A Code Smell Detection Tool for Compositional-based Software Product Lines

Ramon Abilio[1], Gustavo Vale[2], Johnatan Oliveira[2], Eduardo Figueiredo[2], Heitor Costa[3]

[1]IT Department - Federal University of Lavras (UFLA), Lavras, MG, Brazil
[2]Department of Computer Science - Federal University of Minas Gerais (UFMG)
[3]Department of Computer Science - Federal University of Lavras (UFLA)

ramon.abilio@dgti.ufla.br,
{gustavovale,johnatan,figueiredo}@dcc.ufmg.br, heitor@dcc.ufla.br

Abstract. *Software systems have different properties that can be measured. Developers may perform manual inspection or use software measure-based detection strategies for evaluating software quality. Detection strategies may be implemented in a computational tool and they perform detection faster. We developed an Eclipse plug-in called VSD (Variability Smell Detection) to measure and detect code smells in AHEAD-based Software Product Line.*

https://www.youtube.com/watch?v=M8VybWpcNI8

1. Introduction

Despite of the extensive use of software measures, isolated values of measures are not meaningful because they are too fine-grained. However, we can combine measures to obtain measure-based detection strategies to detect code smells, for example. Detection strategies are based on the combination of measures and thresholds using logical operators (AND and OR) [Marinescu, 2004; Lanza; Marinescu, 2006]. The values of thresholds can be represented with the labels: Low, Avg (average), and High, because real values may be different depending on the context [Marinescu, 2004; Lanza; Marinescu, 2006]. A measure-based detection strategy may be implemented in a computational tool and used to detect code smells faster than manual inspections, which are time-consuming.

Measure-based detection strategies have been used to localize code smells in Object-Oriented (OO) [Lanza; Marinescu, 2006] and Aspect-Oriented (AO) [Figueiredo et al., 2012] software, but they have not been applied to detect code smells in Feature-Oriented (FO) software. Feature-oriented programming (FOP) is particularly useful in applications where a large variety of similar objects is needed [Prehofer, 1997; Batory et al., 2003], such as in the development of Software Product Lines (SPL). SPL is an approach to design and implementation of software systems that share common properties and differ themselves by some features to meet needs of the market or of the specific clients [Pohl et al., 2005].

A *feature* may be defined as a prominent or distinctive user-visible aspect, quality, or characteristic of software [Kang et al., 1990] and can be implemented with different approaches, such as Compositional and Annotative. In the Compositional approach, features are implemented in separated artifacts, using FOP [Batory et al., 2003] or Aspect-Oriented Programming (AOP) [Kiczales et al., 1997]. AHEAD is a compositional approach based on gradual refinements, in which programs are defined as

constants and features are added using refinement functions [Batory et al., 2003]. Besides, classes implement the basic functions of a system (constants) and extensions, variations, and adaptations in these functions constitute the features (refinements). Features are implemented in modules syntactically independent of the classes and can insert or modify methods and attributes [Batory et al., 2003].

The definition of code smells and its detection strategies for OO and AO address mechanisms of those techniques, such as classes, methods, aspects, and pointcuts [Fowler et al., 1999; Lanza; Marinescu, 2006; Macia et al., 2010]. In SPL, there are smells which indicate potentially inadequate feature modeling or implementation. To emphasize the difference in the focus on variability, Apel et al. (2013) called them variability smells. A variability smell is a perceivable property of a SPL that is an indicator of an undesired code property. It may be related to all kinds of artifacts in a SPL, including feature models, domain artifacts, feature selections, and products.

Focusing on the variability smells related to implementation of features, Abilio (2014) adapted three traditional code smells - God Method, God Class, and Shotgun Surgery - to address specific characteristics of compositional-based SPL. The adaptation of the code smells was based on literature [Fowler et al., 1999; Lanza; Marinescu, 2006; Macia et al., 2010] and on analysis of a set of AHEAD-based SPLs. This adaptation was necessary because the traditional code smells do not address mechanisms of FOP, such as constants and refinements. Besides, detection strategies for those code smells were defined [Abilio, 2014].

