Assessing the Responsibility of Software Product Line Platform Framework for Business Applications

Zdravko Roško
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
zrosko@gmail.hr

Abstract. *Within the context of software product lines for business applications, early indicators of the software product line architecture quality attributes can be used in order to avoid low-quality products during the later stages of product development. Today's application engineering is not mainly concerned with business user requirements, as it should be, but in practice we find it concerned with the technical complexity and also, not enough decoupled from directly using of too many external third party components. In this paper we propose a „Platform Framework Responsibility“ metrics which can be used as an early indicators of the future product's quality. The domain and application engineering processes that use and apply the early indicators of the platform framework quality attributes will help ensure that final products are stable, maintainable and better decoupled from external third party component's dependencies.*

Keywords. Software Product Lines, Metrics, Platform Framework Responsibility, External dependency, Business Applications.

1 Introduction

Business applications are a kind of software that is used by business users to perform various business functions. Most of the business applications are interactive, they interact with a user through a user interface in order to read, process or change some of the persistent business data. The software product line (SPL) for interactive business applications defines product line requirements, software architecture and a set of reusable components. One of the most important parts of a SPL is its architecture (PLA). The PLA plays a central role at the development of products from a SPL as it is the abstraction of the products that can be generated, and it represents similarities and variabilities of a product line [1]. The PLA must consider the needs of the complete set of products in order to provide a framework for the development and reuse of new assets. These new assets have to be conceived with the required flexibility in order to satisfy the needs of the different products in the SPL [2]. PLA consist of *frameworks* (Szyperski., 2002) as core assets, whose design captures recurring structures, connectors, and control flow in an application domain, along with the points of variation explicitly allowed among these entities [1]. In this paper we use the term „SPL platform framework“ to represent the implementation of the generic architecture and components which are not business-specific but rather generic in the sense that they can be used by more than one business domain such as: banking, insurance, manufacturing, and etc. The platform framework implements considerable part of the product's functionalities and is shared by all or most of the products within the product line.
The philosophy of component frameworks is to develop reusable components that are well-defined and have specific use contexts and variability points, which helps reduce the effort associated with using external components, low-level middleware interfaces or OS APIs [3].

Prod 1	Prod 2	Prod 3	Prod 4
Business-specific components			
SPL Platform Framework			
External Components			
OS/Language Environment			

Table 1. Proposed PLA structure

A product spawned from a SPL may depend on architectural aspects at different levels of abstraction and generality, from OS/Language at the low level through external components and product line platform framework, to business-specific components, shown in Table 1 [4].
Platform framework is being developed through the life time of a SPL, but most of the core features are

developed by the end of the development of the third product in a SPL. We may call this time as an early stage of the SPL development (Figure 1) partially taken from [5]. It is commonly believed that the early software process phases are the most important ones, since the rest of the development depends on the artifacts they produce [6].

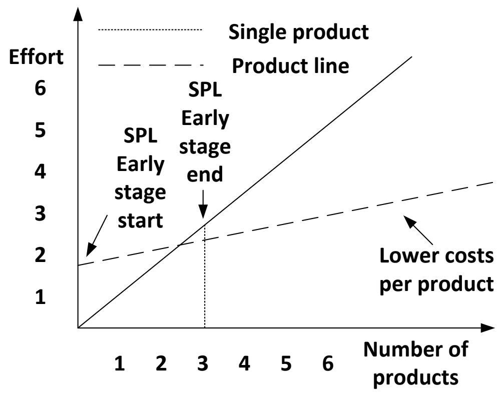

Figure 1. Cost of a SPL development

Since SPL platform framework is used by many products, the emphasis should be on maintaining its quality attributes at the early stage, rather than trying to enforce quality at the later stages, as it will directly impact the quality of the final SPL products.
At this time no metrics for measuring the “responsibility” of the SPL platform framework exists. One of the reasons for this is the lack of appropriate mechanisms for measuring the properties of software product lines [7]. Many software engineering researchers have used measurement as means of improving software quality [8]. The objective of this paper is to define a metric for software product line platform framework for measuring “responsibility” of the platform framework (PR) in the context of already developed product from the SPL. The metric may be used as an early indicator of product *stability* which is an important external quality attribute of a product. The metric would also serve to improve the quality of the resulting software products by helping to predict the possible quality of the final system and improve the resource allocation process based on these predictions [9]. The rest of this paper is structured as follows. In section 2, we introduce the context of proposed metrics. Section 3 explains quality characteristics of platform frameworks. Section 4 provides set of “responsibility” metrics and its details.

