

KEY CONCEPTS - DOMAIN DEVELOPER

This document is an integral part of the training and presents a theoretical synthesis of the main information for understanding the SMartyPerspective for the Domain Developer scenario: Software Product Line, SMarty approach and software inspection.

1. Software Product Line (SPL)

An SPL is defined by Northrop (2002) LPS as set of software systems that share common and variable characteristics, which satisfy the needs of a particular market segment developed from common artifacts.

SPL Engineering establishes the steps, processes and methodologies for using SPL in software development. It has two essential activities: (i) Domain Engineering, a process in which there is the development and evolution of the core of artifacts, (ii) Application Engineering, in which the configuration of a specific product takes place.

Variable features define the variable software, that is the capacity or ability of a system or device be efficiently extended, changed, or custom configured for use in a particular context (Van Gurp et al., 2001).

Variability is described by variation points and variants:

- **Variation Point** A specific location in an artifact where a design decision has not yet been made, that is, it has been postponed;
- **Variant:** Corresponds to a design alternative to solve a given variability.
- **Constraints between Variants:** Defines the relationships between two or more variants so that you can resolve a point of variation or variability.

Variability management is one of the most important activities in controlling an SPL. It includes activities to identify variability, explicitly represent it in software artifacts throughout the lifecycle, and trace dependencies between variability (Pohl et al., 2005). Therefore, the success of an SPL is related to the organization's ability to manage variability and produce customized systems that stand out in the market.

a. Features Diagram

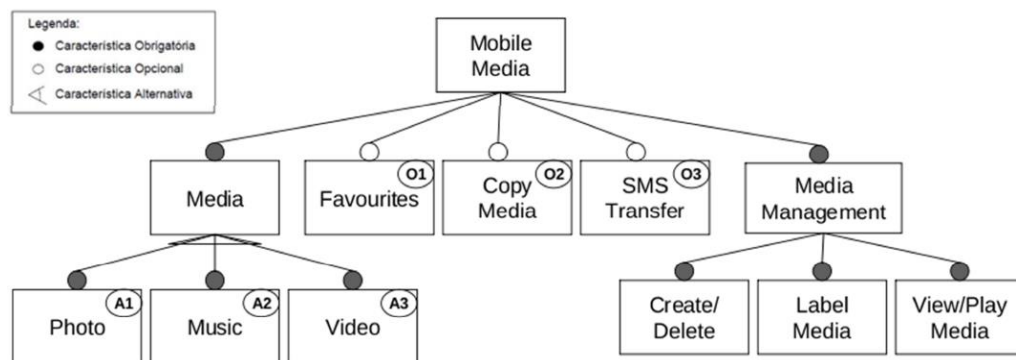
The feature diagram allows you to hierarchically model the common and variable characteristics of all products that can be derived from an SPL, and can thus be understood by all stakeholders in the development process, including the customer.

A feature can be (Benbassat, 2017):

- Mandatory: the feature must be present in any SPL configuration (filled circle);
- Optional: it may or may not be present in an SPL application (empty circle);
- Alternative: only one of the features must be chosen for the SPL configuration (empty arc); and
- OR: it is possible to select all the variant features for the product configuration (filled arc).

In Figure 1, the SPL Mobile Media feature diagram is presented, which is composed of applications that manipulate music, videos and photos for mobile devices. (Geraldi and OliveiraJr, 2017).

Figure 1: SPL Mobile Media feature diagram



Font: (Contieri Junior, 2010)

b. SMarty Approach

The SMarty (Stereotype-based Management of Variability) approach was proposed for the management of SPL variability based on UML models (OliveiraJr et al., 2010). It is in version 5.2 with support for diagrams of: use case, class, activity, component and sequence.

SMarty allows the management of the variability of an SPL in a systematic way through the profile (SMartyProfile), which graphically and visually represents the variability through the UML and a process (SMartyProcess), which defines guidelines to guide the user in identification, representation and tracing variability (OliveiraJr et al., 2010).

The SMartyProfile is a set of stereotypes and meta-attributes (Table 1) that represent the main concepts of variability management: variability, point of variation, variant and variant constraints in UML models for SPL.