Abilio (2014) described the code smells and detection strategies using the structure presented by Lanza and Marinescu (2006). Proposed measures to address specific mechanisms of FOP [Abilio, 2014] and measures indicated as useful to detect the traditional code smells [Lanza; Marinescu, 2006; Padilha et al., 2013; Padilha et al., 2014] were used for filtering. Low, Avg, and High values were calculated from a set of SPLs (not products). To measure source code of an AHEAD-based SPL and detect the proposed code smells, we developed the Variability Smell Detection[1] (VSD) tool as an Eclipse plug-in.

Software measures are key means for assessing software modularity and detecting design flaws [Blonski et al., 2013; Lanza; Marinescu, 2006; Marinescu, 2004]. The community of software measures has traditionally explored quantifiable module properties, such as class coupling, cohesion, and interface size, in order to identify code smells in software systems [Lanza; Marinescu, 2006; Marinescu, 2004]. For instance, Marinescu (2004) relies on traditional measures to detect code smells in object-oriented systems. Following a similar trend, Blonski et al. (2013) propose a tool, called ConcernMeBS, to detect code smells based on concern measures. However, as far as we are concerned, there is not tool to measure the source code and to detect code smell in AHEAD-based SPLs.

Therefore, the main goal of this work is presenting VSD by showing a high-level view of its architecture, the implemented measures and the detection strategies, and presenting its main functions (Section 2). Section 3 presents an example of the VSD use and discusses the results of a preliminary evaluation. Finally, Section 5 concludes this paper and suggests future work.

---

1 VSD source code is available in <https://code.google.com/p/vsdtool/>

2. Variability Smell Detection Tool (VSD)

This section presents VSD, a tool to detect code smells in AHEAD-based SPL. Section 2.1 presents the VSD architecture highlighting the main components and their interaction. The implemented measures and an example of the detection strategies are summarized in Section 2.2, and the main functions are detailed.

To illustrate VSD use, we measured and detected code smells in TankWar SPL [Schulze et al., 2012]. This SPL is a game developed by students of the University of Magdeburg (German). It has medium size (~5,000 LOC), has 37 features (31 concrete features), and runs on PC and mobile phones [Schulze et al., 2012]. This SPL was chosen due to its size (lines of code and number of features) and because it was used in other studies [Schulze; Apel; Kastner, 2010; Apel; Beyer, 2011; Schulze et al., 2012].

2.1. VSD Architecture

Figure 1 presents a high level view of VSD architecture. We used Eclipse IDE 4.3 (Kepler) and FeatureIDE to develop VSD. FeatureIDE is an Eclipse-based IDE that supports feature-oriented software development for the development of SPLs [Thüm et al., 2014]. It integrates different SPL implementation techniques, such as FOP, AOP, and Preprocessors, and provides resources to deal with AHEAD projects.

VSD has 3,5 KLoC and its classes were distributed in three packages illustrated in Figure 1: i) Detection Strategies: contains classes that implement the detection strategies; ii) Measurement: contains measures and classes that perform the measurement; and iii) Plugin: contains classes responsible to interact with Eclipse.

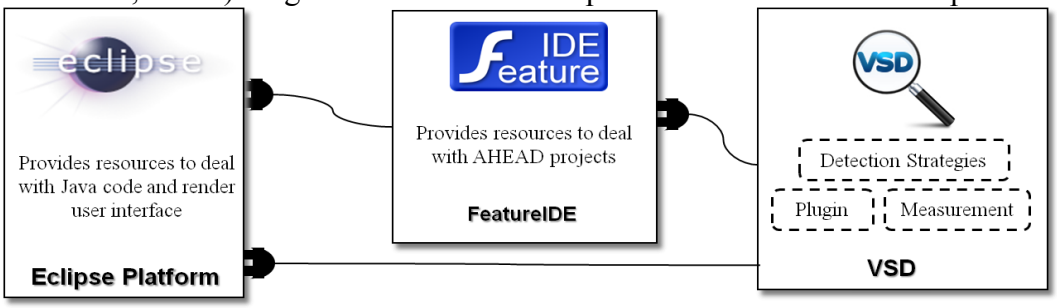

Figure 1 – High Level View of VSD Architecture

2.2. Implemented Measures and Detection Strategies

To measure an AHEAD-based SPL source code, VSD implements traditional, OO, and FO measures (Table 1), such as McCabe's Cyclomatic Complexity (Cyclo) [McCabe, 1976], Weighted Methods Per Class (WMC) [Chidamber; Kemerer, 1994], and Number of Constants (NOCt) [Abilio, 2014], respectively. As AHEAD uses the programming language Jak (a Java superset), traditional and OO measures assess properties of OO source code and FO measures assess the properties added by the mechanisms that Jak provides to implement features. We organized the measures in three groups: i) Method group: measures related to individual properties of methods; ii) Component group: measures for properties of components (classes, interfaces, constants, refinements); and iii) SPL group: measures to assess properties of source code of an entire SPL.

