thus, it is more effective the use of *aspects* to manage these type of features in DSPL evolution. When considering the inclusion of `or`-features, it indicates that the AO solution has greater design stability impact. It means that *aspects* reduce the absorbing of changes.

VI. Lessons Learned

After concluding the exploratory study, there are some aspects to consider. Such aspects may aid further replications of the study, as they could be understood as limitations faced in the present execution.

Paradigm. The transfer from AO solution to DSPL development is largely dependent on the ability to understand the paradigm specification and implementation. In this sense, a complete and detailed planning should be defined by software designers in order to cover the *concerns* to be modularized into an *aspect* in the DSPL evolution.

Evolution. In the exploratory study, the evolution of DSPL project was carried out manually. However, as the number of new requirements increases, the inclusion of features may become a complex task. Thus, an automatic inclusion of features could facilitate the engineering work. In this sense, the evolution at runtime should be investigated in order to identify more flexible solutions in offering automatic configuration and reconfiguration.

VII. Threats to Validity

Construction validity. Complexity, cohesion, coupling, and instability are difficult concepts to measure. For this study, the metrics were selected based on the previously performed studies [13, 14, 16, 17] which obtained relevant results using them. In order to increase the reliability of measures, the **AOPMetrics** tool was used to collect the data set that was evaluated in this exploratory study.

Internal validity. There are some threats that were considered in this exploratory study, such as the assessment of a single domain, the variability mechanisms, and the number of releases. In order to mitigate these threats, we aim to carry out further empirical studies considering other domains, different variability mechanisms, and new releases.

External validity. The project was developed from scratch, since we did not find a DSPL application available for empirical evaluation purposes. Nonetheless, we understand that other applications in a different domains could be used in order to assess the solutions. It means that another exploratory study following the same research issues can be more conclusive stating which solution is more suitable to implement DSPL in evolutionary scenarios.

The findings of the analysis can be used as baseline for comparing other studies in the context of DSPL, since the exploratory study protocol has been developed in detail and reviewed by researchers. Thus, it reduces the threat to the external validity of the exploratory study.

Conclusion validity. Since 3664 data points were collected, the reliability of the measurement process might be an issue. For this reason, an independent author who did not collect the respective data applied statistical analysis.

Finally, the results of the study were described using *Descriptive statistics*, which deal with numerical processing and presentation of a data set. It is the most suitable method to describe the analysis and interpretation of this data type collected.

VIII. Concluding Remarks

The central purpose of DSPL engineering is to deal with adaptability at runtime, as well as to maximize the reuse of components. For this reason, it is essential to identify a suitable solution to implement DSPL applications aiming to increase software quality by affecting internal attributes such as *size, cohesion, coupling*, and *instability*. Identifying the most suitable solution promotes benefits to the maintenance activities and reuse of components of a DSPL application. As a result, it can be possible to ensure continuous improvement for implementing these systems.

An exploratory study was conducted with two subjects, and the OO and AO solutions were compared among each other. A set of metrics was used for both paradigms aiming to meet the following requirements: *(i)* to measure quality factors and *(ii)* to support the identification of advantages and drawbacks from the use of *aspects* in the DSPL evolution in comparison to OO design. In this exploratory study was analyzed the possibility to implement components with lower complexity, lower coupling, higher cohesion, and a more stable design. Moreover, the assessment encompassed the change propagation impact.

The main findings of the study showed that a solution fits best in particular cases. In this sense, AO was pointed out as a suitable solution for dealing with specifics situations. For example, the use of *aspects* indicates that it provides assets with lower complexity, lower coupling, higher cohesion, and support lower change propagation impact.

Nonetheless, we believe that other domains should be used in order to assess the application of the solutions to implement DSPL evolution and cope with dynamic adaptation. Further research actions should be taken in order to gather more solid evidence about the findings. As future work, we plan to conduct new studies in other domains, considering also other paradigms and languages, such as Delta-oriented Programming and Context-oriented Programming.

Acknowledgments

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[1]), funded by CNPq and FAPESB.

References

[1] N. Bencomo, S. O. Hallsteinsen, and E. S. de Almeida. A view of the dynamic software product line landscape. *IEEE Computer*, pages 36–41, 2012.

[2] K. Berg, J. Bishop, and D. Muthig. Tracing software product line variability: From problem to solution space. In *Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, pages 182–191. South African Institute for Computer Scientists and Information Technologists, 2005.

[3] J. Bosch and R. Capilla. Dynamic variability in software-intensive embedded system families. *IEEE Computer*, pages 28–35, 2012.

[4] V. A. Burégio, S. R. de Lemos Meira, and E. S. de Almeida. Characterizing dynamic software product lines - A preliminary mapping study. In *Proceedings of the 14th International Conference in Software Product Lines (SPLC)*, pages 53–60, 2010.