Table 1: Stereotypes of the SMarty Approach

Stereotype	Summary
<<variationPoint>>	Represents the place where variability occurs. A variation point is always associated with one or more variants.
<<mandatory>>	The variant must be present in the configuration of any SPL model.
<<optional>>	The variant may or may not be present in the SPL configuration. Optional variants may or may not be associated with a variation point as well.
<<alternative_or>>	They are always associated with the variation points. At least one of the variants must be chosen to resolve the variation point.
<<alternative_xor>>	They are always associated with the variation points. Only one of the variants must be chosen to solve the variation point.
<<variability>>	Indicates existing variability in a UML model and is only applied to UML grades. Associated with this stereotype has the meta-attributes: name (name of variability), maxSelection and minSelection (maximum and minimum number of variants to be selected to resolve a variation point), bindingTime (defines the moment at which a variability will be resolved), allowsAddingVar (displays whether there is the possibility of adding new variants after a variability is resolved), variants (set of instances associated with the variability); realize- and realize+ (set of lower and higher-level model variability, respectively, which performs the variability)
<<requires>>	it is a relationship between variants, where the chosen variant requires another related variant;
<<mutex>>	represents the concept of restriction between variants. If a variant is selected, then the dependent variant must not be in the SPL configuration;

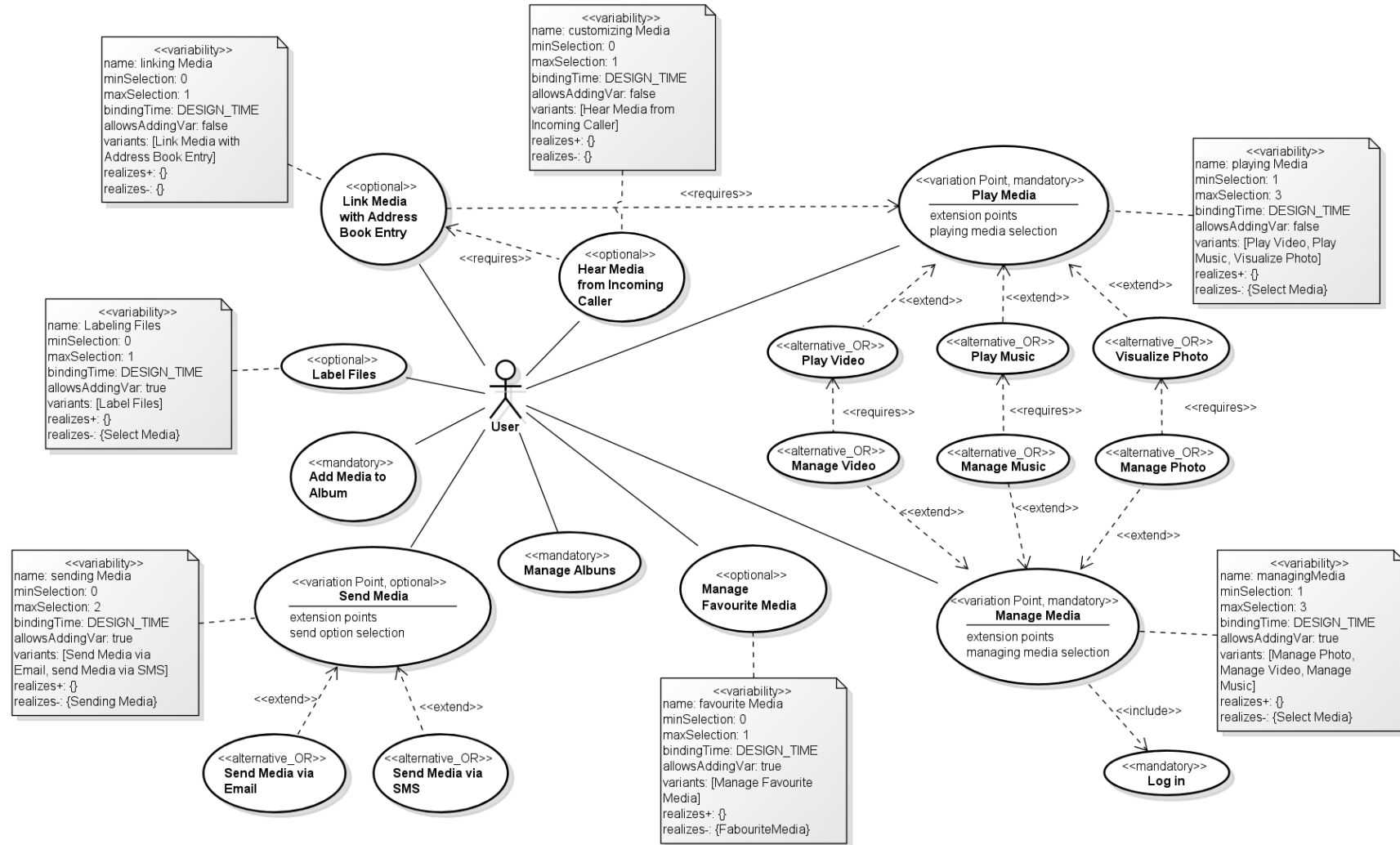
Font: adapted de Geraldi and OliveiraJr (2017)