VSD implements three detection strategies, one for each code smell. When the user selects the option to detect some code smell, VSD performs a measurement, executes the respective detection strategy based on obtained values, and shows the

methods/components found with the selected code smell symptom. For example, the detection strategy for God Method is based on two main characteristics [Abilio, 2014]: i) Methods that may concentrate responsibilities, represented by method overrides, i.e., complete override and refinements; and ii) Long and complex methods. To address these characteristics, four measures were selected: NOOr, NMR, MLoC, and Cyclo. The detection strategy is [Abilio, 2014]:

```
((NOOr + NMR) > HIGH) OR ((MLoC > AVG) AND ((Cyclo/MLoC) > HIGH))
```

Table 1 - VSD Measures

Groups	Measures	
Method	Method's Lines of Code (MLoC)	Number of Method Refinements (NMR)
	McCabe's Cyclomatic Complexity (Cyclo)	Number of Operation Overrides (NOOr)
	Number of Parameters (NP)	
Component	Lines of Code (LoC)	Coupling between Objects (CBO)
	Number of Methods (NOM)	Weighted Methods Per Class (WMC)
	Number of Attributes (NOA)	Number of Constant Refinements (NCR)
SPL	Number of Components (NOC)	Number of Features (NOF)
	Number of Constants (NOCt)	Total Lines of Code (TLoC)
	Number of Refinements (NOR)	Total Number of Methods (TNOM)
	Number of Refined Constants (NRC)	Total Number of Attributes (TNOA)
	Total Number of Method Refinements (TNMR)	Total Cyclomatic Complexity (TCyclo)
	Number of Refined Methods (NRM)	Total Coupling between Objects (TCBO)
	Number of Overridden Operations (NOrO)	Total Number of Operation Overrides (TNOOr)

2.3. Functions

The potential users of VSD are software engineers that want to measure an AHEAD-based SPL and detect some undesired behavior in the implementation of the SPL features. They can access VSD via pop-up menu showed after right-clicking an AHEAD project. The available options are `Detect Shotgun Surgery`, `Detect God Class`, `Detect God Method`, and `Measure`.

After selecting an option, VSD shows the results in the respective view: `VSD Shotgun Surgery`, `VSD God Class`, `VSD God Method`, and `VSD Measures`. Figure 2 depicts `VSD Measures` view that presents the measurement results in a tree view; thus, users can see the value per measure - e.g. Total of Method's Lines of Code is 3,910. Users can also expand each measure to check the value for a method / component - e.g. `malen()` method, from the `ExplodierenEffekt.jak` component and the `explodieren` feature, has 5 as Cyclomatic Complexity value. If the user selects `Save as CSV` button (on top-right position), VSD saves the results in three Comma-Separated Values (CSV) files - one file per group (Table 1) - in vsd-output folder.

`VSD God Method`, `VSD God Class`, and `VSD Shotgun Surgery` views are similar. Figure 3 depicts the `VSD God Method` view. Each view presents the respective strategy with threshold values centralized on top, Save as CSV button at the top-right corner, and one table. The table contains id, feature, component, and method (only, `VSD God Method`) names, indication if component/method is a refinement (Y =

yes, N = no), and measures. For instance, the strategy `((NOOr+NMR) > 2.39) OR ((MLoC > 10.09) AND ((Cyclo/MLoC) > 0.24))` is centralized on top of Figure 3. Line #2 presents the `Beschleunigung` feature, `Tank.jak` component, and `toolKontroller()` method. This method is a refinement (Y) and has: MLoC: 11, Cyclo: 6, NP: 0, NMR: 0, and NOOr: 0. If the user performs a double-click in a row, VSD opens the respective component in a code editor. In addition, if the user selects Save as CSV, result is saved in a CSV file whose identifier (name) is the code smell name (e.g., godmethod.csv) in vsd-output folder.