2 The context of proposed metrics

Software product line engineering is concerned with capturing the commonalities, universal and shared attributes of a set of software-intensive applications for a specific problem domain [5]. In terms of costs, as stated by [5] SPL offer benefits when producing at least a certain number of products.
To set the context of this paper, we begin with an overview of software product line for business applications. Suppose that we want to develop business applications for a specific problem domain such as banking. The process of the development of business applications from a SPL is divided in two main tasks: *Domain Engineering* and *Application Engineering* as illustrated in Figure 2 [2]. Domain engineering refers to the creation of shared assets from scratch or from existing products, whereas application engineering refers to the process of developing individual products from those assets. The development of shared assets is continuous and lasts through the life time of the SPL but the core of the shared assets are developed at the early stage of a SPL as shown in Figure 1.

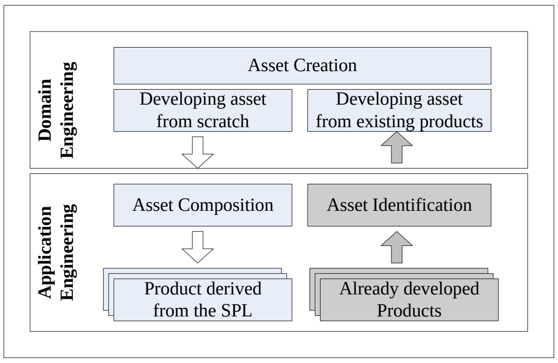
Domain Engineering
Asset Creation
Developing asset from scratch
Developing asset from existing products
Application Engineering
Asset Composition
Asset Identification
Product derived from the SPL
Already developed Products

Figure 2. Domain and application engineering processes

A platform framework is a group of components and services that provide a coherent set of functionalities through inheritance, interfaces and specific design patterns. A product application derived from a given platform framework may use these services without worrying about how those services are implemented. The application development process should be concerned with the business requirements rather than with the low level APIs or external component's interaction rules. Platform framework needs to ensure the application development process independence by taking the „responsibility“ to interact with external third-party components. By external components we refer to a non-development components developed by a third party organizations and used by the SPL platform framework or by a products spawned from it, illustrated in Figure 3. Referencing an external component directly from a business application product makes the product less stable and harder to develop or change. The product line platform framework should take as much as possible of the „responsibility“ to interact with external components. Among other mechanisms, the design patterns such as Strategy [4] can be used as a variability mechanism to enable use and replacement of different external components or its versions. Thus, the external components can be changed or evolved without affecting the core business logic while multiple external component types can be supported with the same business application product. As technology changes over time, the technical and implementation architecture can be evolved to take advantages of the new technologies, while still protecting the existing products from those changes.

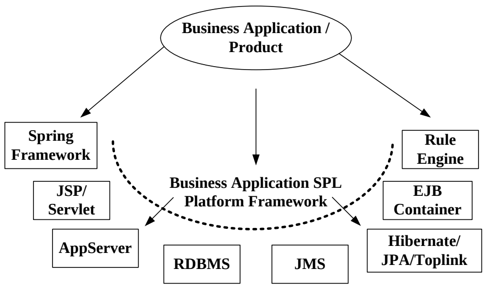

Figure 3. External components context

Software metrics to measure quality attributes of architecture such as "Design Quality" metrics [10], metrics to measure structural soundness of product line architecture [11], PLA metrics [12], PLA architecture metrics [13] and complexity metrics for software product line architectures [1] do not address the quality of platform framework "responsibility". The instability metric [10] measures the stability of a category by calculating the ratio between Afferent Coupling (number of classes outside the category that depend upon classes within the category) and Efferent Coupling (number of classes inside the category that depend upon classes outside the category).
This metrics cannot be used in the context where we measure coupling between SPL and external components, since external components do not depend on any of internally developed components.
The evaluation of a SPL platform framework may be measured by a set of metrics we propose.

3 Platform framework quality

SPL platform framework for business applications provides a set of core components to be used by business applications. Business applications are sharing a set of domain-independent generic components such as transaction, session, logging (Figure 4), and a set of domain specific components that can be used in applications of a particular domain such as banking, manufacturing, and etc.
The domain-independent components packaged in the form of platform framework should be responsible for handling low level interface interactions to external components, however its "responsibility" level depends on the quality of its architectural design.