An example of the use of stereotypes of the SMarty approach can be seen in Figure 2. In Figure 3, a part of the SPL Mobile Media use case diagram in Figure 2 is presented. stereotype of the SMarty <<mandatory>> approach. This means that **Manage Albums** is a mandatory feature in any valid configuration for this SPL.

The **Send Media** use case is an optional feature (<<optional>>) for any SPL configuration, so it is up to the user to define whether or not it will be part of the configuration of their specific application. In addition to being an optional feature, **Send Media** is also a variation point (<<variationPoint>>). To solve it, two inclusive variants were defined: **Send Media via SMS** and **Send Media via Email**.

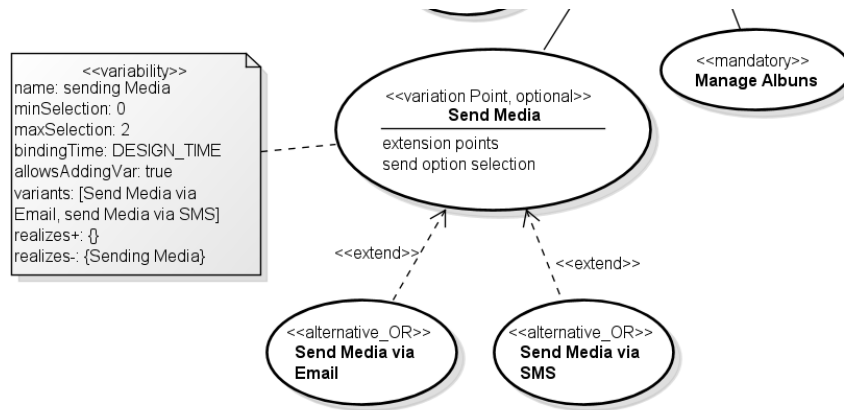


Figure 2: SPLMobile Media use case diagram based on SMarty approach



These variants were defined with the `<<alternative_OR>>` stereotype, so at least one of the variants must be chosen to resolve this variation point. Both variants can even be chosen for a specific application. If, instead of `<<alternative_OR>>`, the stereotype was `<<alternative_XOR>>`, only one of the variants could be chosen: sending media by SMS or by email, but not both.

Figure 3: Example of using SMarty approach stereotypes



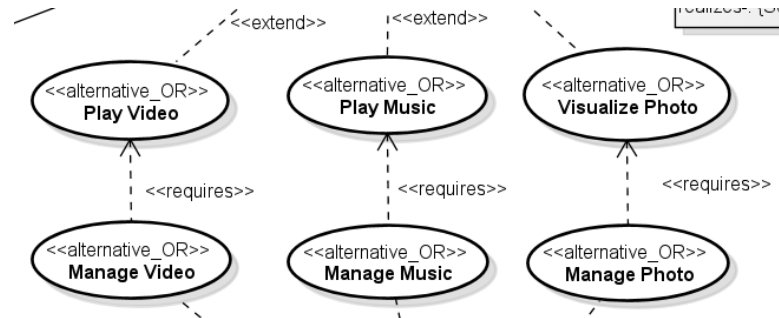
Still in Figure 3, it is possible to observe the UML note contains the variability information in the SMarty approach with the stereotype `<<variability>>`, which represents a variability, and its meta-attributes associated with the Send Media variation point. From the meta-attributes it is possible to extract the following information:

- The name of the variability is sending media. The name is important for tracing variability, therefore, it is how we identify traceabilities that are traceable among the artifacts.
- The minimum selection number of variants is zero, because although variants are of type `<<alternative_OR>>`, which require selection of at least one of the variants, the Send Media element is optional. If this functionality is not present in the product configuration, its variants are also not present. Therefore, **minSelection** must be 0, not 1. **MaxSelection**, on the other hand, has a value of 2, because, if the function is in the configuration, the user can choose one or two variants for a specific application.
- In the meta-attribute variants, the variants related to the use case **Send Media**, **Send Media via SMS** and **Send Media via Email** were defined. This information must be consistent with the number of variants shown in the diagram.
- The realizes- set, for this example, contains the variability realization for the SPL MM class diagram, as the class diagram is less abstract than the use case one. Therefore, the sending media variability must be named in the class diagram in this way to have traceability between the variability.