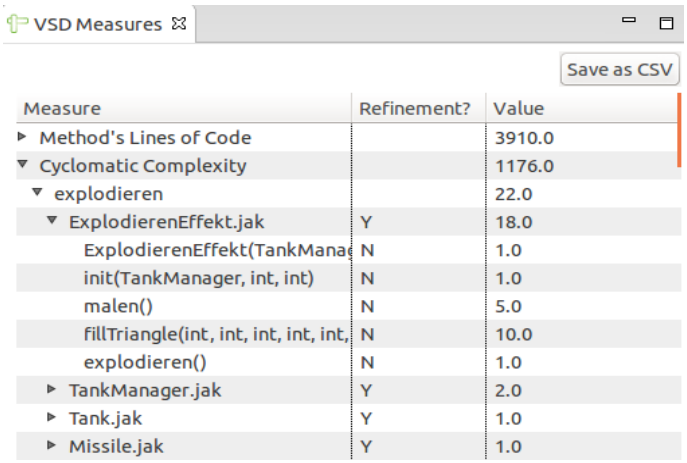

Figure 2 - VSD Measure View

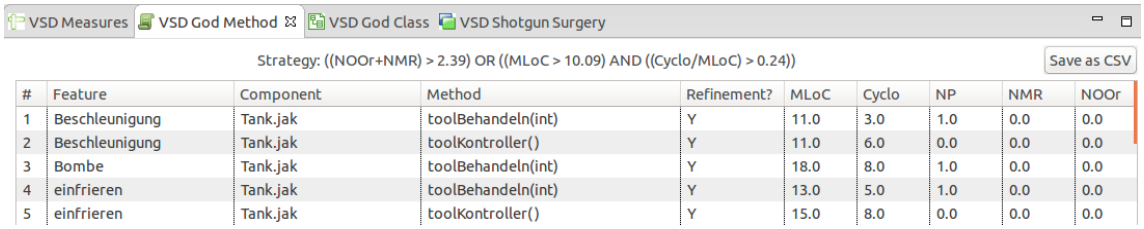

Figure 3 - VSD God Method View

The threshold values may vary depending on the selected (set of) project(s) (i.e., SPL). VSD has a preferences page as presented in Figure 4 - accessed via `Window -> Preferences -> VSD Preferences`. In the VSD preference page, users can define values for each measure in each strategy. The strategies with the acronym and name of each measure are presented. When the user saves or applies changes, VSD updates the values presented in the views and new values are used when the user performs a new detection.

3. An Example of VSD Use

We configured VSD with default threshold values, to detect the proposed code smells, based in empirical data of eight SPLs [Abilio, 2014]. Using those values, VSD found 64 methods with God Method symptoms in TankWar SPL. Table 2, Table 3, and Table 4 show samples of methods and components detected by VSD with code smell symptoms in TankWar SPL.

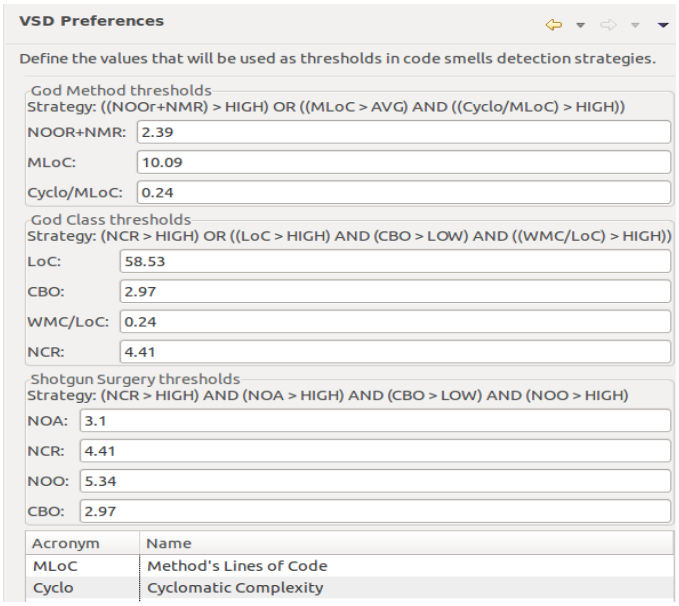

Figure 4 - VSD Preference Page

Table 2 - Sample of Methods with SPL God Method

Feature	Component	Method	Refinement	MLoC	Cyclo	NMR	NOOr
TankWar	Tank.jak	toolBehandeln(int)	N	1	1	8	1
Tools	Tank.jak	toolBehandeln(int)	N	1	1	8	0
Beschleunigung	Tank.jak	toolBehandeln(int)	Y	11	3	0	0
Bombe	Tank.jak	toolBehandeln(int)	Y	18	8	0	0
einfrieren	Tank.jak	toolBehandeln(int)	Y	13	5	0	0
Feuerkraft	Tank.jak	toolBehandeln(int)	Y	11	3	0	0