[5] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, pages 3–23, 2014.

[6] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Using feature models for developing self-configuring smart homes. In *Fifth International Conference on Autonomic and Autonomous Systems (ICAS)*, pages 179–188. IEEE Computer Society, 2009.

[7] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Designing and prototyping dynamic software product lines: Techniques and guidelines. In *Proceedings of the 14th International Conference on Software Product Lines (SPLC)*, pages 331–345. Springer-Verlag, 2010.

[8] C. Cetina, P. Trinidad, V. Pelechano, and A. Ruiz-Cortés. An architectural discussion on dspl. In *2nd SPLC Workshop on Dynamic Software Product Line (DSPL)*,

[1] http://www.ines.org.br

pages 59–68. Irish Software Engineering Research Centre (Lero), September 2008.

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions Software Engineering*, pages 476–493, 1994.

[10] D. Cook, M. Youngblood, I. Heierman, E.O., K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja. MavHome: an agent-based smart home. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 521–524, 2003.

[11] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A dynamic software product line approach using aspect models at runtime. In *Proceedings of the 1st Workshop on Composition and Variability*. ACM, 2010.

[12] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 2nd edition, 1998.

[13] G. C. S. Ferreira, F. N. Gaia, E. Figueiredo, and M. de Almeida Maia. On the use of feature-oriented programming for evolving software product lines - A comparative study. *Sci. Comput. Program.*, 93:65–85, 2014.

[14] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 261–270. ACM, 2008.

[15] C. Gacek and M. Anastasopoules. Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context (SSR)*, pages 109–117. ACM, 2001.

[16] F. N. Gaia, G. C. S. Ferreira, E. Figueiredo, and M. de Almeida Maia. A quantitative and qualitative assessment of aspectual feature modules for evolving software product lines. *Science of Computer Programming*, pages 230–253, 2014.

[17] P. Greenwood, T. T. Bartolomei, E. Figueiredo, M. Dsea, A. F. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *ECOOP–Object-Oriented Programming*, pages 176–200. Springer, 2007.

[18] J. V. Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 45–54. IEEE Computer Society, 2001.

[19] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *IEEE Computer*, pages 93–95, 2008.

[20] A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout, and W. Van Betsbrugge. Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines. In *Proc. of the 3rd International Workshop on Dynamic Software Product Lines (DSPL)*, pages 18–27, 2009.

[21] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, pages 125–151, 2008.

[22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis FODA feasibility study. 1990.

[23] C. Larman. *Applying UML and Patterns: An Introduction to Object -Oriented Analysis and Design and the Unified Process*. Pretice Hall PTR, 2002.

[24] F. J. v. d. Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[25] P. S. M. Hinchey and K. Schmid. Building dynamic software product lines. *IEEE Computer*, pages 22–26, 2012.

[26] B. Morin, O. Barais, and J.-M. Jézéquel. K@RT: An aspect-oriented and model-oriented framework for dynamic software product lines. In *Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS*, 2008.

[27] C. Quinton, R. Rabiser, M. Vierhauser, P. Grünbacher, and L. Baresi. Evolution in dynamic software product lines: Challenges and perspectives. In *Proceedings 19th International Software Product Line Conference (SPL)*, pages 126–130. ACM, 2015.

[28] C. SantAnna, A. Garcia, C. Chavez, C. Lucena, and A. Von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings of Brazilian symposium on software engineering*, pages 19–34, 2003.

[29] L. Shen, X. Peng, J. Liu, and W. Zhao. Towards feature-oriented variability reconfiguration in dynamic software product lines. In *Top Productivity through Software Reuse*, pages 52–68. Springer, 2011.

[30] M. L. G. Silva, M. L. L. Carvalho, A. R. Santos, and E. S. Almeida. SmartHomeRiSE: An DSPL to home automation. *Congresso Brasileiro de Software (CBSoft) - Ferramentas*, pages 32–39, 2015.

[31] M. Svahnberg and J. Bosch. Evolution in software product lines. pages 391–422. Citeseer, 1999.

[32] M. A. Talib, T. Nguyen, A. W. Colman, and J. Han. Requirements for evolvable dynamic software product lines. In *14th International Conference on Software Product Lines (SPLC)*, pages 43–46, 2010.

[33] K. F. Tom Dinkelaker, Ralf Mitschke and M. Mezini. A dynamic software product line approach using aspect models at runtime. In *Proceedings of the 1st Workshop on Composition and Variability*. CEUR Workshop, 2010.

[34] C. Yang, B. Yuan, Y. Tian, Z. Feng, and W. Mao. A smart home architecture based on resource name service. In *Proceedings of the 17th IEEE International Conference on Computational Science and Engineering (CSE)*, pages 1915–1920. IEEE Computer Society, 2015.