In Figure 4, the relationship between **Manage Video** and **Play Video** is of type `<<requires>>`, which means that, if the configuration has **Manage Video** functionality, then

it must also have Play Video. The same goes for **Manage Music and Play Music, View Photo and Manage Photo**. If the relationship were of type `<<mutex>>`, then a product that has **Manage Video**, could not also have **Play Video**.

Figure 4: Example of using the stereotype `<<requires>>`



2. Software Inspection

Software quality assurance is a structured approach that involves defect containment, prevention, detection, and removal activities. To this end, verification and validation (V&V) activities help to quantify the quality assurance process by considering that the process was not conducted correctly and that there is a possibility that the product contains defects that must be corrected (Koscianski and Soares, 2007).

Verification is a technique that checks that the software is being developed according to the user's specifications. To this end, software review can be applied to verify the quality of a product, acting as a “filter” to purify the artifacts generated in the software process by discovering errors that can be removed. Its need is allied to two facts: (i) artifacts are generated by people and so, there can be errors; and (ii) errors are hardly detected by those who create the artifact, thus requiring reading by someone else, the reviewer (Pressman and Maxim, 2016).

Software inspection is considered a type of review, which guides the reviewer in reading the artifact through a set of instructions. It is essential “to understand a given representation of the software artifact and compare it to a set of expectations regarding structure, content and quality” (Biffl and Halling, 2003).

It can be applied at the end of all software development steps as soon as an artifact is created, which reduces rework costs, as defects are usually found close to the point where they were inserted (Höhn, 2003) and if they pass. for the next activity they can cost up to ten times more (Zhu ,2016).

To support the reviewer when reading a document in practice, there are reading techniques, which provide a series of instructions for the reader on how to read and what to look for in the artifact (Basili et al., 1996). They provide a general understanding of the inspection activity through the process characteristic of the chosen technique and the defect classes covered by it for the verification of quality attributes. The main ones are: Ad hoc and Checklist-Based Reading and Scenario-Based Reading.

a. Scenario-Based Reading

Scenario-Based Reading (SBR) was designed under two key factors (Lanubile et al., 2004): (i) active orientation, as the reader actively works with the document while being oriented in reading; and (ii) separation of concern, as it restricts the reader's focus to what he should inspect according to the specific aspect of the document of interest.

SBR uses the concept of relationship between operational scenarios based on the premise that if each of the inspectors uses different systematic techniques and reads about a certain specific perspective of the project, when they put all the scenarios together, they will have greater coverage of the document, combination of scenarios (Zhu, 2016) and classes of defects.

Among the SBR family techniques, the following stand out: Defect-Based Reading, in which each reader searches for a specific defect class and the checklist items are replaced by procedures designed to implement them, and Perspective-Based Reading, which the reader looks for defects under specific objectives of the document's consumers (Zhu, 2016).

b. Perspective-Based Reading

An artifact can support different users with specific needs to perform their tasks during the development process. For example, from the same requirements document, a user checks whether the described requirements adequately capture the functionality he needs, while a designer uses it as a basis for development, and the tester defines a test plan to ensure that the software complies with the functional and performance requirements described (Höhn, 2003).

Perspective-Based Reading (PBR) is a family of reading techniques that address the different objectives of the artifact's consumers, such as the perspectives of a developer, user and tester. Each reader is responsible for a particular perspective, and by putting them together, they will have greater coverage of the document from different perspectives of those involved (Zhu, 2016), avoiding the overlapping of defects between perspectives and can be applied throughout the lifecycle from the project.

PBR advises readers to ask "what information in these artifacts should be verified?" and "how to identify defects in this information?" (Shull et al., 2000) through scenarios. A scenario is a collection of procedures that operationalizes strategies for detecting defects (de Mello et al., 2014), through active reading of the artifact. It is "developed based on knowledge about the environment in which the reading process is applied: roles in the software development process; and defect classes." (Ciolkowski et al., 1997).