Table 3 - Sample of Components with SPL God Class

Feature	Component	Interface	Refinement	LoC	CBO	WMC	NCR
Handy	Maler.jak	N	N	300	12	63	16
PC	Maler.jak	N	N	304	18	58	16
fuer_Handy	Maler.jak	N	Y	320	8	104	0
fuer_PC	Maler.jak	N	Y	322	7	104	0

Table 4 - Sample of Components with SPL Shotgun Surgery

Feature	Component	Interface	Refinement	NOM	NOA	CBO	NCR
Handy	Maler.jak	N	N	32	10	12	16
PC	Maler.jak	N	N	31	14	18	16

The sample of methods (Table 2) addresses one method and its refinements. The `toolBehandeln(int)` method from the `Tank.jak` component was implemented as an empty method in the `TankWar` feature, which is the 'root' of the feature model. This method was re-implemented (overridden) in `Tools` feature, and refined in the `Beschleunigung`, `Bombe`, `einfrieren`, and `Feuerkraft` features. One possible problem is: if one adds some code in the first method (`TankWar` feature), it will be overridden and will not be used in the refinement chain because the second method (`Tools` feature) completely overrides it. That is, the first method was not refined. The other four refinements were detected with code smell because the density of branches (Cyclo/MLoC) is higher than the threshold. In fact, these methods seem to be simple, but the software engineer needs to pay attention to them.

The sample of components (Table 3) detected with SPL God Class shows that `Maler.jak` components are very similar between `Handy` and `PC` features and between `fuer_handy` and `fuer_PC` features. This occurred because `Handy` and `PC` are

alternative features and the selection of one of them implies the selection of only one "fuer". We identified three possible problems with `Maler.jak`. The first problem is that we have duplicated code, which is also code smell. The second one is that we have two constants and a large number of refinements adding behaviors that the developer does not know to which constant until the build of the product. Finally, the third problem is that the refinements are very complex and coupled to other components. Observing the code of the `Maler.jak` (`PC` feature), we noted that this component is responsible for screen, menus, behavior of menus and keys, and help items, for example. That is, this component concentrates many responsibilities.

Table 4 shows two constants of TankWar SPL detected with SPL Shotgun Surgery: `Maler.jak` from the `Handy` and `PC` features. These components were also indicated with SPL God Class and were detected with SPL Shotgun Surgery because they are coupled to many components and share many methods and attributes with many refinements. By a manual inspection in the code of `Maler.jak` (`PC` feature) and its refinements, we noticed that the refinements access directly the attributes of the constant, i.e., they do not use setter and getter methods. Hence, a change in the attributes may be propagated to the refinements. For example, `helpItemErstellen()` method instantiates the protected attribute `menu` in `PC` and this method was refined six times to add items to the `menu`, which is directly accessed into the refinements. That is, changes in the refinements may occur if `Maler.jak` is refactored regarding `menu`.

4. Conclusion

Several tools have been developed to measure properties of software, and measures-based strategies have been proposed to detect code smells in OO and AO software. We developed Variability Smell Detection (VSD) tool to measure FO software and detect specific code smells in AHEAD-based SPLs. We used VSD with eight SPLs of different sizes, e.g. AHEAD-java with 16,719 lines of code; 963 components; and 838 refinements. In an empirical study, Abilio (2014) verified that the results of VSD detection are in agreement with results of a manual inspection performed by specialists. Therefore, it can save time in the analysis of methods and components, allows software engineers to have a notion of the feature implementation, and allows the identification of code smells. The first version of VSD only measures AHEAD-based SPL, but it can be expanded to measure SPLs developed with other techniques, such as AspectJ and FeatureHouse, and we only need to parse the code to VSD structure and adapt the measures, if necessary. It can also implement strategies to detect variability smells related to feature models, for example. Therefore, our future goal is to improve VSD perform further empirical studies to evaluate it.