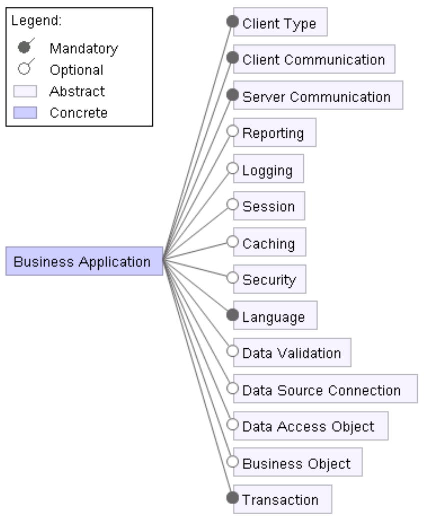

Figure 4. SPL Platform Framework Feature model

As Figure 5 shows, the "responsibility" (dotted line) of a platform framework is higher and provides a *Better Quality Direction* if a number of references from SPL Product to External Components and Environment is lower.

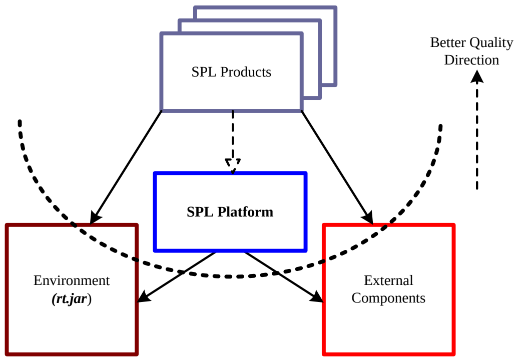

Figure 5. SPL Platform "responsibility"

The final goal for a sound platform framework is to take full "responsibility" for interaction to external components and leave the products free from low-level interactions to the third-party external components (dotted arrow line shows a wanted direction). SPL platform framework properties have an important impact on spawned products stability.
As illustrated in Figure 6 there are 5 distinct high level dependency metrics of a SPL for business applications. SPL platform framework depends on its environment such as Java or .NET and on a number of external third-party components while SPL products depend on its platform framework, its environment and on third-party external components as well. A sound architecture assumes a minimal dependency from SPL products on external components and even on environment or operating system APIs.

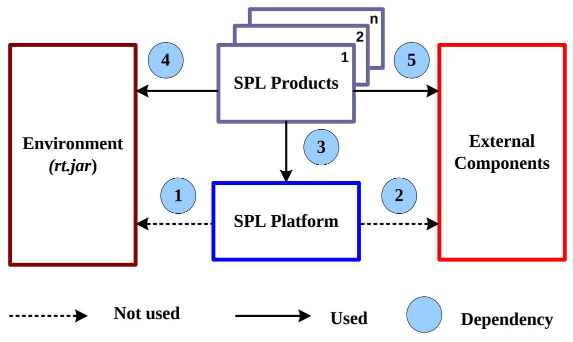

Figure 6. SPL Platform Framework Metrics

4 Responsibility metrics

A few studies for defining suitable product line architecture metrics have been conducted [11] [12] [1] [13]. Given a fact that single metric cannot capture all of the various aspects of complexity [8], in our study we propose five simple and intuitive architectural metrics as a measurement for software product line platform framework quality based on the architectural elements dependency.
Software dependence is a relationship between the two pieces of code, such as a data dependency, call dependency, etc. [14]. Here we use the reference dependence where each distinct reference to an element is counted. For example, a relation (X, Y) between elements X and Y signifies that X references Y. Given these two elements X and Y, Y may be referenced by X more than once. The reference count is 0 or more, depending on total number of references from X to Y.
A dependency analysis for a product which is spawned from a SPL platform can be performed to ensure the stability of spawned products. The dependencies that are computed can potentially be viewed from more different angels: dependencies between product and SPL platform, product and external components, product and environment (e.g. Java RTE), SPL platform and external components, SPL platform and environment, projects, packages, or types (classes and interfaces). The responsibility, interdependence and stability of a category can be measured by counting the dependencies that interact with that category [10]. We use SPL product and SPL platform as a dependency category. The proposed SPL platform „responsibility“ metric uses the three dependencies metrics:

D3: Platform Afferent Coupling: The number of distinct references outside the Platform that depend upon classes within the Platform.
D4: Product Efferent Coupling: the number of distinct references inside the product that depend upon classes within environment components (e.g. Java RTE).