The benefits of the technique can be defined by the attributes that characterize the technique (Shull et al., 2000):

- **systematic:** provides a definitive procedure to guide how the artifact should be read and verified;

- **focused:** the different perspectives allow readers to focus on specific defects types for each scenario, thus contributing to the efficacy of the technique, as it avoids the duplicated effort between team members by detecting only defects related to their particular point of view;
- **objective-oriented and customizable:** perspectives are chosen by the organization to reflect the requirements uses and the specific objectives of the inspection, adapting the procedure to its needs; and
- **transferable through training:** as the PBR depends on the procedure defined by the organization and not only on the reviewer's experience, a new reviewer can receive training in the steps of the procedure. SMartyPerspective

Domain Engineering is where the greatest effort of an SPL is concentrated, therefore, the artifacts produced in this activity must contain a high level of quality, as they supply the artifact core and from them the products will be configured in Application Engineering as needed. of a specific customer. Thus, a defect incorporated in this phase could spread to all SPL products.

SMartyPerspective is characterized as a family of inspection techniques based on PBR for inspection of feature diagrams and UML SMarty of use case, class, component and sequence to ensure, as much as possible, the quality of these artifacts in the variability management process in the Domain Engineering activity.

The operational SMartyPerspective' scenarios family of techniques were defined on two bases: (i) the similar defect taxonomies of Travassos: Ambiguity, Incorrect Fact, Inconsistency, Extraneous Information and Omission; and (ii) the roles in Domain Engineering of Van der Linden et al. (2007): Product Manager, Domain Requirements Engineer, Domain Architect, Domain Developer, Domain Asset Manager.

The roles served as a model for the generation of the scenarios, as the perspectives, introductions and expected quality levels of the artifacts to carry out their tasks were defined from the study of each of them. The questions for each of the operational scenarios were defined and derived from the taxonomy defect classes that encompass the main types of defects found in the SPL diagrams, context and particularities.

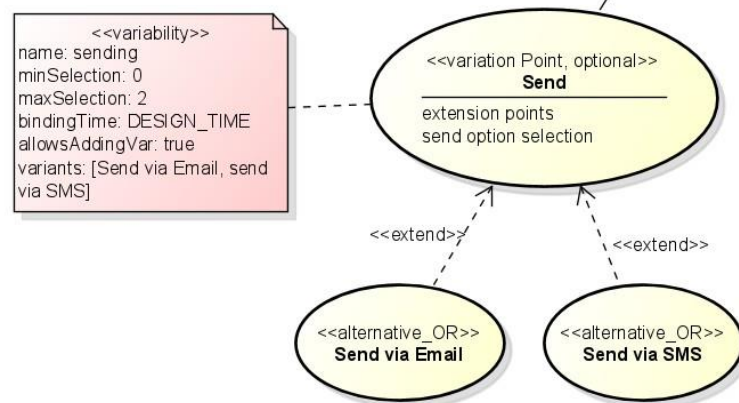
c. SMartyPerspective Defect Taxonomy

SMartyPerspective's defects taxonomy considering the definitions for the types of defects in the work of Travassos et al.(1999) for OO models based on the classification of Basili et al.(1996), since the original similar studies (Shull et al. (1998) e Basili et al. (1996)) were specific for requirements defects.

Defect classes will be exemplified by injecting defects in part of the use case diagram in Figure 2 for the **Send Media** use case and its related variants. SMartyPerspective's defect classes are shown below:

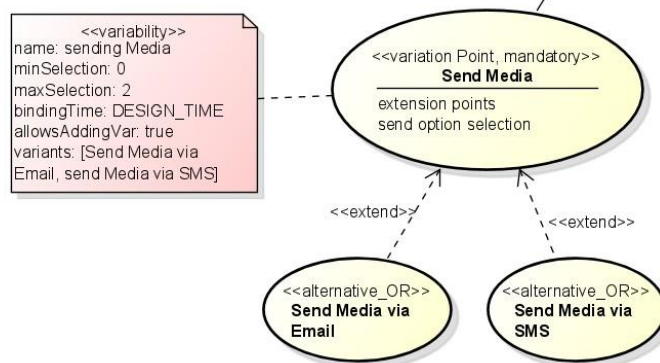
- **Ambiguity:** refers to information that has not been specified clearly enough to understand and can generate multiple interpretations of the same element for each different reader of the diagram. An example for this defect can be seen in Figure 5, as the use case was defined only as “**Send**”. Therefore, it is not clear enough what should be sent and/or by what means, depending on the reader's interpretation of the diagram.