Acknowledges

This work was partially supported by Capes, CNPq (grant Universal 485907/2013-5) and FAPEMIG (grants APQ-02532-12 and PPM-00382-14).

References

Abilio, R. (2014) "Detecting Code Smells in Software Product Lines". Master's thesis, Federal University of Lavras (UFLA), 141p.

Apel, S.; Batory, D.; Kastner, C.; Saake, G. (2013) "Feature-Oriented Software Product Lines: Concepts and Implementation". Springer, 315p.

Apel, S.; Beyer, D. (2011) “Feature Cohesion in Software Product Lines: An Exploratory Study”. In: International Conf. on Software Engineering, pp. 421-430.

Batory, D.; Sarvela, J.; Rauschmayer, A. (2003) “Scaling Step-Wise Refinement”. In: 25th International Conference on Software Engineering, pp. 187-197.

Blonski, H.; Padilha, J.; Barbosa, M.; Santana, D.; Figueiredo, E. (2013) “ConcernMeBS: Metrics-based Detection of Code Smells”. In: Brazilian Conference on Software (CBSoft), Tools Session. Brasilia, Brazil, 2013.

Chidamber, S.; Kemerer, C. (1994) “A Metrics Suite for Object Oriented Design”. IEEE Transactions on Software Engineering, v. 20, n. 6, pp. 476-493.

Figueiredo, E.; Sant'Anna, C.; Garcia, A.; Lucena, C.et al. (2012) “Applying and Evaluating Concern-Sensitive Design Heuristics”. Journal of Systems and Software, v.85, n.2, pp. 227-243.

Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. (1999) “Refactoring: Improving the Design of Existing Code”. Addison Wesley, 464p.

Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S. (1990) “Feature-Oriented Domain Analysis (FODA) Feasibility Study”, Technical Report, SEI.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loigtier, J.; Irwin, J.; (1997) “Aspect-Oriented Programming”. In ECOOP’97, pp.220-242.

Lanza, M.; Marinescu, R. (2006) “Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems”. Springer, 205p.

Macia, I.; Garcia, A.; Staa, A. von. (2010) “Defining and Applying Detection Strategies for Aspect-Oriented Code Smells”. In: 24th Brazilian Symposium on Software Engineering, pp. 60-69.

Marinescu, R. (2004) “Detection Strategies: Metrics-Based Rules for Detecting Design Flaws”. In: International Conference on Software Maintenance, pp. 350-359.

McCabe, T. J. (1976) “A Complexity Measure”. IEEE Transactions on Software Engineering, v. 2, n. 4, pp. 308-320.

Padilha, J.; Figueiredo, E.; Sant'Anna, C.; Garcia, A. (2013) “Detecting God Methods with Concern Metrics: An Exploratory Study”. In: 7th Latin-American Workshop on Aspect-Oriented Software Development, pp. 24-29.

Padilha, J.; Pereira, J.; Figueiredo, E.; Almeida, J.; Garcia, A.; Sant’Anna, C. (2014) “On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study”. In: International Conference on Advanced Information Systems Engineering.

Pohl, K.; Bockle, G.; Linden, F. J. van der. (2005) “Software Product Line Engineering: Foundations, Principles, and Techniques”. Springer, 490p.

Prehofer, C. (1997) “Feature-Oriented Programming: A Fresh Look at Objects”. In: European Conference of Object-Oriented Programming, pp.419-443.

Schulze, S.; Apel, S.; Kastner, C. (2010) “Code Clones in Feature-Oriented Software Product Lines”. In: 9th International Conference on Generative Programming and Component Engineering, pp. 103-112.

Schulze, S.; Thüm, T.; Kuhlemann, M.; Saake, G. (2012) “Variant-Preserving Refactoring in Feature-Oriented Software Product Lines”. In: 6th Workshop on Variability Modeling of Software-Intensive Systems, pp. 73-81.

Thüm, T.; Kästner, C.; Benduhn, F.; Meinicke, J.; Saake, G.; Leich, T. (2014) “FeatureIDE: An Extensible Framework for Feature-Oriented Software Development”. Science of Computer Programming, v.79, pp. 70-85.