D5: Product Efferent Coupling: The number of distinct references inside the product that depend upon classes within external components.

Here we use the data from the dependency analysis to calculate five metrics:

Measure type	Measure name
Size	Number of Platform/environment (language) class dependencies (D1)
	Number of Platform/external components class dependencies (D2)
	Number of Product/Platform class dependencies (D3)
	Number of Product/environment (language) class dependencies (D4)
	Number of Product/external components class dependencies (D5)
Complexity	Platform framework responsibility (PR)

Table 2. Dependencies metrics

PR: Platform Responsibility: (D4+D5 / (D3+D4+D5): The range for this metric is from 0 to 1, where PR=0 indicates that SPL platform used by product makes the products more stable and protected from frequent changes to the external third party components, while the SPL platform serves the products by taking the responsibility to interact with external components. PR=1 indicates a completely irresponsible SPL platform where products are directly referencing external components while SPL platform does not help to improve their stability. Figure 7 shows a dependency counts from Eclipse Java sample project.
Dependencies between the elements are displayed as directed lines (lines with arrows at either one end or both in case of mutually dependent elements). The elements are divided into three groups, SPL elements (ceciis2013 product and ceciis2013 platform), external elements (itext.zip, jxl.jar) and environment element (rt.jar).

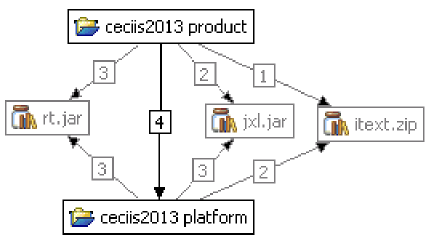

Figure 7. SPL Product dependencies

We assume there are no mutually dependent elements, since they would be an early sign of not using even the basic principles of the SPL approach. A quick glance at this graph (Figure 7) shows 4 dependencies between SPL Product and SPL Platform, 3 dependencies between SPL Product and external components, 3 dependencies between SPL Platform and rt.jar, 5 dependencies between SPL Platform and external components and 3 dependencies between SPL Product and rt.jar.
Figure 8 show the difference between Efferent Coupling (Ce) defined by Robert Martin and D3 metrics which we use here. According to Ce (the number of classes inside the category that depend upon classes outside the category) the counter would be 3, but since we use number of references instead of number of classes and we may count 4 dependencies. Also, we count references at the first level of abstraction which is a number of relationships among the classes. The second level dependency where a number of class and data references are counted is not used here. Number of first level references (we count arrows) from left to right side classes as shown below is four while number of second level references where class and data references are included is five (2,1,1,1).

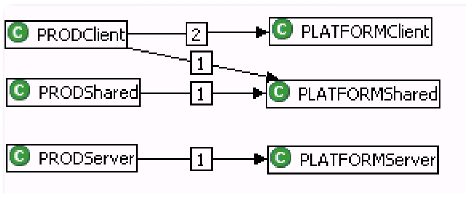

Figure 8. SPL Product class references

PR: *Platform Responsibility* calculated from the elements on figure 7 is shown in table 3. Calculation of the *PR* metric: D4+D5 / (D3+D4+D5) = 3+3/ (4+3+3) = 0,6 shows that product derived from the platform is instable while the platform framework is not fully responsible for interactions with the external components and the environment API.

Dependency (D1)	3
Dependency (D2)	5
Dependency (D3)	4
Dependency (D4)	3
Dependency (D5)	3
Platform Responsibility [0-1]	**0,60**

Table 3. PR calculation

The dependency metrics, proposed by Robert Martin, measure the responsibility, independence and stability of a category. According to Martin, a category can be at different levels of granularity: projects, packages, or types. In the context we analyze here in this paper we add to the levels, and we view a software product line business application (product) as a level of granularity. Also, the common components of a product line which are produced within the domain engineering process in the form of the SPL platform framework are viewed as a category in the context of dependency analysis. Categories in this paper are the SPL platform framework and the products derived from it.
The most responsible product lines are those that are both (D4, D5) independent and responsible (D3).
Figure 9 illustrates the case where the product dependencies on external components are transferred to the platform framework. Platform framework takes the responsibility of interactions to the external components and helps to make a product more stable.