Figure 5: Example of an Ambiguity Defect



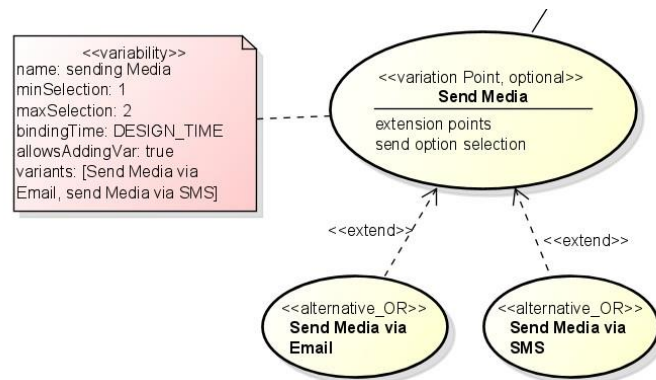
- **Incorrect Fact:** when an information depicted in the diagram is in disagreement with the requirements specification and/or knowledge of the SPL domain. In Figure 6, **Send Media** has been defined as mandatory, that is, it should be included in all SPL configurations, when in fact it is an optional element and should only be included in the configuration as per the user's needs.

Figure 6: Example of an Incorrect Fact defect



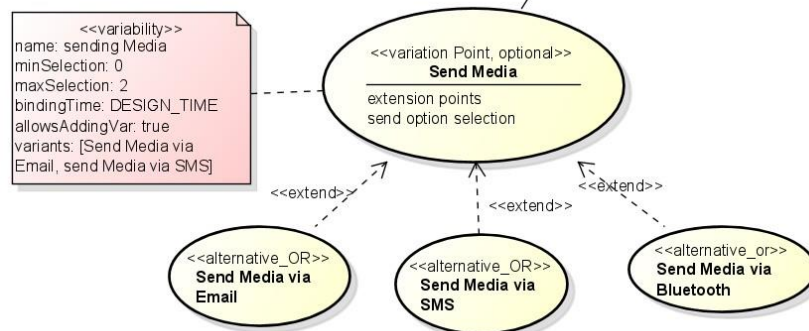
- **Inconsistency:** happens when the information of a diagram element under inspection is not consistent with another element of the same diagram or another artifact. In Figure 7, the minSelection meta attribute was set to 1, however, this would force the client to choose one of the variant options for the element, but Send Media is an optional element and may not be included in a configuration SPL and none of the variants will be like this, in the configuration if the element is not also, then minSelection=0)

Figure 7: Example of an Inconsistency defect



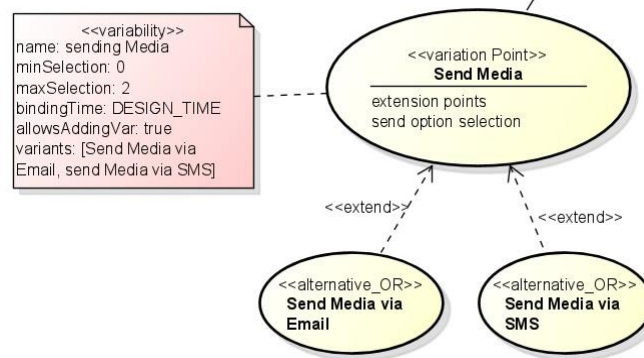
- **Extraneous Information:** refers to information that was included in the diagram, but that is not necessary for the modeled context or does not belong to the SPL domain, that is, the information is surplus in the artifact. In Figure 8, the **Send Media via Bluetooth** use case was described in the requirements document that it is possible to send media via email or SMS, but not via bluetooth. Therefore, this element does not belong to the SPL MM domain and is “left over” in the diagram.

Figura 8: Example of a Extraneous Information Defect



- **Omission:** refers to required information that has been omitted from the diagram under inspection, whether it is an element or a stereotype that has not been defined in the diagram. The **Send Media** element was specified in Figure 9 without the SMarty stereotype that defines whether it is mandatory or optional. That way, the reader of the diagram won't know if sending media is mandatory or not if it's based only on this artifact.