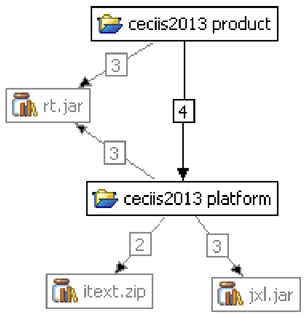

Figure 9. SPL Product dependencies

Calculation of the PR metric for a product derived from a product line (Figure 9):

PR =D4+D5 / (D3+D4+D5) = 3+0 / (4+3+0) = 0,43

The calculated result shows that product derived from the platform is stable while the platform framework is responsible for interactions with external components.

Dependency (1)	3
Dependency (2)	5
Dependency (3)	4
Dependency (4)	3
Dependency (5)	0
Platform Responsibility [0-1]	**0,43**

Table 4. PR calculation

We can calculate a total *Platform Responsibility* (PR) for a product line platform framework by taking in account all of the products spawned from it through the following equation:

$$PR = \sum_{i=1}^{n} \frac{D5_i + D4_i}{D3_i + D5_i + D4_i}$$

where n is the number of products spawned from the platform framework. The PR can be calculated for each product or for all of the products spawned from the product line. Table 5 shows the calculation of the PR for three products (P1, P2, and P3). The early stage for a product line ends when the third product has been developed. At that time, the calculation of total “responsibly” for all products may be used as a validator for the platform framework. Let’s assume that PR <= 0,5 indicates that platform framework is responsible enough and the new products may be developed. In case PR > 0, 5 it indicates that the PR measure of platform framework does not indicate a new products based on it should be developed before the platform is improved.

	D3	D4	D5	PR
P1	4	3	3	0,60
P2	4	3	0	0,43
P3	4	0	0	0,00
Total	12	6	3	0,43

Table 5. Total Platform Responsibility

The proposed metrics may be analyzed within the framework of measurement theory such as the Distance framework [15] and framework based on desirable properties which serves guidance provided to define proper measures for specific problem [6]. These frameworks ensure that the metrics developed using these guidelines are tested to be valid and that they can be used as measurement instruments. The Distance framework proposes a set of mandatory properties, such as: *identity, non-negativity, triangular inequality and symmetry* that are mandatory for any metric in order to be considered as an acceptable measure. Property-based measurement framework provides a set of properties for metric types such as: complexity (*additivity, identity, monotonicity, non-negativity, and symmetry*), length (*identity, monotonicity, non-negativity, null-value*) and size *(null value, aditivity, non-negativity*).
Table 6 shows that proposed metrics respect the four mandatory properties required by the Distance framework. Furthermore, Table 6 shows that metrics for size respect the properties defined by property-based measurement framework.

Properties	**Size**	**Complexity**
	D1-D5	**PR**
Non-negativity	Y	Y
Null value	Y	Y
Symetry	Y	Y
Non-increasing monotonicity	NA	Y
Identity	Y	Y
Non-decreasing monotonicity	NA	Y
Additivity	Y	Y
Triangular inequality	Y	Y

Table 6. Theoretical properties of defined metric

The complexity metrics defines five desirable properties while the metrics respect all five of them.
Given the introduction of a set of theoretically valid software metrics for software product line platform responsibility, an empirical validation of their usefulness can be done in the future research work.

5 Related works

The major research in the area of product line architecture have been done by [16] where they have developed a class of closely related metrics that specifically target product line Architectures such as metrics base on *Provided Service Utilization (PSU)* and *Required Service Utilization (RSU)*. The metrics are based on the concept of service utilization and explicitly take into account the context in which individual architectural elements are placed. However, such metrics are based on concept of service that is defined as any public accessible resource and do not consider dependencies on external components and its influence on the quality of the final products
Rahman [11] proposes a component based product line architecture metrics to measure PLA quality attributes like observability, customizability, interface complexity, modularity, service utilization, and maturity. This metrics can contribute to better understand the product line quality attributes but do not measure the “responsibility” of core assets from the perspective of spawned products.
Aldekoa [17] extended the Maintainability Index where the maintainability index of each features is measured. The metric is based on the average of the McCabe’s Cyclomatic Complexity value. However, it is based on the generated code and not on the structure of dependencies among set of classes which are used by final product spawned from a software product line.
Oliviera et al. [1] proposed a measurement suite for product line complexity quality attributes (ComplPLA) and empirically validated them. The proposed metrics measure complexity of interfaces, variation points, variability, but does not address the platform framework stability and responsibility contribution to the over all stability and maintainability of products.

5 Conclusions

A sound software product line for business application’s architecture in the form of platform framework is the main foundation to build the products within time, quality and budget constraints. Since platform framework serves as base for deriving many applications from a software product line, its quality influences the final properties of the developed applications. Therefore it is important to consider ensuring the quality of platform framework at the early stage of its development. The early stage of product line architecture is the period when platform framework is developed from scratch or from initial products. The platform framework is

developed all the time product line is alive. The external quality of the final products spawned from the product line depends on the "responsibility" of its platform framework. For this reason, we have proposed and theoretically validated a platform "responsibility" and its related metrics.
Given the introduction of a set of theoretically valid software product line metrics for evaluation of its platform framework, in the future we will do an empirical validation of their usefulness for quality assessment in practice. The advanced statistical analysis techniques will be used to evaluate the efficiency of these metrics for external quality attributes prediction such as stability, maintainability, error prediction, and etc.

References

[1] I. M. G. J. C. M. Edson A Oliveira Junior, "Empirical Validation of Variability-based Complexity Metrics for Software Product Line Architecture," 2001.

[2] C. Parra, "Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations," 2011.

[3] D. C. S. A. G. J. G. Y. L. G. L. Gan Deng, "Evolution in model-driven software product-line architectures," 2008.

[4] Z. Roško, "Strategy Pattern as a Variability Enabling Mechanism in Product Line Architecture," 2012.

[5] K. a. G. B. Pohl, "Software product line engineering: foundations, principles, and techniques," 2005.

[6] L. C. S. M. a. V. R. B. Briand, "Property-based software engineering measurement," *Software Engineering, IEEE Transactions on 22.1,* pp. 68-86, 1996.

[7] D. G. Ebrahim Bagheri, "Assessing the maintainability of software product line feature models using structural metrics," 2011.

[8] M. N. NE Fenton, "Editorial-Software metrics: Successes, failures and new directions," *Journal of Systems and Software,* pp. 149-158, 1999.

[9] J. Bansiya, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering,* pp. 4-17, 2002.

[10] R. Martin, "OO design quality metrics. An analysis of dependencies.," 1994.

[11] A. Rahman, "Metrics for the Structural Assessment of Product Line Architecture," 2004.

[12] N. M. a. A. v. d. H. Ebru Dincel, "Measuring Product Line Architectures," *Software Product-Family Engineering,* pp. 151-170, 2002.

[13] L. D. J. W. Q. Z. C. M. Tao Zhang, "Some Metrics for Accessing Quality of Product Line Architecture," *International Conference on Computer Science and Software Engineering,* 2008.

[14] T. B. Nachiappan Nagappan, "Using software dependencies and churn metrics to predict field failures: An empirical case study," *Empirical Software Engineering and Measurement, 2007. ESEM 2007.,* pp. 364-373, 20 9 2007.

[15] G. a. G. D. ". a. f. f. s. m. c. D. R. R. 9. (. 1.-4. Poels, "DISTANCE: a framework for software measure construction," *DTEW Research Report 9937,* pp. 1-47, 1999.

[16] A. E. D. a. N. M. Van Der Hoek, "Using service utilization metrics to assess the structure of product line architectures," *Software Metrics Symposium, 2003. Proceedings. Ninth International. IEEE.,* pp. 298-308, 2003.

[17] G. T. S. S. G. D. O. U. M. A. G. .. &. D. Aldekoa, "Experience measuring maintainability in software product lines," *Jornadas de Ingenieria del Software y Bases de Datos,* 2006.

[18] C. K. F. B. J. M. G. S. a. T. L. Thomas Thüm, " FeatureIDE: An Extensible Framework for Feature-Oriented Software Development," *Science of Computer Programming,* 2012.

[19] W. Harrison, "Software measurement: a decision-process approach," *Advances in Computers, volume 39,* p. 51–105, 1994.

[20] F. S. Mohammad Ali Torkamania, "Some metrics to evaluate reusability of software product line architecture," *2nd World Conference on Information Technology (WCIT-2011),* 2011.