Figure 9: Example of a Omission Defect



d. SMartyPerspective Perspectives

SMartyPerspective is composed of five scenarios (perspectives) that differ from each other by the inspected diagrams and defects types found in each of them, for example, the class diagram can have a conceptual view for the Domain Requirements Engineer, a design vision for the Domain Architect and implementation for the Domain Developer.

Thus, the scenario for each of the perspectives differs in the elements that must be inspected, avoiding the exhaustion and unnecessary effort of having to inspect the entire document and its diagrams, when only a part is necessary for its function. The perspectives that can be taken on by an inspector at SMartyPerspective are:

- ● **Product Manager:** responsible for planning and evolving the portfolio in accordance with the organization's objectives;
 - inspects feature and use case diagrams;
- **Domain Requirements Engineer:** analyzes and identifies common and variable requirements for the asset base;
 - inspects the use case, class (conceptual view) and sequence diagrams;
- **Domain Architect:** validates that the reusable asset designs meet the LPS architecture and determines the configuration devices that will be used in the applications.
 - inspection of class (design view) and component diagrams;
- **Domain Developer:** implements the components that will integrate the core asset, that is, there is no implementation of a run application, only loosely coupled components;
 - inspects the class (implementation view), component and sequence diagrams;
- **Domain Asset Manager:** maintains a valid configuration and version of domain assets and their traceability to be used at all stages of development;
 - inspection of feature, use case, class, component and sequence diagrams.

e. SMartyPerspective Scenario

A scenario for PBR, as well as for SMartyPerspective, consists of three sections:

- **introduction:** contains a brief description of the perspective, the role the instructor will assume during the inspection activity, and the most relevant quality attributes.

- **instructions:** describe the activities that will be performed by the inspector during the inspection and how they should be done. Questions help break down the document into smaller parts; and
- **questions:** allow inspectors to judge the information read in the document, verifying whether it fulfills a set of quality factors previously determined through a series of questions about specific aspects of the information (Laitenberger and Kohler, 2001).

SMartyPerspective's Initial Introduction guides the reader when finding a defect by entering in the Defect Identification Form (FID) table some data about the element and which defect found so that it can be corrected later (Figure 10):

- **Diagram:** the reader marks the cell corresponding to the diagram under inspection: use case (UC), class (CL), component (CP), sequence (SQ) and features (FT);
- **Question Number:** the questions are listed according to the step they belong to. Therefore, the number of the question that guided the reader to detect the defect must be indicated on the form;
- **Element:** a diagram is composed of several elements that characterize it. In this cell, the name of the element in which the defect was found must be informed; and
- **Identified defect:** in this cell, the inspector puts the defect found for that element and diagram. It is also possible to inform in this cell a solution to correct the defect to facilitate the next phases of the inspection.

Figure 10: DIF - Defect Identification Form

n°	DIAGRAM					QUESTION NUMBER	ELEMENT	IDENTIFIED DEFECT
	FT	UC	CL	CP	SQ			
1								
2								
3								
4								

3. Domain Developer Perspective Scenario

The domain developer (DDP) is responsible for implementing all components and interfaces that will be reused by the SPL product range and that will be configured and executed as an application in the Application Engineering activity depending on the customer's choices.

The DDP must then be concerned with the different contexts of the SPL and develop the components and interfaces that meet all the needs and all possible configurations of the

SPL. Its main difference in relation to the traditional software process for single systems is that in Domain Engineering the realization is loosely coupled, as there is no implementation of a working application, only the components that will integrate the core asset (Van der Linden et al., 2007; Pohl et al., 2005).

The DDP perspective should review the diagrams that allow it to have an overview of the SPL, with information about the functionality it offers and the activities that must be performed by a user so that he can implement the interfaces and components of the platform. Therefore, the diagrams that provide this vision for the role are the class, sequence, and component diagrams.

The DDP scenario is composed of 8 steps, as shown in Figure 10. The complete scenario with the questions and introductions for each of the diagrams can be found in Document 5 of this instrumentation.

Figure 10: DDP ScenarioPerspective

DOMAIN DEVELOPER	
<p>This perspective should implement and test the components and interfaces that will be reusable throughout the product portfolio. You must develop according to the requirements of a range of products, for this you must ensure that the elements represented in the SMarty diagrams of class, sequence and component are not in disagreement with the requirements, contemplate the expected functions and allow you to have a vision implementation of reusable components.</p> <p>To diagrams correctly express the user requirements without inconsistencies, you must review such diagrams and elements. To achieve your goal, perform the steps outlined below to inspect each of the informed diagrams. When you find a defect in one of the steps, fill in the Defect Identification Form indicating the diagram, the step item (number question), and the element and the defect found.</p>	
LOCATE CLASS DIAGRAM AND REQUIREMENTS SPECIFICATION	
Step 1	inspection of use case diagram
Step 2	
LOCATE THE COMPONENT AND CLASS DIAGRAM	
Step 3	inspection of component diagram
Step 4	
LOCATE THE SEQUENCE DIAGRAM, CLASS AND REQUIREMENT SPECIFICATION	
Step 5	Inspection of sequence diagram
Step 6	
Step7	
Step 8	

4. References

- Benbassat, F. Evolução Segura de Linhas de Produtos de Software: cenários de extração de features. 67 f. Dissertação (Mestrado) - Curso de Ciência da Computação, UFPE, Recife, 2017.
- Basili, V. R.; Green, S.; Laitenberger, O.; Lanubile, F.; Shull, F.; Sørungård, S.; Zelkowitz, M. The empirical investigation of perspective-based reading. *Empirical Software Engineering*, v. 1, n. 2, p. 133–164, 1996
- Biffi, Halling. Investigating the defect detection effectiveness and cost benefit of nominal inspection teams. *IEEE Transactions on Software Engineering*, v. 29, n. 5, p. 385-397, 2003.
- Ciolkowski, M.; Differding, C.; Laitenberger, O.; Münch, J. Empirical investigation of perspective-based reading: A replicated experiment. Fraunhofer Institute for Experimental Software Engineering, Germany, 1997.
- Contieri Junior, A.; Aplicação de Métricas em Arquiteturas de Linhas de Produto de Software. 2010. 73 f. Monografia de TCC. Universidade Estadual de Maringá, 2010.
- Geraldi, R. T.; Oliveira Jr, E. Towards Initial Evidence of SMartyCheck for Defect Detection on Product-Line Use Case and Class Diagrams. *Journal of Software*, v. 12, p. 379–392, 2017
- Höhn, E. Técnicas de leitura de especificação de requisitos de software: estudos empíricos e gestão de conhecimento em ambientes acadêmico e industrial. Tese de Doutorado, USP, 2003.
- Lanubile et al. Assessing the impact of active guidance for defect detection: a replicated experiment. *International Symposium on Software Metrics*, 2004, pp. 269-278.
- de Mello, R. M.; Teixeira, E. N.; Schots, M.; Werner, C. M. L.; Travassos, G. H. Verification of Software Product Line Artifacts: a Checklist to Support Feature Model Inspections. *Journal of Universal Computer Science*, v. 20, n. 5, p. 720–745, 2014.
- Oliveira Jr, E.; Gimenes, I. ; Maldonado, J. C. Systematic Management of Variability in UML-based Software Product Lines. *Journal of Universal Computer Science*, v. 16, n. 17, p. 2374–2393, 2010.
- Pressman, R.; Maxim, B. *Software engineering: a practitioner's approach*. 8 ed. Palgrave macmillan, 2016.
- Pohl, K.; Böckle, G.; van Der Linden, F. J. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media, 2005.
- Northrop, L. M. Sei's software product line tenets. *IEEE Software*, v. 19, n. 4, p. 32–40, 2002.
- Shull, F. J. Developing techniques for using software documents: A series of empirical studies. Tese de Doutorado, University of Maryland at College Park, USA, 1998.
- Travassos, G.; Shull, F.; Fredericks, M.; Basili, V. Detecting defects in object-oriented designs: using reading techniques to increase software quality. *ACM Sigplan Notices*, v. 34, n. 10, p. 47–56, 1999.
- Van Gurp, J.; Bosch, J.; Svahnberg, M. On the notion of variability in software product lines. In: *Proceedings Working IEEE / IFIP Conference on Software Architecture*, 2001, p. 45–54.
- Van der Linden, F. J.; Schmid, K.; Rommes, E. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.
- Zhu, Y. *Software Reading Techniques: Twenty Techniques for More Effective Software Review and Inspection*. Apress, 2016.